Contents

- Azure Cosmos DB documentation
- Overview
  - Welcome to Azure Cosmos DB
  - NoSQL vs. relational databases
- Concepts
  - Azure Cosmos DB resource model
  - Global distribution
    - Global distribution overview
    - Consistency levels
    - High availability
    - Conflict types and resolution policies
    - Global distribution - under the hood
  - Partitioning and horizontal scaling
  - Capacity
    - Request units
    - Provisioned throughput
    - Autoscale throughput
      - Autoscale introduction
      - Autoscale FAQ
    - Serverless
    - Choose between autoscale and standard (manual) throughput
    - Choose between provisioned throughput and serverless
  - Security
    - Security overview
    - Data encryption
    - Service tags
    - Azure role-based access control
    - Azure Policy support
    - Security controls by Azure Policy

- Security baseline
- Analytics with Azure Synapse Link
  - Azure Synapse Link for Cosmos DB
  - Analytical store - Overview
  - Synapse Link supported features
  - Synapse Link use cases
  - Synapse Link FAQ
- Optimize your Azure Cosmos DB cost
  - Plan and manage costs
  - Pricing model
  - Total Cost of Ownership (TCO)
  - Understand your bill
  - Optimize provisioned throughput cost
  - Optimize request cost
  - Optimize storage cost
  - Optimize multi-region cost
  - Optimize development/testing cost
  - Optimize cost with reserved capacity
- Change feed overview
- Built-in Jupyter notebooks
- Service quotas
- Enterprise readiness - Compliance
- Automatic recommendations

SQL/Core API

- Quickstarts
  - Create Cosmos DB resources
    - Azure portal
    - ARM template
    - .NET app - V4 SDK
    - .NET app - V3 SDK
    - Java app
    - Spring Data app

Node.js app
Python app
Xamarin app
Tutorials
1 - Create and manage data
Build a console app
.NET
Java
Async Java
Node.js
Build a web app
.NET
Java
Node.js
Build a mobile app
Xamarin
2 - Migrate data
Using the Data Migration tool
Using .NET bulk support
3 - Query data
4 - Create a notebook
5 - Distribute data globally
Samples
.NET V2 SDK samples
.NET V3 SDK samples
Java SDK v4 samples
Spring Data v3 samples
Node.js samples
Python samples
PowerShell
Azure CLI
Azure Resource Manager

- Concepts
  - Modeling and partitioning
    - Model document data
    - Modeling and partitioning - a real-world example
    - Synthetic partition key
  - SQL query
    - Getting started
    - Clauses
      - SELECT
      - FROM
      - WHERE
      - ORDER BY
      - GROUP BY
      - OFFSET LIMIT
    - Working with JSON
    - Subquery
    - Joins
    - Arrays and objects
    - Keywords
    - Constants
    - Operators
    - Scalar expressions
    - Functions
      - User-defined functions
    - Aggregate functions
      - Aggregate functions
      - Avg
      - Count
      - Max
      - Min
      - Sum
    - System functions

System functions
Array functions
  Array functions
  ARRAY_CONCAT
  ARRAY_CONTAINS
  ARRAY_LENGTH
  ARRAY_SLICE
Date and time functions
  Date and time functions
  DateTimeAdd
  DateTimeDiff
  DateTimeFromParts
  DateTimePart
  DateTimeToTicks
  DateTimeToTimestamp
  GetCurrentDateTime
  GetCurrentTicks
  GetCurrentTimestamp
  TicksToDateTime
  TimestampToDateTime
Mathematical functions
  Mathematical functions
  ABS
  ACOS
  ASIN
  ATAN
  ATN2
  CEILING
  COS
  COT
  DEGREES
  EXP

FLOOR
LOG
LOG10
PI
POWER
RADIANS
RAND
ROUND
SIGN
SIN
SQRT
SQUARE
TAN
TRUNC
Spatial functions
Spatial functions
ST_DISTANCE
ST_WITHIN
ST_INTERSECTS
ST_ISVALID
ST_ISVALIDDETAILED
String functions
String functions
CONCAT
CONTAINS
ENDSWITH
INDEX_OF
LEFT
LENGTH
LOWER
LTRIM
REGEXMATCH

REPLACE
REPLICATE
REVERSE
RIGHT
RTRIM
STARTSWITH
STRINGEQUALS
StringToArray
StringToBoolean
StringToNull
StringToNumber
StringToObject
SUBSTRING
ToString
TRIM
UPPER
Type checking functions
Type checking functions
IS_ARRAY
IS_BOOL
IS_DEFINED
IS_NULL
IS_NUMBER
IS_OBJECT
IS_PRIMITIVE
IS_STRING
Geospatial data
Geospatial overview
Query geospatial data
Index geospatial data
DateTime
Parameterized query

- LINQ to SQL
- Containers and items
  - Indexing overview
  - Time to live
  - Unique key constraints
  - Transactions and optimistic concurrency control
  - Attachments
- Change feed
  - Change feed design patterns
  - Reading change feed
  - Change feed processor
  - Trigger Azure Functions
  - Change feed pull model
- Server-side programming
  - Stored procedures, triggers, and UDFs
  - JavaScript query API
- Security
  - Secure access to data
  - Advanced threat protection
- Connectivity modes
- How-to guides
  - Manage Azure Cosmos DB resources
    - Manage an Azure Cosmos account
    - Using PowerShell
    - Using Azure CLI
    - Using Resource Manager templates
    - Move between regions
    - Prevent changes or deletion
  - Global distribution
    - Configure multi-region writes
    - Manage consistency levels
    - Configure conflict resolution policies

- Convert session token format
- Capacity
  - Provisioned throughput
    - Provision standard (manual) throughput - container
    - Provision standard (manual) throughput - database
    - Provision autoscale throughput - database or container
    - Estimate RU/s with Cosmos capacity planner
    - Scale using Azure Functions timer
  - Find request unit charge
- Operations on containers and items
  - Create a container - SQL API
  - Create a container with large partition key
  - Query a container
  - Migrate to partitioned containers
  - Configure time to live
  - Define unique keys
  - Perform transactional batch operations
- Performance
  - Query
    - Get query execution metrics
    - Tune query performance
  - Indexing
    - Indexing policies
    - Manage indexing policies
  - .NET
    - Performance tips for .NET SDK v3
    - Performance tips for .NET SDK v2
  - Java
    - Performance tips for Java SDK v4
    - Async Java SDK v2
    - Sync Java SDK v2
  - Performance test - sample app

- Cost-effective reads and writes
- Migrate to newer SDKs
  - Migrate to Java V4 SDK
  - Migrate to .NET V3 SDK
  - Migrate to .NET V2 SDK
- Troubleshooting
  - Common errors
    - Troubleshoot query performance
    - Troubleshoot service request timeout
    - Troubleshoot request rate too large
    - Troubleshoot not found
    - Troubleshoot unauthorized
    - Availability issues in multiregional environments
  - .NET
    - General Troubleshooting .NET SDK
    - Troubleshoot request header too large
    - Troubleshoot request timeout
    - Troubleshoot service unavailable
  - .Java
    - Troubleshoot Java SDK v4
    - Troubleshoot request timeout
    - Troubleshoot service unavailable
    - Troubleshoot- Async Java SDK v2
- Change feed
  - Azure Cosmos DB Trigger advanced configuration
  - Multiple Azure Cosmos DB Triggers
  - Troubleshoot Azure Cosmos DB Triggers
  - Using the change feed estimator
  - Migrate from change feed processor library
  - Change feed - Async Java sample
- Server-side programming
  - Write stored procedures, triggers, and UDFs

- Write stored procedures and triggers with JavaScript query API
- Use stored procedures, triggers, and UDFs
- Built-in Jupyter notebooks
  - Enable notebooks
  - Import notebooks from GitHub
  - Use built-in Python notebook commands
  - Use built-in C# notebook commands
- Bulk executor library
  - About bulk executor library
  - Bulk executor - .NET library
  - Migrate from bulk executor to .NET SDK
  - Bulk executor - Java library
- Visualize Cosmos DB data
  - Use Power BI
  - Create a live weather dashboard
  - Use Qlik Sense
- Built-in analytics with Apache Spark
  - Azure Databricks Spark connector
  - Streaming with Apache Kafka and Cosmos DB
- Migrate data to Cosmos DB
  - Migrate to SQL API using Striim
  - Migrate one-to-few relational data
  - Migrate - Couchbase to SQL API
  - Migrate - DynamoDB to SQL API
- Security
  - Configure Cross Origin Resource Sharing(CORS)
  - Use managed identities to access data
  - Use certificates to access data
- Integrate with other Azure services
  - Change feed Ecommerce solution
  - Azure App Service
  - Azure Cognitive Search indexer

- Azure Functions
  - Serverless computing
  - Azure Function bindings
- Azure Data Factory
- Azure Stream Analytics
- Azure Data Explorer
- Azure Event Hubs and Azure Storage
- Use Spring Boot Starter
- Spring Data developer guide
- ODBC driver

Reference

- .NET SDKs
  - .NET SDK v3
  - .NET SDK v2
  - .NET Core SDK v2
  - .NET change feed library v2
  - .NET bulk executor library v2
- Java SDKs and Connectors
  - Java SDK v4
  - Async Java SDK v2
  - Sync Java SDK v2
  - Spring Data Connector v2
  - Spring Data Connector v3
  - Spark Connector
  - Java bulk executor library v2
- Node.js
- Python
- REST (Data plane)
- REST (Management plane)
- PowerShell
- Resource Manager template
- Azure Policy built-ins

Cassandra API
- Overview
  - What is Cassandra API in Cosmos DB?
  - Wire protocol support
- Quickstarts
  - .NET
  - .NET Core
  - Java v3
  - Java v4
  - Node.js
  - Python
  - Golang
- Tutorials
  - 1 - Create and manage data
  - 2 - Load data
  - 3 - Query data
  - 4 - Migrate data
- Concepts
  - FAQ
  - Elastic scale
  - Partitioning
  - Secondary Indexes
  - Mapping consistency levels
- How-to guides
  - Migrate to Cassandra API
    - Migrate - Oracle DB to Cassandra API using Striim
    - Migrate - Oracle DB to Cassandra API using Blitzz
    - Migrate - Apache Cassandra to Cassandra API using Blitzz
    - Migrate - Apache Cassandra to Cassandra API using Databricks
  - Change feed
  - Create a container - Cassandra API
  - Provision throughput- Cassandra API

- Find request unit charge
- Store and manage Spring Data
- Cassandra and Spark
  - Introduction
  - Connect using Databricks
  - Connect using HDInsight
  - Create keyspace and table
  - Insert data
  - Read data
  - Upsert data
  - Delete data
  - Aggregation operations
  - Copy table data
- Manage using Resource Manager templates
- Manage using PowerShell
- Manage using Azure CLI
- Reference
  - REST resource provider
  - PowerShell
  - Resource Manager template
- MongoDB API
- Overview
  - What is MongoDB API in Cosmos DB?
- Quickstarts
  - .NET
  - Java
  - Node.js
  - Python
  - Xamarin
  - Golang
- Tutorials
  - 1 - Create and manage data

- Node.js console app
- Node.js and Angular app
  - Part 1 - Introduction
  - Part 2 - Create Node app
  - Part 3 - Add UI with Angular
  - Part 4 - Create an account
  - Part 5 - Connect to Cosmos DB
  - Part 6 - Perform CRUD operations
- Node.js and React app
- 2 - Query data
- 3 - Distribute data globally

Concepts

- Wire protocol support (3.6)
- Wire protocol support (3.2)
- FAQ
- Mapping consistency levels

How-to guides

- Migrate to MongoDB API
  - Pre-migration guide
  - Database Migration Service (online)
  - Database Migration Service (offline)
  - Post-migration guide
- Upgrade from MongoDB 3.2 to 3.6
- Get the connection string
- Connect using Studio 3T
- Connect using Compass
- Connect using Robo 3T
- Connect using Mongoose
- Distribute reads globally
- Time to live - MongoDB
- Change streams
- Manage data indexing

- Create a container - MongoDB API
- Provision throughput- MongoDB API
- Find request unit charge
- Troubleshoot query performance
- Store and manage Spring Data
- MongoDB extension commands
- Manage using Resource Manager templates
- Manage using PowerShell
- Manage using Azure CLI
- Troubleshoot common issues
- Reference
  - REST resource provider
  - PowerShell
  - Resource Manager template
- Gremlin API
  - Overview
    - What is Gremlin API in Cosmos DB?
    - Wire protocol support
  - Quickstarts
    - Gremlin console
    - .NET
    - Java
    - Node.js
    - Python
    - PHP
  - Tutorials
    - 1 - Query data
  - Concepts
    - FAQ
    - Response headers
    - Limits
  - How-to guides

- Graph data modeling
- Graph data partitioning
- Import graph data
- Optimize Gremlin queries
- Create a container - Gremlin API
- Provision throughput- Gremlin API
- Find request unit charge
- Use resource tokens
- Use regional endpoints
- Access system document properties
- Visualize graph data
- Store and manage Spring Data
- Manage using Resource Manager templates
- Manage using PowerShell
- Manage using Azure CLI
- Reference
  - Response Headers
  - PowerShell
  - REST resource provider
  - Resource Manager template
- Table API
- Overview
  - What is Table API in Cosmos DB?
- Quickstarts
  - .NET
  - Java
  - Node.js
  - Python
- Tutorials
  - 1 - Create and manage data
  - 2 - Migrate data
  - 3 - Query data

- 4 - Distribute data globally
- Samples
  - .NET
  - Java
  - Node.js
  - Python
  - PHP
  - C++
  - Ruby
  - F#
- Concepts
  - FAQ
- How-to guides
  - Build apps with Table API
  - Create a container - Table API
  - Find request unit charge
  - Table storage design guide
  - Manage using Resource Manager templates
  - Manage using PowerShell
  - Manage using Azure CLI
- Reference
  - .NET
  - .NET Standard
  - Java
  - Node.js
  - Python
  - PowerShell
  - Table storage REST APIs
- How-to guides
  - Use developer tools
    - Develop locally with emulator
      - Install and use the emulator

- Export emulator certificates
- Emulator command-line parameters
- Emulator troubleshooting
- Emulator release notes
- Visual Studio Code extension
- Set up Azure DevOps CI/CD pipeline
- Use Azure Storage Explorer
- Use Azure Cosmos DB Explorer
- Analytics and BI with Azure Synapse Link
  - Configure Azure Synapse Link
  - Connect to Azure Synapse Link
  - Query with Azure Synapse Spark
  - Query with serverless SQL pool
  - Use Power BI with Synapse Link
- Security
  - Configure IP firewall
  - Configure access from virtual networks
  - Configure access from private endpoints
  - Connect privately to a Cosmos DB account
  - Use Azure Key Vault to access data
  - Restrict user access to data operations only
  - Configure customer-managed keys
- Enterprise readiness
  - Online backup and on-demand restore
  - Request quota increase
- Monitor
  - Monitor Cosmos DB
  - Use Azure monitor metrics
    - Azure Monitor for Cosmos (preview)
    - Scenarios to monitor
    - Monitor server-side latency
    - Monitor request unit usage

Monitor normalized request unit usage
Use Azure monitor logs
Monitor with diagnostic logs
Audit control plane logs
Configure alerts
Monitoring data reference
Application logging with Logic Apps
Migrate data to Cosmos DB
Options to migrate data into Cosmos DB
Migrate hundreds of terabytes of data into Azure Cosmos DB
Resources
Build your skills with Microsoft Learn
Query cheat sheet
SQL playground
FAQ
Whitepapers
Partners
Microsoft Q&A question page
Videos
Azure Roadmap
Try Azure Cosmos DB for free
Pricing
Performance levels
Use cases
Common use cases
Social media apps

Welcome to Azure Cosmos DB

12/4/2020 • 3 minutes to read • Edit Online

APPLIES TO: SQL API Cassandra API Gremlin API Table API Azure Cosmos DB API for MongoDB

Today's applications are required to be highly responsive and always online. To achieve low latency and high availability, instances of these applications need to be deployed in datacenters that are close to their users. Applications need to respond in real time to large changes in usage at peak hours, store ever increasing volumes of data, and make this data available to users in milliseconds.

Azure Cosmos DB is a fully managed NoSQL database for modern app development. Single-digit millisecond response times, and automatic and instant scalability, guarantee speed at any scale. Business continuity is assured with SLA-backed availability and enterprise-grade security. App development is faster and more productive thanks to turnkey multi-master data distribution anywhere in the world, open source APIs and SDKs for popular languages. As a fully managed service, Azure Cosmos DB takes database administration off your hands with automatic management, updates and patching. It also handles capacity management with cost-effective serverless and automatic scaling options that respond to application needs to match capacity with demand.

You can Try Azure Cosmos DB for Free without an Azure subscription, free of charge and commitments or use the Azure Cosmos DB free tier to get an account with the first 400 RU/s and 5 GB of storage free.

Try Azure Cosmos DB for Free

Key Benefits

Guaranteed speed at any scale

Gain unparalleled SLA-backed speed and throughput, fast global access, and instant elasticity.

- Real-time access with fast read and write latencies globally, and throughput and consistency all backed by SLAs
- Multi-region writes and data distribution to any Azure region with the click of a button.
- Independently and elastically scale storage and throughput across any Azure region – even during unpredictable traffic bursts – for unlimited scale worldwide.

Simplified application development

Build fast with open source APIs, multiple SDKs, schemaless data and no-ETL analytics over operational data.

- Deeply integrated with key Azure services used in modern (cloud-native) app development including Azure Functions, IoT Hub, AKS (Azure Kubernetes Service), App Service, and more.
- Choose from multiple database APIs including the native Core (SQL) API, API for MongoDB, Cassandra API, Gremlin API, and Table API.
- Build apps on Core (SQL) API using the languages of your choice with SDKs for .NET, Java, Node.js and Python. Or your choice of drivers for any of the other database APIs.
- Run no-ETL analytics over the near-real time operational data stored in Azure Cosmos DB with Azure Synapse Analytics.
- Change feed makes it easy to track and manage changes to database containers and create triggered events with Azure Functions.
- Azure Cosmos DB's schema-less service automatically indexes all your data, regardless of the data model, to delivery blazing fast queries.

Mission-critical ready

Guarantee business continuity, 99.999% availability, and enterprise-level security for every application.

- Azure Cosmos DB offers a comprehensive suite of SLAs including industry-leading availability worldwide.
- Easily distribute data to any Azure region with automatic data replication. Enjoy zero downtime with multi-region writes or RPO 0 when using Strong consistency.
- Enjoy enterprise-grade encryption-at-rest with self-managed keys.
- Azure role-based access control keeps your data safe and offers fine-tuned control.

Fully managed and cost-effective

End-to-end database management, with serverless and automatic scaling matching your application and TCO needs

- Fully-managed database service. Automatic, no touch, maintenance, patching, and updates, saving developers time and money.
- Cost-effective options for unpredictable or sporadic workloads of any size or scale, enabling developers to get started easily without having to plan or manage capacity.
- Serverless model offers spiky workloads automatic and responsive service to manage traffic bursts on demand.
- Autoscale provisioned throughput automatically and instantly scales capacity for unpredictable workloads, while maintaining SLAs.

Solutions that benefit from Azure Cosmos DB

Any web, mobile, gaming, and IoT application that needs to handle massive amounts of data, reads, and writes at a global scale with near-real response times for a variety of data will benefit from Cosmos DB's guaranteed high availability, high throughput, low latency, and tunable consistency. Learn about how Azure Cosmos DB can be used to build IoT and telematics, retail and marketing, gaming and web and mobile applications.

Next steps

Get started with Azure Cosmos DB with one of our quickstarts:

- Get started with Azure Cosmos DB SQL API
- Get started with Azure Cosmos DB's API for MongoDB
- Get started with Azure Cosmos DB Cassandra API
- Get started with Azure Cosmos DB Gremlin API
- Get started with Azure Cosmos DB Table API

Try Azure Cosmos DB for free

Understanding the differences between NoSQL and relational databases

11/2/2020 • 8 minutes to read • Edit Online

APPLIES TO: SQL API Cassandra API Gremlin API Table API Azure Cosmos DB API for MongoDB

This article will enumerate some of the key benefits of NoSQL databases over relational databases. We will also discuss some of the challenges in working with NoSQL. For an in-depth look at the different data stores that exist, have a look at our article on choosing the right data store.

High throughput

One of the most obvious challenges when maintaining a relational database system is that most relational engines apply locks and latches to enforce strict ACID semantics. This approach has benefits in terms of ensuring a consistent data state within the database. However, there are heavy trade-offs with respect to concurrency, latency, and availability. Due to these fundamental architectural restrictions, high transactional volumes can result in the need to manually shard data. Implementing manual sharding can be a time consuming and painful exercise.

In these scenarios, distributed databases can offer a more scalable solution. However, maintenance can still be a costly and time-consuming exercise. Administrators may have to do extra work to ensure that the distributed nature of the system is transparent. They may also have to account for the "disconnected" nature of the database.

Azure Cosmos DB simplifies these challenges, by being deployed worldwide across all Azure regions. Partition ranges are capable of being dynamically subdivided to seamlessly grow the database in line with the application, while simultaneously maintaining high availability. Fine-grained multi-tenancy and tightly controlled, cloud-native resource governance facilitates astonishing latency guarantees and predictable performance. Partitioning is fully managed, so administrators need not have to write code or manage partitions.

If your transactional volumes are reaching extreme levels, such as many thousands of transactions per second, you should consider a distributed NoSQL database. Consider Azure Cosmos DB for maximum efficiency, ease of maintenance, and reduced total cost of ownership.

Hierarchical data

There are a significant number of use cases where transactions in the database can contain many parent-child relationships. These relationships can grow significantly over time, and prove difficult to manage. Forms of hierarchical databases did emerge during the 1980s, but were not popular due to inefficiency in storage. They also lost traction as Ted Codd's relational model became the de facto standard used by virtually all mainstream database management systems.

However, today the popularity of document-style databases has grown significantly. These databases might be considered a reinventing of the hierarchical database paradigm, now uninhibited by concerns around the cost of storing data on disk. As a result, maintaining many complex parent-child entity relationships in a relational database could now be considered an anti-pattern compared to modern document-oriented approaches.

The emergence of object oriented design, and the impedance mismatch that arises when combining it with relational models, also highlights an anti-pattern in relational databases for certain use cases. Hidden but often significant maintenance costs can arise as a result. Although ORM approaches have evolved to partly mitigate this, document-oriented databases nonetheless coalesce much better with object-oriented approaches. With this approach, developers are not forced to be committed to ORM drivers, or bespoke language specific OO Database engines. If your data contains many parent-child relationships and deep levels of hierarchy, you may want to consider using a NoSQL document database such as the Azure Cosmos DB SQL API.

Complex networks and relationships

Ironically, given their name, relational databases present a less than optimal solution for modeling deep and complex relationships. The reason for this is that relationships between entities do not actually exist in a relational database. They need to be computed at runtime, with complex relationships requiring cartesian joins in order to allow mapping using queries. As a result, operations become exponentially more expensive in terms of computation as relationships increase. In some cases, a relational database attempting to manage such entities will become unusable.

Various forms of "Network" databases did emerge during the time that relational databases emerged, but as with hierarchical databases, these systems struggled to gain popularity. Slow adoption was due to a lack of use cases at the time, and storage inefficiencies. Today, graph database engines could be considered a re-emergence of the network database paradigm. The key benefit with these systems is that relationships are stored as "first class citizens" within the database. Thus, traversing relationships can be done in constant time, rather than increasing in time complexity with each new join or cross product.

If you are maintaining a complex network of relationships in your database, you may want to consider a graph

database such as the Azure Cosmos DB Gremlin API for managing this data.

Azure Cosmos DB is a multi-model database service, which offers an API projection for all the major NoSQL model types; Column-family, Document, Graph, and Key-Value. The Gremlin (graph) and SQL (Core) Document API layers are fully interoperable. This has benefits for switching between different models at the programmability level. Graph stores can be queried in terms of both complex network traversals as well as transactions modeled as document records in the same store.

Fluid schema

Another particular characteristic of relational databases is that schemas are required to be defined at design time. This has benefits in terms of referential integrity and conformity of data. However, it can also be restrictive as the application grows. Responding to changes in the schema across logically separate models sharing the same table or database definition can become complex over time. Such use cases often benefit from the schema being devolved to the application to manage on a per record basis. This requires the database to be "schema agnostic" and allow records to be "self-describing" in terms of the data contained within them.

If you are managing data whose structures are constantly changing at a high rate, particularly if transactions can come from external sources where it is difficult to enforce conformity across the database, you may want to consider a more schema-agnostic approach using a managed NoSQL database service like Azure Cosmos DB.

Microservices

The microservices pattern has grown significantly in recent years. This pattern has its roots in Service-Oriented Architecture. The de-facto standard for data transmission in these modern microservices architectures is JSON, which also happens to be the storage medium for the vast majority of document-oriented NoSQL Databases. This makes NoSQL document stores a much more seamless fit for both the persistence and synchronization (using event sourcing patterns) across complex Microservice implementations. More traditional relational databases can be much more complex to maintain in these architectures. This is due to the greater amount of transformation required for both state and synchronization across APIs. Azure Cosmos DB in particular has a number of features that make it an even more seamless fit for JSON-based Microservices Architectures than many NoSQL databases:

- a choice of pure JSON data types
- a JavaScript engine and query API built into the database.
- a state-of-the-art change feed which clients can subscribe to in order to get notified of modifications to a

container.

Some challenges with NoSQL databases

Although there are some clear advantages when implementing NoSQL databases, there are also some challenges that you may want to take into consideration. These may not be present to the same degree when working with the relational model:

- transactions with many relations pointing to the same entity.
- transactions requiring strong consistency across the entire dataset.

Looking at the first challenge, the rule-of-thumb in NoSQL databases is generally denormalization, which as articulated earlier, produces more efficient reads in a distributed system. However, there are some design challenges that come into play with this approach. Let's take an example of a product that's related to one category and multiple tags:

A best practice approach in a NoSQL document database would be to denormalize the category name and tag names directly in a "product document". However, in order to keep categories, tags, and products in sync, the design options to facilitate this have added maintenance complexity, because the data is duplicated across multiple records in the product, rather than being a simple update in a "one-to-many" relationship, and a join to retrieve the data.

The trade-off is that reads are more efficient in the denormalized record, and become increasingly more efficient as the number of conceptually joined entities increases. However, just as the read efficiency increases with increasing numbers of joined entities in a denormalize record, so too does the maintenance complexity of keeping entities in sync. One way of mitigating this trade-off is to create a hybrid data model.

While there is more flexibility available in NoSQL databases to deal with these trade-offs, increased flexibility can also produce more design decisions. Consult our article how to model and partition data on Azure Cosmos DB using a real-world example, which includes an approach for keeping denormalized user data in sync where users not only sit in different partitions, but in different containers.

With respect to strong consistency, it is rare that this will be required across the entire data set. However, in cases where this is necessary, it can be a challenge in distributed databases. To ensure strong consistency, data needs to be synchronized across all replicas and regions before allowing clients to read it. This can increase the latency of reads.

Again, Azure Cosmos DB offers more flexibility than relational databases for the various trade-offs that are relevant here, but for small scale implementations, this approach may add more design considerations. Consult our article on Consistency, availability, and performance tradeoffs for more detail on this topic.

Next steps

Learn how to manage your Azure Cosmos account and other concepts:

- How-to manage your Azure Cosmos account
- Global distribution
- Consistency levels
- Working with Azure Cosmos containers and items
- VNET service endpoint for your Azure Cosmos account
- IP-firewall for your Azure Cosmos account
- How-to add and remove Azure regions to your Azure Cosmos account
- Azure Cosmos DB SLAs

Azure Cosmos DB resource model

12/4/2020 • 9 minutes to read • Edit Online

APPLIES TO: SQL API Cassandra API Gremlin API Table API Azure Cosmos DB API for MongoDB

Azure Cosmos DB is a fully managed platform-as-a-service (PaaS). To begin using Azure Cosmos DB, you should initially create an Azure Cosmos account in your Azure subscription and databases, containers, items under it. This article describes the Azure Cosmos DB resource model and different entities in the resource model hierarchy.

The Azure Cosmos account is the fundamental unit of global distribution and high availability. Your Azure Cosmos account contains a unique DNS name and you can manage an account by using Azure portal, Azure CLI or by using different language-specific SDKs. For more information, see how to manage your Azure Cosmos account. For globally distributing your data and throughput across multiple Azure regions, you can add and remove Azure regions to your account at any time. You can configure your account to have either a single region or multiple write regions. For more information, see how to add and remove Azure regions to your account. You can configure the default consistency level on an account.

Elements in an Azure Cosmos account

Azure Cosmos container is the fundamental unit of scalability. You can virtually have an unlimited provisioned throughput (RU/s) and storage on a container. Azure Cosmos DB transparently partitions your container using the logical partition key that you specify in order to elastically scale your provisioned throughput and storage.

Currently, you can create a maximum of 50 Azure Cosmos accounts under an Azure subscription (this is a soft limit that can be increased via support request). A single Azure Cosmos account can virtually manage unlimited amount of data and provisioned throughput. To manage your data and provisioned throughput, you can create one or more Azure Cosmos databases under your account and within that database, you can create one or more containers. The following image shows the hierarchy of elements in an Azure Cosmos account:

After you create an account under your Azure subscription, you can manage the data in your account by

creating databases, containers, and items.

The following image shows the hierarchy of different entities in an Azure Cosmos DB account:

Azure Cosmos databases

You can create one or multiple Azure Cosmos databases under your account. A database is analogous to a namespace. A database is the unit of management for a set of Azure Cosmos containers. The following table shows how a database is mapped to various API-specific entities:

AZURE COSMOS ENTITY	SQL API	CASSANDRA API	AZURE COSMOS DB API FOR MONGODB	GREMLIN API	TABLE API
Azure Cosmos database	Database	Keyspace	Database	Database	NA

> **NOTE**
> With Table API accounts, when you create your first table, a default database is automatically created in your Azure Cosmos account.

Operations on an Azure Cosmos database

You can interact with an Azure Cosmos database with Azure Cosmos APIs as described in the following table:

OPERATION	AZURE CLI	SQL API	CASSANDRA API	AZURE COSMOS DB API FOR MONGODB	GREMLIN API	TABLE API
Enumerate all databases	Yes	Yes	Yes (database is mapped to a keyspace)	Yes	NA	NA
Read database	Yes	Yes	Yes (database is mapped to a keyspace)	Yes	NA	NA
Create new database	Yes	Yes	Yes (database is mapped to a keyspace)	Yes	NA	NA

OPERATION	AZURE CLI	SQL API	CASSANDRA API	AZURE COSMOS DB API FOR MONGODB	GREMLIN API	TABLE API
Update database	Yes	Yes	Yes (database is mapped to a keyspace)	Yes	NA	NA

Azure Cosmos containers

An Azure Cosmos container is the unit of scalability both for provisioned throughput and storage. A container is horizontally partitioned and then replicated across multiple regions. The items that you add to the container are automatically grouped into logical partitions, which are distributed across physical partitions, based on the partition key. The throughput on a container is evenly distributed across the physical partitions. To learn more about partitioning and partition keys, see Partition data.

When you create a container, you configure throughput in one of the following modes:

- **Dedicated provisioned throughput mode**: The throughput provisioned on a container is exclusively reserved for that container and it is backed by the SLAs. To learn more, see How to provision throughput on a container.
- **Shared provisioned throughput mode**: These containers share the provisioned throughput with the other containers in the same database (excluding containers that have been configured with dedicated provisioned throughput). In other words, the provisioned throughput on the database is shared among all the "shared throughput" containers. To learn more, see How to provision throughput on a database.

> **NOTE**
> You can configure shared and dedicated throughput only when creating the database and container. To switch from dedicated throughput mode to shared throughput mode (and vice versa) after the container is created, you have to create a new container and migrate the data to the new container. You can migrate the data by using the Azure Cosmos DB change feed feature.

An Azure Cosmos container can scale elastically, whether you create containers by using dedicated or shared provisioned throughput modes.

A container is a schema-agnostic container of items. Items in a container can have arbitrary schemas. For example, an item that represents a person and an item that represents an automobile can be placed in the *same container*. By default, all items that you add to a container are automatically indexed without requiring explicit index or schema management. You can customize the indexing behavior by configuring the indexing policy on a container.

You can set Time to Live (TTL) on selected items in a container or for the entire container to gracefully purge those items from the system. Azure Cosmos DB automatically deletes the items when they expire. It also guarantees that a query performed on the container doesn't return the expired items within a fixed bound. To learn more, see Configure TTL on your container.

You can use change feed to subscribe to the operations log that is managed for each logical partition of your container. Change feed provides the log of all the updates performed on the container, along with the before and after images of the items. For more information, see Build reactive applications by using change feed. You can also configure the retention duration for the change feed by using the change feed policy on the container.

You can register stored procedures, triggers, user-defined functions (UDFs), and merge procedures for your container.

You can specify a unique key constraint on your Azure Cosmos container. By creating a unique key policy, you ensure the uniqueness of one or more values per logical partition key. If you create a container by using a unique key policy, no new or updated items with values that duplicate the values specified by the unique key constraint can be created. To learn more, see Unique key constraints.

A container is specialized into API-specific entities as shown in the following table:

AZURE COSMOS ENTITY	SQL API	CASSANDRA API	AZURE COSMOS DB API FOR MONGODB	GREMLIN API	TABLE API
Azure Cosmos container	Container	Table	Collection	Graph	Table

> **NOTE**
> When creating containers, make sure you don't create two containers with the same name but different casing. That's because some parts of the Azure platform are not case-sensitive, and this can result in confusion/collision of telemetry and actions on containers with such names.

Properties of an Azure Cosmos container

An Azure Cosmos container has a set of system-defined properties. Depending on which API you use, some properties might not be directly exposed. The following table describes the list of system-defined properties:

SYSTEM-DEFINED PROPERTY	SYSTEM-GENERATED OR USER-CONFIGURABLE	PURPOSE	SQL API	CASSANDRA API	AZURE COSMOS DB API FOR MONGODB	GREMLIN API	TABLE API
_rid	System-generated	Unique identifier of container	Yes	No	No	No	No
_etag	System-generated	Entity tag used for optimistic concurrency control	Yes	No	No	No	No
_ts	System-generated	Last updated timestamp of the container	Yes	No	No	No	No
_self	System-generated	Addressable URI of the container	Yes	No	No	No	No
id	User-configurable	User-defined unique name of the container	Yes	Yes	Yes	Yes	Yes

SYSTEM-DEFINED PROPERTY	SYSTEM-GENERATED OR USER-CONFIGURABLE	PURPOSE	SQL API	CASSANDRA API	AZURE COSMOS DB API FOR MONGODB	GREMLIN API	TABLE API
indexingPolicy	User-configurable	Provides the ability to change the index path, index type, and index mode	Yes	No	No	No	Yes
TimeToLive	User-configurable	Provides the ability to delete items automatically from a container after a set time period. For details, see Time to Live.	Yes	No	No	No	Yes
changeFeedPolicy	User-configurable	Used to read changes made to items in a container. For details, see Change feed.	Yes	No	No	No	Yes
uniqueKeyPolicy	User-configurable	Used to ensure the uniqueness of one or more values in a logical partition. For more information, see Unique key constraints.	Yes	No	No	No	Yes

Operations on an Azure Cosmos container

An Azure Cosmos container supports the following operations when you use any of the Azure Cosmos APIs:

OPERATION	AZURE CLI	SQL API	CASSANDRA API	AZURE COSMOS DB API FOR MONGODB	GREMLIN API	TABLE API
Enumerate containers in a database	Yes	Yes	Yes	Yes	NA	NA
Read a container	Yes	Yes	Yes	Yes	NA	NA
Create a new container	Yes	Yes	Yes	Yes	NA	NA
Update a container	Yes	Yes	Yes	Yes	NA	NA
Delete a container	Yes	Yes	Yes	Yes	NA	NA

Azure Cosmos items

Depending on which API you use, an Azure Cosmos item can represent either a document in a collection, a row in a table, or a node or edge in a graph. The following table shows the mapping of API-specific entities to an Azure Cosmos item:

COSMOS ENTITY	SQL API	CASSANDRA API	AZURE COSMOS DB API FOR MONGODB	GREMLIN API	TABLE API
Azure Cosmos item	Item	Row	Document	Node or edge	Item

Properties of an item

Every Azure Cosmos item has the following system-defined properties. Depending on which API you use, some of them might not be directly exposed.

SYSTEM-DEFINED PROPERTY	SYSTEM-GENERATED OR USER-CONFIGURABLE	PURPOSE	SQL API	CASSANDRA API	AZURE COSMOS DB API FOR MONGODB	GREMLIN API	TABLE API
_rid	System-generated	Unique identifier of the item	Yes	No	No	No	No
_etag	System-generated	Entity tag used for optimistic concurrency control	Yes	No	No	No	No

SYSTEM-DEFINED PROPERTY	SYSTEM-GENERATED OR USER-CONFIGURABLE	PURPOSE	SQL API	CASSANDRA API	AZURE COSMOS DB API FOR MONGODB	GREMLIN API	TABLE API
_ts	System-generated	Timestamp of the last update of the item	Yes	No	No	No	No
_self	System-generated	Addressable URI of the item	Yes	No	No	No	No
id	Either	User-defined unique name in a logical partition.	Yes	Yes	Yes	Yes	Yes
Arbitrary user-defined properties	User-defined	User-defined properties represented in API-native representation (including JSON, BSON, and CQL)	Yes	Yes	Yes	Yes	Yes

NOTE

Uniqueness of the `id` property is only enforced within each logical partition. Multiple documents can have the same `id` property with different partition key values.

Operations on items

Azure Cosmos items support the following operations. You can use any of the Azure Cosmos APIs to perform the operations.

OPERATION	AZURE CLI	SQL API	CASSANDRA API	AZURE COSMOS DB API FOR MONGODB	GREMLIN API	TABLE API
Insert, Replace, Delete, Upsert, Read	No	Yes	Yes	Yes	Yes	Yes

Next steps

Learn how to manage your Azure Cosmos account and other concepts:

- How-to manage your Azure Cosmos account
- Global distribution
- Consistency levels
- VNET service endpoint for your Azure Cosmos account
- IP-firewall for your Azure Cosmos account
- How-to add and remove Azure regions to your Azure Cosmos account
- Azure Cosmos DB SLAs

Distribute your data globally with Azure Cosmos DB

12/4/2020 • 3 minutes to read • Edit Online

APPLIES TO: SQL API Cassandra API Gremlin API Table API Azure Cosmos DB API for MongoDB

Today's applications are required to be highly responsive and always online. To achieve low latency and high availability, instances of these applications need to be deployed in datacenters that are close to their users. These applications are typically deployed in multiple datacenters and are called globally distributed. Globally distributed applications need a globally distributed database that can transparently replicate the data anywhere in the world to enable the applications to operate on a copy of the data that's close to its users.

Azure Cosmos DB is a globally distributed database system that allows you to read and write data from the local replicas of your database. Azure Cosmos DB transparently replicates the data to all the regions associated with your Cosmos account. Azure Cosmos DB is a globally distributed database service that's designed to provide low latency, elastic scalability of throughput, well-defined semantics for data consistency, and high availability. In short, if your application needs fast response time anywhere in the world, if it's required to be always online, and needs unlimited and elastic scalability of throughput and storage, you should build your application on Azure Cosmos DB.

You can configure your databases to be globally distributed and available in any of the Azure regions. To lower the latency, place the data close to where your users are. Choosing the required regions depends on the global reach of your application and where your users are located. Cosmos DB transparently replicates the data to all the regions associated with your Cosmos account. It provides a single system image of your globally distributed Azure Cosmos database and containers that your application can read and write to locally.

With Azure Cosmos DB, you can add or remove the regions associated with your account at any time. Your application doesn't need to be paused or redeployed to add or remove a region. It continues to be highly available all the time because of the multi-homing capabilities that the service natively provides.

Key benefits of global distribution

Build global active-active apps. With its novel multi-region writes replication protocol, every region supports both writes and reads. The multi-region writes capability also enables:

- Unlimited elastic write and read scalability.
- 99.999% read and write availability all around the world.
- Guaranteed reads and writes served in less than 10 milliseconds at the 99th percentile.

By using the Azure Cosmos DB multi-homing APIs, your application is aware of the nearest region and can send requests to that region. The nearest region is identified without any configuration changes. As you add and remove regions to and from your Azure Cosmos account, your application does not need to be redeployed or paused, it continues to be highly available at all times.

Build highly responsive apps. Your application can perform near real-time reads and writes against all the regions you chose for your database. Azure Cosmos DB internally handles the data replication between regions with consistency level guarantees of the level you've selected.

Build highly available apps. Running a database in multiple regions worldwide increases the availability of a database. If one region is unavailable, other regions automatically handle application requests. Azure Cosmos DB offers 99.999% read and write availability for multi-region databases.

Maintain business continuity during regional outages. Azure Cosmos DB supports automatic failover

during a regional outage. During a regional outage, Azure Cosmos DB continues to maintain its latency, availability, consistency, and throughput SLAs. To help make sure that your entire application is highly available, Cosmos DB offers a manual failover API to simulate a regional outage. By using this API, you can carry out regular business continuity drills.

Scale read and write throughput globally. You can enable every region to be writable and elastically scale reads and writes all around the world. The throughput that your application configures on an Azure Cosmos database or a container is provisioned across all regions associated with your Azure Cosmos account. The provisioned throughput is guaranteed up by financially backed SLAs.

Choose from several well-defined consistency models. The Azure Cosmos DB replication protocol offers five well-defined, practical, and intuitive consistency models. Each model has a tradeoff between consistency and performance. Use these consistency models to build globally distributed applications with ease.

Next steps

Read more about global distribution in the following articles:

- Global distribution - under the hood
- How to configure multi-region writes in your applications
- Configure clients for multihoming
- Add or remove regions from your Azure Cosmos DB account
- Create a custom conflict resolution policy for SQL API accounts
- Programmable consistency models in Cosmos DB
- Choose the right consistency level for your application
- Consistency levels across Azure Cosmos DB APIs
- Availability and performance tradeoffs for various consistency levels

Consistency levels in Azure Cosmos DB

11/2/2020 • 12 minutes to read • Edit Online

APPLIES TO: SQL API Cassandra API Gremlin API Table API Azure Cosmos DB API for MongoDB

Distributed databases that rely on replication for high availability, low latency, or both, must make a fundamental tradeoff between the read consistency, availability, latency, and throughput as defined by the PACLC theorem. The linearizability of the strong consistency model is the gold standard of data programmability. But it adds a steep price from higher write latencies due to data having to replicate and commit across large distances. Strong consistency may also suffer from reduced availability (during failures) because data cannot replicate and commit in every region. Eventual consistency offers higher availability and better performance, but its more difficult to program applications because data may not be completely consistent across all regions.

Most commercially available distributed NoSQL databases available in the market today provide only strong and eventual consistency. Azure Cosmos DB offers five well-defined levels. From strongest to weakest, the levels are:

- *Strong*
- *Bounded staleness*
- *Session*
- *Consistent prefix*
- *Eventual*

Each level provides availability and performance tradeoffs. The following image shows the different consistency levels as a spectrum.

The consistency levels are region-agnostic and are guaranteed for all operations regardless of the region from which the reads and writes are served, the number of regions associated with your Azure Cosmos account, or whether your account is configured with a single or multiple write regions.

Consistency levels and Azure Cosmos DB APIs

Azure Cosmos DB provides native support for wire protocol-compatible APIs for popular databases. These include MongoDB, Apache Cassandra, Gremlin, and Azure Table storage. When using Gremlin API and Table API, the default consistency level configured on the Azure Cosmos account is used. For details on consistency level mapping between Cassandra API or the API for MongoDB and Azure Cosmos DB's consistency levels see, Cassandra API consistency mapping and API for MongoDB consistency mapping.

Scope of the read consistency

Read consistency applies to a single read operation scoped within a logical partition. The read operation can be issued by a remote client or a stored procedure.

Configure the default consistency level

You can configure the default consistency level on your Azure Cosmos account at any time. The default consistency level configured on your account applies to all Azure Cosmos databases and containers under that account. All reads and queries issued against a container or a database use the specified consistency level by default. To learn more, see how to configure the default consistency level. You can also override the default consistency level for a specific request, to learn more, see how to Override the default consistency level article.

Guarantees associated with consistency levels

Azure Cosmos DB guarantees that 100 percent of read requests meet the consistency guarantee for the consistency level chosen. The precise definitions of the five consistency levels in Azure Cosmos DB using the TLA+ specification language are provided in the azure-cosmos-tla GitHub repo.

The semantics of the five consistency levels are described here:

- **Strong**: Strong consistency offers a linearizability guarantee. Linearizability refers to serving requests concurrently. The reads are guaranteed to return the most recent committed version of an item. A client never sees an uncommitted or partial write. Users are always guaranteed to read the latest committed write.

  The following graphic illustrates the strong consistency with musical notes. After the data is written to the "West US 2" region, when you read the data from other regions, you get the most recent value:

- **Bounded staleness**: The reads are guaranteed to honor the consistent-prefix guarantee. The reads might lag behind writes by at most *"K"* versions (that is, "updates") of an item or by *"T"* time interval, whichever is reached first. In other words, when you choose bounded staleness, the "staleness" can be configured in two ways:

- The number of versions (*K*) of the item

- The time interval (*T*) reads might lag behind the writes

For a single region account, the minimum value of *K* and *T* is 10 write operations or 5 seconds. For multi-region accounts the minimum value of *K* and *T* is 100,000 write operations or 300 seconds.

Bounded staleness offers total global order outside of the "staleness window." When a client performs read operations within a region that accepts writes, the guarantees provided by bounded staleness

consistency are identical to those guarantees by the strong consistency. As the staleness window approaches for either time or updates, whichever is closer, the service will throttle new writes to allow replication to catch up and honor the consistency guarantee.

Inside the staleness window, Bounded Staleness provides the following consistency guarantees:

- Consistency for clients in the same region for an account with single write region = Strong
- Consistency for clients in different regions for an account with single write region = Consistent Prefix
- Consistency for clients writing to a single region for an account with multiple write regions = Consistent Prefix
- Consistency for clients writing to different regions for an account with multiple write regions = Eventual

  Bounded staleness is frequently chosen by globally distributed applications that expect low write latencies but require total global order guarantee. Bounded staleness is great for applications featuring group collaboration and sharing, stock ticker, publish-subscribe/queueing etc. The following graphic illustrates the bounded staleness consistency with musical notes. After the data is written to the "West US 2" region, the "East US 2" and "Australia East" regions read the written value based on the configured maximum lag time or the maximum operations:

- **Session**: Within a single client session reads are guaranteed to honor the consistent-prefix, monotonic reads, monotonic writes, read-your-writes, and write-follows-reads guarantees. This assumes a single "writer" session or sharing the session token for multiple writers.

Clients outside of the session performing writes will see the following guarantees:

- Consistency for clients in same region for an account with single write region = Consistent Prefix
- Consistency for clients in different regions for an account with single write region = Consistent Prefix
- Consistency for clients writing to a single region for an account with multiple write regions = Consistent Prefix
- Consistency for clients writing to multiple regions for a account with multiple write regions = Eventual

Session consistency is the most widely used consistency level for both single region as well as globally distributed applications. It provides write latencies, availability, and read throughput comparable to that of eventual consistency but also provides the consistency guarantees that suit the needs of applications written to operate in the context of a user. The following graphic illustrates the session consistency with musical notes. The "West US 2 writer" and the "West US 2 reader" are using the same session (Session A) so they both read the same data at the same time. Whereas the "Australia East" region is using "Session B" so, it receives data later but in the same order as the writes.

- **Consistent prefix**: Updates that are returned contain some prefix of all the updates, with no gaps. Consistent prefix consistency level guarantees that reads never see out-of-order writes.

If writes were performed in the order `A, B, C`, then a client sees either `A`, `A,B`, or `A,B,C`, but never out-of-order permutations like `A,C` or `B,A,C`. Consistent Prefix provides write latencies, availability, and read throughput comparable to that of eventual consistency, but also provides the order guarantees that suit the needs of scenarios where order is important.

Below are the consistency guarantees for Consistent Prefix:

- Consistency for clients in same region for an account with single write region = Consistent Prefix
- Consistency for clients in different regions for an account with single write region = Consistent Prefix
- Consistency for clients writing to a single region for an account with multiple write region = Consistent Prefix
- Consistency for clients writing to multiple regions for an account with multiple write region = Eventual

The following graphic illustrates the consistency prefix consistency with musical notes. In all the regions, the reads never see out of order writes:

- **Eventual**: There's no ordering guarantee for reads. In the absence of any further writes, the replicas eventually converge.
  Eventual consistency is the weakest form of consistency because a client may read the values that are older than the ones it had read before. Eventual consistency is ideal where the application does not require any ordering guarantees. Examples include count of Retweets, Likes, or non-threaded comments. The following graphic illustrates the eventual consistency with musical notes.

Consistency guarantees in practice

In practice, you may often get stronger consistency guarantees. Consistency guarantees for a read operation correspond to the freshness and ordering of the database state that you request. Read-consistency is tied to the ordering and propagation of the write/update operations.

If there are no write operations on the database, a read operation with **eventual**, **session**, or **consistent prefix** consistency levels is likely to yield the same results as a read operation with strong consistency level.

If your Azure Cosmos account is configured with a consistency level other than the strong consistency, you can find out the probability that your clients may get strong and consistent reads for your workloads by looking at the *Probabilistically Bounded Staleness* (PBS) metric. This metric is exposed in the Azure portal, to learn more, see Monitor Probabilistically Bounded Staleness (PBS) metric.

Probabilistic bounded staleness shows how eventual is your eventual consistency. This metric provides an insight into how often you can get a stronger consistency than the consistency level that you have currently configured on your Azure Cosmos account. In other words, you can see the probability (measured in milliseconds) of getting strongly consistent reads for a combination of write and read regions.

Consistency levels and latency

The read latency for all consistency levels is always guaranteed to be less than 10 milliseconds at the 99th percentile. The average read latency, at the 50th percentile, is typically 4 milliseconds or less.

The write latency for all consistency levels is always guaranteed to be less than 10 milliseconds at the 99th percentile. The average write latency, at the 50th percentile, is usually 5 milliseconds or less. Azure Cosmos accounts that span several regions and are configured with strong consistency are an exception to this guarantee.

Write latency and Strong consistency

For Azure Cosmos accounts configured with strong consistency with more than one region, the write latency is equal to two times round-trip time (RTT) between any of the two farthest regions, plus 10 milliseconds at the 99th percentile. High network RTT between the regions will translate to higher latency for Cosmos DB requests since strong consistency completes an operation only after ensuring that it has been committed to all regions within an account.

The exact RTT latency is a function of speed-of-light distance and the Azure networking topology. Azure networking doesn't provide any latency SLAs for the RTT between any two Azure regions, however it does publish Azure network round-trip latency statistics. For your Azure Cosmos account, replication latencies are displayed in the Azure portal. You can use the Azure portal (go to the Metrics blade, select Consistency tab) to monitor the replication latencies between various regions that are associated with your Azure Cosmos account.

> **IMPORTANT**
> Strong consistency for accounts with regions spanning more than 5000 miles (8000 kilometers) is blocked by default due to high write latency. To enable this capability please contact support.

Consistency levels and throughput

- For strong and bounded staleness, reads are done against two replicas in a four replica set (minority quorum) to provide consistency guarantees. Session, consistent prefix and eventual do single replica reads. The result is that, for the same number of request units, read throughput for strong and bounded staleness is half of the other consistency levels.
- For a given type of write operation, such as insert, replace, upsert, and delete, the write throughput for request units is identical for all consistency levels.

CONSISTENCY LEVEL	QUORUM READS	QUORUM WRITES
Strong	Local Minority	Global Majority
Bounded Staleness	Local Minority	Local Majority
Session	Single Replica (using session token)	Local Majority

CONSISTENCY LEVEL	QUORUM READS	QUORUM WRITES
Consistent Prefix	Single Replica	Local Majority
Eventual	Single Replica	Local Majority

> **NOTE**
> The RU/s cost of reads for Local Minority reads are twice that of weaker consistency levels because reads are made from two replicas to provide consistency guarantees for Strong and Bounded Staleness.

Consistency levels and data durability

Within a globally distributed database environment there is a direct relationship between the consistency level and data durability in the presence of a region-wide outage. As you develop your business continuity plan, you need to understand the maximum acceptable time before the application fully recovers after a disruptive event. The time required for an application to fully recover is known as **recovery time objective** (RTO). You also need to understand the maximum period of recent data updates the application can tolerate losing when recovering after a disruptive event. The time period of updates that you might afford to lose is known as **recovery point objective** (RPO).

The table below defines the relationship between consistency model and data durability in the presence of a region wide outage. It is important to note that in a distributed system, even with strong consistency, it is impossible to have a distributed database with an RPO and RTO of zero due to CAP Theorem.

REGION(S)	REPLICATION MODE	CONSISTENCY LEVEL	RPO	RTO
1	Single or Multiple write regions	Any Consistency Level	< 240 Minutes	<1 Week
>1	Single write region	Session, Consistent Prefix, Eventual	< 15 minutes	< 15 minutes
>1	Single write region	Bounded Staleness	*K* & *T*	< 15 minutes
>1	Single write region	Strong	0	< 15 minutes
>1	Multiple write regions	Session, Consistent Prefix, Eventual	< 15 minutes	0
>1	Multiple write regions	Bounded Staleness	*K* & *T*	0

K = The number of *"K"* versions (i.e., updates) of an item.

T = The time interval *"T"* since the last update.

For a single region account, the minimum value of *K* and *T* is 10 write operations or 5 seconds. For multi-region accounts the minimum value of *K* and *T* is 100,000 write operations or 300 seconds. This defines the minimum RPO for data when using Bounded Staleness.

Strong consistency and multiple write regions

Cosmos accounts configured with multiple write regions cannot be configured for strong consistency as it is not possible for a distributed system to provide an RPO of zero and an RTO of zero. Additionally, there are no write latency benefits on using strong consistency with multiple write regions because a write to any region must be replicated and committed to all configured regions within the account. This results in the same write latency as a single write region account.

Additional reading

To learn more about consistency concepts, read the following articles:

- High-level TLA+ specifications for the five consistency levels offered by Azure Cosmos DB
- Replicated Data Consistency Explained Through Baseball (video) by Doug Terry
- Replicated Data Consistency Explained Through Baseball (whitepaper) by Doug Terry
- Session guarantees for weakly consistent replicated data
- Consistency Tradeoffs in Modern Distributed Database Systems Design: CAP is Only Part of the Story
- Probabilistic Bounded Staleness (PBS) for Practical Partial Quorums
- Eventually Consistent - Revisited

Next steps

To learn more about consistency levels in Azure Cosmos DB, read the following articles:

- Configure the default consistency level
- Override the default consistency level
- Azure Cosmos DB SLA

How does Azure Cosmos DB provide high availability

12/4/2020 • 10 minutes to read • Edit Online

APPLIES TO: SQL API Cassandra API Gremlin API Table API Azure Cosmos DB API for MongoDB

Azure Cosmos DB provides high availability in two primary ways. First, Azure Cosmos DB replicates data across regions configured within a Cosmos account. Second, Azure Cosmos DB maintains 4 replicas of data within a region.

Azure Cosmos DB is a globally distributed database service and is a foundational service in Azure. By default, is available in all regions where Azure is available. You can associate any number of Azure regions with your Azure Cosmos account and your data is automatically and transparently replicated. You can add or remove a region to your Azure Cosmos account at any time. Cosmos DB is available in all five distinct Azure cloud environments available to customers:

- **Azure public** cloud, which is available globally.
- **Azure China 21Vianet** is available through a unique partnership between Microsoft and 21Vianet, one of the country's largest internet providers in China.
- **Azure Germany** provides services under a data trustee model, which ensures that customer data remains in Germany under the control of T-Systems International GmbH, a subsidiary of Deutsche Telekom, acting as the German data trustee.
- **Azure Government** is available in four regions in the United States to US government agencies and their partners.
- **Azure Government for Department of Defense (DoD)** is available in two regions in the United States to the US Department of Defense.

Within a region, Azure Cosmos DB maintains four copies of your data as replicas within physical partitions as shown in the following image:

- The data within Azure Cosmos containers is horizontally partitioned.
- A partition-set is a collection of multiple replica-sets. Within each region, every partition is protected by a replica-set with all writes replicated and durably committed by a majority of replicas. Replicas are distributed across as many as 10-20 fault domains.
- Each partition across all the regions is replicated. Each region contains all the data partitions of an Azure Cosmos container and can accept writes and serve reads.

If your Azure Cosmos account is distributed across *N* Azure regions, there will be at least *N* x 4 copies of all your data. Having an Azure Cosmos account in more than 2 regions improves the availability of your application and provides low latency across the associated regions.

SLAs for availability

As a globally distributed database, Azure Cosmos DB provides comprehensive SLAs that encompass throughput, latency at the 99th percentile, consistency, and high availability. The table below shows the guarantees for high availability provided by Azure Cosmos DB for single and multi-region accounts. For high availability, always configure your Azure Cosmos accounts to have multiple write regions.

OPERATION TYPE	SINGLE REGION	MULTI-REGION (SINGLE REGION WRITES)	MULTI-REGION (MULTI-REGION WRITES)
Writes	99.99	99.99	99.999
Reads	99.99	99.999	99.999

> **NOTE**
>
> In practice, the actual write availability for bounded staleness, session, consistent prefix and eventual consistency models is significantly higher than the published SLAs. The actual read availability for all consistency levels is significantly higher than the published SLAs.

High availability with Azure Cosmos DB in the event of regional outages

For the rare cases of regional outage, Azure Cosmos DB makes sure your database is always highly available. The following details capture Azure Cosmos DB behavior during an outage, depending on your Azure Cosmos account configuration:

- With Azure Cosmos DB, before a write operation is acknowledged to the client, the data is durably committed by a quorum of replicas within the region that accepts the write operations. For more details, see Consistency levels and throughput
- Multi-region accounts configured with multiple-write regions will be highly available for both writes and reads. Regional failovers are detected and handled in the Azure Cosmos DB client. They are also instantaneous and don't require any changes from the application.
- Single-region accounts may lose availability following a regional outage. It's always recommended to set up **at least two regions** (preferably, at least two write regions) with your Azure Cosmos account to ensure high availability at all times.

Multi-region accounts with a single-write region (write region outage)

- During a write region outage, the Azure Cosmos account will automatically promote a secondary region to be the new primary write region when **enable automatic failover** is configured on the Azure Cosmos account. When enabled, the failover will occur to another region in the order of region priority you've specified.
- When the previously impacted region is back online, any write data that was not replicated when the region failed, is made available through the conflicts feed. Applications can read the conflicts feed, resolve the conflicts based on the application-specific logic, and write the updated data back to the Azure Cosmos container as appropriate.
- Once the previously impacted write region recovers, it becomes automatically available as a read region. You can switch back to the recovered region as the write region. You can switch the regions by using PowerShell, Azure CLI or Azure portal. There is **no data or availability loss** before, during or after you switch the write region and your application continues to be highly available.

> **IMPORTANT**
>
> It is strongly recommended that you configure the Azure Cosmos accounts used for production workloads to **enable automatic failover**. Manual failover requires connectivity between secondary and primary write region to complete a consistency check to ensure there is no data loss during the failover. If the primary region is unavailable, this consistency check cannot complete and the manual failover will not succeed, resulting in loss of write availability for the duration of the regional outage.

Multi-region accounts with a single-write region (read region outage)

- During a read region outage, Azure Cosmos accounts using any consistency level or strong consistency with three or more read regions will remain highly available for reads and writes.
- Azure Cosmos accounts using strong consistency with two or fewer read regions (which includes the read & write region) will lose read write availability during a read region outage.
- The impacted region is automatically disconnected and will be marked offline. The Azure Cosmos DB SDKs will redirect read calls to the next available region in the preferred region list.
- If none of the regions in the preferred region list is available, calls automatically fall back to the current write region.

- No changes are required in your application code to handle read region outage. When the impacted read region is back online it will automatically sync with the current write region and will be available again to serve read requests.
- Subsequent reads are redirected to the recovered region without requiring any changes to your application code. During both failover and rejoining of a previously failed region, read consistency guarantees continue to be honored by Azure Cosmos DB.
- Even in a rare and unfortunate event when the Azure region is permanently irrecoverable, there is no data loss if your multi-region Azure Cosmos account is configured with *Strong* consistency. In the event of a permanently irrecoverable write region, a multi-region Azure Cosmos account configured with bounded-staleness consistency, the potential data loss window is restricted to the staleness window (*K* or *T*) where K=100,000 updates and T=5 minutes. For session, consistent-prefix and eventual consistency levels, the potential data loss window is restricted to a maximum of 15 minutes. For more information on RTO and RPO targets for Azure Cosmos DB, see Consistency levels and data durability

Availability Zone support

In addition to cross region resiliency, you can now enable **zone redundancy** when selecting a region to associate with your Azure Cosmos database.

With Availability Zone support, Azure Cosmos DB will ensure replicas are placed across multiple zones within a given region to provide high availability and resiliency during zonal failures. There are no changes to latency and other SLAs in this configuration. In the event of a single zone failure, zone redundancy provides full data durability with RPO=0 and availability with RTO=0.

Zone redundancy is a *supplemental capability* to the replication in multi-region writes feature. Zone redundancy alone cannot be relied upon to achieve regional resiliency. For example, in the event of regional outages or low latency access across the regions, it's advised to have multiple write regions in addition to zone redundancy.

When configuring multi-region writes for your Azure Cosmos account, you can opt into zone redundancy at no extra cost. Otherwise, please see the note below regarding the pricing for zone redundancy support. You can enable zone redundancy on an existing region of your Azure Cosmos account by removing the region and adding it back with the zone redundancy enabled. For a list of regions where availability zones are supported, see the Availability zones documentation.

The following table summarizes the high availability capability of various account configurations:

KPI	SINGLE REGION WITHOUT AVAILABILITY ZONES (NON-AZ)	SINGLE REGION WITH AVAILABILITY ZONES (AZ)	MULTI-REGION WRITES WITH AVAILABILITY ZONES (AZ, 2 REGIONS) – MOST RECOMMENDED SETTING
Write availability SLA	99.99%	99.99%	99.999%
Read availability SLA	99.99%	99.99%	99.999%
Price	Single region billing rate	Single region Availability Zone billing rate	Multi-region billing rate
Zone failures – data loss	Data loss	No data loss	No data loss
Zone failures – availability	Availability loss	No availability loss	No availability loss
Read latency	Cross region	Cross region	Low

KPI	SINGLE REGION WITHOUT AVAILABILITY ZONES (NON-AZ)	SINGLE REGION WITH AVAILABILITY ZONES (AZ)	MULTI-REGION WRITES WITH AVAILABILITY ZONES (AZ, 2 REGIONS) – MOST RECOMMENDED SETTING
Write latency	Cross region	Cross region	Low
Regional outage – data loss	Data loss	Data loss	Data loss When using bounded staleness consistency with multiple write regions and more than one region, data loss is limited to the bounded staleness configured on your account You can avoid data loss during a regional outage by configuring strong consistency with multiple regions. This option comes with trade-offs that affect availability and performance. It can be configured only on accounts that are configured for single-region writes.
Regional outage – availability	Availability loss	Availability loss	No availability loss
Throughput	X RU/s provisioned throughput	X RU/s provisioned throughput * 1.25	2X RU/s provisioned throughput This configuration mode requires twice the amount of throughput when compared to a single region with Availability Zones because there are two regions.

NOTE

To enable Availability Zone support for a multi region Azure Cosmos account, the account must have multi-region writes enabled.

You can enable zone redundancy when adding a region to new or existing Azure Cosmos accounts. To enable zone redundancy on your Azure Cosmos account, you should set the `isZoneRedundant` flag to `true` for a specific location. You can set this flag within the locations property. For example, the following PowerShell snippet enables zone redundancy for the "Southeast Asia" region:

Availability Zones can be enabled via:

- Azure portal
- Azure PowerShell
- Azure CLI

- Azure Resource Manager templates

Building highly available applications

- Review the expected behavior of the Azure Cosmos SDKs during these events and which are the configurations that affect it.
- To ensure high write and read availability, configure your Azure Cosmos account to span at least two regions with multiple-write regions. This configuration will provide the highest availability, lowest latency, and best scalability for both reads and writes backed by SLAs. To learn more, see how to configure your Azure Cosmos account with multiple write-regions.
- For multi-region Azure Cosmos accounts that are configured with a single-write region, enable automatic-failover by using Azure CLI or Azure portal. After you enable automatic failover, whenever there is a regional disaster, Cosmos DB will automatically failover your account.
- Even if your Azure Cosmos account is highly available, your application may not be correctly designed to remain highly available. To test the end-to-end high availability of your application, as a part of your application testing or disaster recovery (DR) drills, temporarily disable automatic-failover for the account, invoke the manual failover by using PowerShell, Azure CLI or Azure portal, then monitor your application's failover. Once complete, you can fail back over to the primary region and restore automatic-failover for the account.
- Within a globally distributed database environment, there is a direct relationship between the consistency level and data durability in the presence of a region-wide outage. As you develop your business continuity plan, you need to understand the maximum acceptable time before the application fully recovers after a disruptive event. The time required for an application to fully recover is known as recovery time objective (RTO). You also need to understand the maximum period of recent data updates the application can tolerate losing when recovering after a disruptive event. The time period of updates that you might afford to lose is known as recovery point objective (RPO). To see the RPO and RTO for Azure Cosmos DB, see Consistency levels and data durability

Next steps

Next you can read the following articles:

- Availability and performance tradeoffs for various consistency levels
- Globally scaling provisioned throughput
- Global distribution - under the hood
- Consistency levels in Azure Cosmos DB
- How to configure your Cosmos account with multiple write regions
- SDK behavior on multi-regional environments

Conflict types and resolution policies when using multiple write regions

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: SQL API Cassandra API Gremlin API Table API Azure Cosmos DB API for MongoDB

Conflicts and conflict resolution policies are applicable if your Azure Cosmos DB account is configured with multiple write regions.

For Azure Cosmos accounts configured with multiple write regions, update conflicts can occur when writers concurrently update the same item in multiple regions. Update conflicts can be of the following three types:

- **Insert conflicts**: These conflicts can occur when an application simultaneously inserts two or more items with the same unique index in two or more regions. For example, this conflict might occur with an ID property.
- **Replace conflicts**: These conflicts can occur when an application updates the same item simultaneously in two or more regions.
- **Delete conflicts**: These conflicts can occur when an application simultaneously deletes an item in one region and updates it in another region.

Conflict resolution policies

Azure Cosmos DB offers a flexible policy-driven mechanism to resolve write conflicts. You can select from two conflict resolution policies on an Azure Cosmos container:

- **Last Write Wins (LWW)**: This resolution policy, by default, uses a system-defined timestamp property. It's based on the time-synchronization clock protocol. If you use the SQL API, you can specify any other custom numerical property (e.g., your own notion of a timestamp) to be used for conflict resolution. A custom numerical property is also referred to as the *conflict resolution path*.

  If two or more items conflict on insert or replace operations, the item with the highest value for the conflict resolution path becomes the winner. The system determines the winner if multiple items have the same numeric value for the conflict resolution path. All regions will converge to a single winner and end up with the same version of the committed item. When delete conflicts are involved, the deleted version always wins over either insert or replace conflicts. This outcome occurs no matter what the value of the conflict resolution path is.

  > **NOTE**
  > Last Write Wins is the default conflict resolution policy and uses timestamp `_ts` for the following APIs: SQL, MongoDB, Cassandra, Gremlin and Table. Custom numerical property is available only for SQL API.

  To learn more, see examples that use LWW conflict resolution policies.

- **Custom**: This resolution policy is designed for application-defined semantics for reconciliation of conflicts. When you set this policy on your Azure Cosmos container, you also need to register a *merge stored procedure*. This procedure is automatically invoked when conflicts are detected under a database transaction on the server. The system provides exactly once guarantee for the execution of a merge procedure as part of

the commitment protocol.

If you configure your container with the custom resolution option, and you fail to register a merge procedure on the container or the merge procedure throws an exception at runtime, the conflicts are written to the *conflicts feed*. Your application then needs to manually resolve the conflicts in the conflicts feed. To learn more, see examples of how to use the custom resolution policy and how to use the conflicts feed.

> **NOTE**
>
> Custom conflict resolution policy is available only for SQL API accounts.

Next steps

Learn how to configure conflict resolution policies:

- How to configure multiple write regions for your applications
- How to manage conflict resolution policies
- How to read from the conflicts feed

Global data distribution with Azure Cosmos DB - under the hood

11/2/2020 • 9 minutes to read • Edit Online

APPLIES TO: SQL API Cassandra API Gremlin API Table API Azure Cosmos DB API for MongoDB

Azure Cosmos DB is a foundational service in Azure, so it's deployed across all Azure regions worldwide including the public, sovereign, Department of Defense (DoD) and government clouds. Within a data center, we deploy and manage the Azure Cosmos DB on massive stamps of machines, each with dedicated local storage. Within a data center, Azure Cosmos DB is deployed across many clusters, each potentially running multiple generations of hardware. Machines within a cluster are typically spread across 10-20 fault domains for high availability within a region. The following image shows the Cosmos DB global distribution system topology:

Global distribution in Azure Cosmos DB is turnkey: At any time, with a few clicks or programmatically with a single API call, you can add or remove the geographical regions associated with your Cosmos database. A Cosmos database, in turn, consists of a set of Cosmos containers. In Cosmos DB, containers serve as the logical units of distribution and scalability. The collections, tables, and graphs you create are (internally) just Cosmos containers. Containers are completely schema-agnostic and provide a scope for a query. Data in a Cosmos container is automatically indexed upon ingestion. Automatic indexing enables users to query the data without the hassles of schema or index management, especially in a globally distributed setup.

- In a given region, data within a container is distributed by using a partition-key, which you provide and is transparently managed by the underlying physical partitions (*local distribution*).
- Each physical partition is also replicated across geographical regions (*global distribution*).

When an app using Cosmos DB elastically scales throughput on a Cosmos container or consumes more storage, Cosmos DB transparently handles partition management operations (split, clone, delete) across all the regions. Independent of the scale, distribution, or failures, Cosmos DB continues to provide a single system image of the

data within the containers, which are globally distributed across any number of regions.

As shown in the following image, the data within a container is distributed along two dimensions - within a region and across regions, worldwide:

A physical partition is implemented by a group of replicas, called a *replica-set*. Each machine hosts hundreds of replicas that correspond to various physical partitions within a fixed set of processes as shown in the image above. Replicas corresponding to the physical partitions are dynamically placed and load balanced across the machines within a cluster and data centers within a region.

A replica uniquely belongs to an Azure Cosmos DB tenant. Each replica hosts an instance of Cosmos DB's database engine, which manages the resources as well as the associated indexes. The Cosmos database engine operates on an atom-record-sequence (ARS) based type system. The engine is agnostic to the concept of a schema, blurring the boundary between the structure and instance values of records. Cosmos DB achieves full schema agnosticism by automatically indexing everything upon ingestion in an efficient manner, which allows users to query their globally distributed data without having to deal with schema or index management.

The Cosmos database engine consists of components including implementation of several coordination primitives, language runtimes, the query processor, and the storage and indexing subsystems responsible for transactional storage and indexing of data, respectively. To provide durability and high availability, the database engine persists its data and index on SSDs and replicates it among the database engine instances within the replica-set(s) respectively. Larger tenants correspond to higher scale of throughput and storage and have either bigger or more replicas or both. Every component of the system is fully asynchronous – no thread ever blocks, and each thread does short-lived work without incurring any unnecessary thread switches. Rate-limiting and back-pressure are plumbed across the entire stack from the admission control to all I/O paths. Cosmos database engine is designed to exploit fine-grained concurrency and to deliver high throughput while operating within frugal amounts of system resources.

Cosmos DB's global distribution relies on two key abstractions – *replica-sets* and *partition-sets*. A replica-set is a modular Lego block for coordination, and a partition-set is a dynamic overlay of one or more geographically distributed physical partitions. To understand how global distribution works, we need to understand these two key abstractions.

Replica-sets

A physical partition is materialized as a self-managed and dynamically load-balanced group of replicas spread across multiple fault domains, called a replica-set. This set collectively implements the replicated state machine protocol to make the data within the physical partition highly available, durable, and consistent. The replica-set membership *N* is dynamic – it keeps fluctuating between *NMin* and *NMax* based on the failures, administrative operations, and the time for failed replicas to regenerate/recover. Based on the membership changes, the replication protocol also reconfigures the size of read and write quorums. To uniformly distribute the throughput that is assigned to a given physical partition, we employ two ideas:

- First, the cost of processing the write requests on the leader is higher than the cost of applying the updates on the follower. Correspondingly, the leader is budgeted more system resources than the followers.
- Secondly, as far as possible, the read quorum for a given consistency level is composed exclusively of the follower replicas. We avoid contacting the leader for serving reads unless required. We employ a number of ideas from the research done on the relationship of load and capacity in the quorum-based systems for the five consistency models that Cosmos DB supports.

Partition-sets

A group of physical partitions, one from each of the configured with the Cosmos database regions, is composed to manage the same set of keys replicated across all the configured regions. This higher coordination primitive is called a *partition-set* - a geographically distributed dynamic overlay of physical partitions managing a given set of keys. While a given physical partition (a replica-set) is scoped within a cluster, a partition-set can span clusters, data centers, and geographical regions as shown in the image below:

You can think of a partition-set as a geographically dispersed "super replica-set", which is composed of multiple replica-sets owning the same set of keys. Similar to a replica-set, a partition-set's membership is also dynamic – it fluctuates based on implicit physical partition management operations to add/remove new partitions to/from a

given partition-set (for instance, when you scale out throughput on a container, add/remove a region to your Cosmos database, or when failures occur). By virtue of having each of the partitions (of a partition-set) manage the partition-set membership within its own replica-set, the membership is fully decentralized and highly available. During the reconfiguration of a partition-set, the topology of the overlay between physical partitions is also established. The topology is dynamically selected based on the consistency level, geographical distance, and available network bandwidth between the source and the target physical partitions.

The service allows you to configure your Cosmos databases with either a single write region or multiple write regions, and depending on the choice, partition-sets are configured to accept writes in exactly one or all regions. The system employs a two-level, nested consensus protocol – one level operates within the replicas of a replica-set of a physical partition accepting the writes, and the other operates at the level of a partition-set to provide complete ordering guarantees for all the committed writes within the partition-set. This multi-layered, nested consensus is critical for the implementation of our stringent SLAs for high availability, as well as the implementation of the consistency models, which Cosmos DB offers to its customers.

Conflict resolution

Our design for the update propagation, conflict resolution, and causality tracking is inspired from the prior work on epidemic algorithms and the Bayou system. While the kernels of the ideas have survived and provide a convenient frame of reference for communicating the Cosmos DB's system design, they have also undergone significant transformation as we applied them to the Cosmos DB system. This was needed, because the previous systems were designed neither with the resource governance nor with the scale at which Cosmos DB needs to operate, nor to provide the capabilities (for example, bounded staleness consistency) and the stringent and comprehensive SLAs that Cosmos DB delivers to its customers.

Recall that a partition-set is distributed across multiple regions and follows Cosmos DBs (multi-region writes) replication protocol to replicate the data among the physical partitions comprising a given partition-set. Each physical partition (of a partition-set) accepts writes and serves reads typically to the clients that are local to that region. Writes accepted by a physical partition within a region are durably committed and made highly available within the physical partition before they are acknowledged to the client. These are tentative writes and are propagated to other physical partitions within the partition-set using an anti-entropy channel. Clients can request either tentative or committed writes by passing a request header. The anti-entropy propagation (including the frequency of propagation) is dynamic, based on the topology of the partition-set, regional proximity of the physical partitions, and the consistency level configured. Within a partition-set, Cosmos DB follows a primary commit scheme with a dynamically selected arbiter partition. The arbiter selection is dynamic and is an integral part of the reconfiguration of the partition-set based on the topology of the overlay. The committed writes (including multi-row/batched updates) are guaranteed to be ordered.

We employ encoded vector clocks (containing region ID and logical clocks corresponding to each level of consensus at the replica-set and partition-set, respectively) for causality tracking and version vectors to detect and resolve update conflicts. The topology and the peer selection algorithm are designed to ensure fixed and minimal storage and minimal network overhead of version vectors. The algorithm guarantees the strict convergence property.

For the Cosmos databases configured with multiple write regions, the system offers a number of flexible automatic conflict resolution policies for the developers to choose from, including:

- **Last-Write-Wins (LWW)**, which, by default, uses a system-defined timestamp property (which is based on the time-synchronization clock protocol). Cosmos DB also allows you to specify any other custom numerical property to be used for conflict resolution.
- **Application-defined (Custom) conflict resolution policy** (expressed via merge procedures), which is designed for application-defined semantics reconciliation of conflicts. These procedures get invoked upon detection of the write-write conflicts under the auspices of a database transaction on the server side. The system provides exactly once guarantee for the execution of a merge procedure as a part of the commitment

protocol. There are several conflict resolution samples available for you to play with.

Consistency Models

Whether you configure your Cosmos database with a single or multiple write regions, you can choose from the five well-defined consistency models. With multiple write regions, the following are a few notable aspects of the consistency levels:

The bounded staleness consistency guarantees that all reads will be within *K* prefixes or *T* seconds from the latest write in any of the regions. Furthermore, reads with bounded staleness consistency are guaranteed to be monotonic and with consistent prefix guarantees. The anti-entropy protocol operates in a rate-limited manner and ensures that the prefixes do not accumulate and the backpressure on the writes does not have to be applied. Session consistency guarantees monotonic read, monotonic write, read your own writes, write follows read, and consistent prefix guarantees, worldwide. For the databases configured with strong consistency, the benefits (low write latency, high write availability) of multiple write regions does not apply, because of synchronous replication across regions.

The semantics of the five consistency models in Cosmos DB are described here, and mathematically described using a high-level TLA+ specifications here.

Next steps

Next learn how to configure global distribution by using the following articles:

- Add/remove regions from your database account
- How to configure clients for multi-homing
- How to create a custom conflict resolution policy

Partitioning and horizontal scaling in Azure Cosmos DB

11/2/2020 • 9 minutes to read • Edit Online

APPLIES TO: SQL API Cassandra API Gremlin API Table API Azure Cosmos DB API for MongoDB

Azure Cosmos DB uses partitioning to scale individual containers in a database to meet the performance needs of your application. In partitioning, the items in a container are divided into distinct subsets called *logical partitions*. Logical partitions are formed based on the value of a *partition key* that is associated with each item in a container. All the items in a logical partition have the same partition key value.

For example, a container holds items. Each item has a unique value for the `UserID` property. If `UserID` serves as the partition key for the items in the container and there are 1,000 unique `UserID` values, 1,000 logical partitions are created for the container.

In addition to a partition key that determines the item's logical partition, each item in a container has an *item ID* (unique within a logical partition). Combining the partition key and the *item ID* creates the item's *index*, which uniquely identifies the item. Choosing a partition key is an important decision that will affect your application's performance.

This article explains the relationship between logical and physical partitions. It also discusses best practices for partitioning and gives an in-depth view at how horizontal scaling works in Azure Cosmos DB. It's not necessary to understand these internal details to select your partition key but we have covered them so you have clarity on how Azure Cosmos DB works.

Logical partitions

A logical partition consists of a set of items that have the same partition key. For example, in a container that contains data about food nutrition, all items contain a `foodGroup` property. You can use `foodGroup` as the partition key for the container. Groups of items that have specific values for `foodGroup`, such as `Beef Products`, `Baked Products`, and `Sausages and Luncheon Meats`, form distinct logical partitions. You don't have to worry about deleting a logical partition when the underlying data is deleted.

A logical partition also defines the scope of database transactions. You can update items within a logical partition by using a transaction with snapshot isolation. When new items are added to a container, new logical partitions are transparently created by the system.

There is no limit to the number of logical partitions in your container. Each logical partition can store up to 20GB of data. Good partition key choices have a wide range of possible values. For example, in a container where all items contain a `foodGroup` property, the data within the `Beef Products` logical partition can grow up to 20 GB. Selecting a partition key with a wide range of possible values ensures that the container is able to scale.

Physical partitions

A container is scaled by distributing data and throughput across physical partitions. Internally, one or more logical partitions are mapped to a single physical partition. Typically smaller containers have many logical partitions but they only require a single physical partition. Unlike logical partitions,

physical partitions are an internal implementation of the system and they are entirely managed by Azure Cosmos DB.

The number of physical partitions in your container depends on the following configuration:

- The number of throughput provisioned (each individual physical partition can provide a throughput of up to 10,000 request units per second).
- The total data storage (each individual physical partition can store up to 50GB data).

There is no limit to the total number of physical partitions in your container. As your provisioned throughput or data size grows, Azure Cosmos DB will automatically create new physical partitions by splitting existing ones. Physical partition splits do not impact your application's availability. After the physical partition split, all data within a single logical partition will still be stored on the same physical partition. A physical partition split simply creates a new mapping of logical partitions to physical partitions.

Throughput provisioned for a container is divided evenly among physical partitions. A partition key design that doesn't distribute requests evenly might result in too many requests directed to a small subset of partitions that become "hot." Hot partitions lead to inefficient use of provisioned throughput, which might result in rate-limiting and higher costs.

You can see your container's physical partitions in the **Storage** section of the **Metrics blade** of the Azure portal:

In the above screenshot, a container has `/foodGroup` as the partition key. Each of the three bars in the graph represents a physical partition. In the image, **partition key range** is the same as a physical partition. The selected physical partition contains three logical partitions: `Beef Products`, `Vegetable and Vegetable Products`, and `Soups, Sauces, and Gravies`.

If you provision a throughput of 18,000 request units per second (RU/s), then each of the three physical

partition can utilize 1/3 of the total provisioned throughput. Within the selected physical partition, the logical partition keys `Beef Products`, `Vegetable and Vegetable Products`, and `Soups, Sauces, and Gravies` can, collectively, utilize the physical partition's 6,000 provisioned RU/s. Because provisioned throughput is evenly divided across your container's physical partitions, it's important to choose a partition key that evenly distributes throughput consumption by choosing the right logical partition key.

> **NOTE**
>
> If you choose a partition key that evenly distributes throughput consumption across logical partitions, you will ensure that throughput consumption across physical partitions is balanced.

Managing logical partitions

Azure Cosmos DB transparently and automatically manages the placement of logical partitions on physical partitions to efficiently satisfy the scalability and performance needs of the container. As the throughput and storage requirements of an application increase, Azure Cosmos DB moves logical partitions to automatically spread the load across a greater number of physical partitions. You can learn more about physical partitions.

Azure Cosmos DB uses hash-based partitioning to spread logical partitions across physical partitions. Azure Cosmos DB hashes the partition key value of an item. The hashed result determines the physical partition. Then, Azure Cosmos DB allocates the key space of partition key hashes evenly across the physical partitions.

Transactions (in stored procedures or triggers) are allowed only against items in a single logical partition.

You can learn more about how Azure Cosmos DB manages partitions. (It's not necessary to understand the internal details to build or run your applications, but added here for a curious reader.)

Replica sets

Each physical partition consists of a set of replicas, also referred to as a *replica set*. Each replica set hosts an instance of the database engine. A replica set makes the data stored within the physical partition durable, highly available, and consistent. Each replica that makes up the physical partition inherits the partition's storage quota. All replicas of a physical partition collectively support the throughput that's allocated to the physical partition. Azure Cosmos DB automatically manages replica sets.

Typically smaller containers only require a single physical partition but they will still have at least 4 replicas.

The following image shows how logical partitions are mapped to physical partitions that are distributed globally:

Choosing a partition key

A partition key has two components: **partition key path** and the **partition key value**. For example, consider an item { "userId" : "Andrew", "worksFor": "Microsoft" } if you choose "userId" as the partition key, the following are the two partition key components:

- The partition key path (For example: "/userId"). The partition key path accepts alphanumeric and underscore(_) characters. You can also use nested objects by using the standard path notation(/).
- The partition key value (For example: "Andrew"). The partition key value can be of string or numeric types.

To learn about the limits on throughput, storage, and length of the partition key, see the Azure Cosmos DB service quotas article.

Selecting your partition key is a simple but important design choice in Azure Cosmos DB. Once you select your partition key, it is not possible to change it in-place. If you need to change your partition key, you should move your data to a new container with your new desired partition key.

For **all** containers, your partition key should:

- Be a property that has a value which does not change. If a property is your partition key, you can't update that property's value.
- Have a high cardinality. In other words, the property should have a wide range of possible values.
- Spread request unit (RU) consumption and data storage evenly across all logical partitions. This ensures even RU consumption and storage distribution across your physical partitions.

If you need multi-item ACID transactions in Azure Cosmos DB, you will need to use stored procedures or triggers. All JavaScript-based stored procedures and triggers are scoped to a single logical partition.

Partition keys for read-heavy containers

For most containers, the above criteria is all you need to consider when picking a partition key. For large read-heavy containers, however, you might want to choose a partition key that appears frequently as a filter in your queries. Queries can be efficiently routed to only the relevant physical partitions by including the partition key in the filter predicate.

If most of your workload's requests are queries and most of your queries have an equality filter on the same property, this property can be a good partition key choice. For example, if you frequently run a query that filters on `UserID`, then selecting `UserID` as the partition key would reduce the number of cross-partition queries.

However, if your container is small, you probably don't have enough physical partitions to need to worry about the performance impact of cross-partition queries. Most small containers in Azure Cosmos DB only require one or two physical partitions.

If your container could grow to more than a few physical partitions, then you should make sure you pick a partition key that minimizes cross-partition queries. Your container will require more than a few physical partitions when either of the following are true:

- Your container will have over 30,000 RU's provisioned
- Your container will store over 100 GB of data

Using item ID as the partition key

If your container has a property that has a wide range of possible values, it is likely a great partition key choice. One possible example of such a property is the *item ID*. For small read-heavy containers or write-heavy containers of any size, the *item ID* is naturally a great choice for the partition key.

The system property *item ID* exists in every item in your container. You may have other properties that represent a logical ID of your item. In many cases, these are also great partition key choices for the same reasons as the *item ID*.

The *item ID* is a great partition key choice for the following reasons:

- There are a wide range of possible values (one unique *item ID* per item).
- Because there is a unique *item ID* per item, the *item ID* does a great job at evenly balancing RU consumption and data storage.
- You can easily do efficient point reads since you'll always know an item's partition key if you know its *item ID*.

Some things to consider when selecting the *item ID* as the partition key include:

- If the *item ID* is the partition key, it will become a unique identifier throughout your entire container. You won't be able to have items that have a duplicate *item ID*.
- If you have a read-heavy container that has a lot of physical partitions, queries will be more efficient if they have an equality filter with the *item ID*.
- You can't run stored procedures or triggers across multiple logical partitions.

Next steps

- Learn about provisioned throughput in Azure Cosmos DB.
- Learn about global distribution in Azure Cosmos DB.
- Learn how to provision throughput on an Azure Cosmos container.
- Learn how to provision throughput on an Azure Cosmos database.

Request Units in Azure Cosmos DB

12/4/2020 • 5 minutes to read • Edit Online

APPLIES TO: SQL API Cassandra API Gremlin API Table API Azure Cosmos DB API for MongoDB

Azure Cosmos DB supports many APIs, such as SQL, MongoDB, Cassandra, Gremlin, and Table. Each API has its own set of database operations. These operations range from simple point reads and writes to complex queries. Each database operation consumes system resources based on the complexity of the operation.

The cost of all database operations is normalized by Azure Cosmos DB and is expressed by Request Units (or RUs, for short). Request unit is a performance currency abstracting the system resources such as CPU, IOPS, and memory that are required to perform the database operations supported by Azure Cosmos DB.

The cost to do a point read (i.e. fetching a single item by its ID and partition key value) for a 1 KB item is 1 Request Unit (or 1 RU). All other database operations are similarly assigned a cost using RUs. No matter which API you use to interact with your Azure Cosmos container, costs are always measured by RUs. Whether the database operation is a write, point read, or query, costs are always measured in RUs.

The following image shows the high-level idea of RUs:

To manage and plan capacity, Azure Cosmos DB ensures that the number of RUs for a given database operation over a given dataset is deterministic. You can examine the response header to track the number of RUs that are consumed by any database operation. When you understand the factors that affect RU charges and your application's throughput requirements, you can run your application cost effectively.

The type of Azure Cosmos account you're using determines the way consumed RUs get charged. There are 3 modes in which you can create an account:

1. **Provisioned throughput mode**: In this mode, you provision the number of RUs for your application on a per-second basis in increments of 100 RUs per second. To scale the

provisioned throughput for your application, you can increase or decrease the number of RUs at any time in increments or decrements of 100 RUs. You can make your changes either programmatically or by using the Azure portal. You are billed on an hourly basis for the amount of RUs per second you have provisioned. To learn more, see the Provisioned throughput article.

You can provision throughput at two distinct granularities:

- **Containers**: For more information, see Provision throughput on an Azure Cosmos container.
- **Databases**: For more information, see Provision throughput on an Azure Cosmos database.

2. **Serverless mode**: In this mode, you don't have to provision any throughput when creating resources in your Azure Cosmos account. At the end of your billing period, you get billed for the amount of Request Units that has been consumed by your database operations. To learn more, see the Serverless throughput article.

3. **Autoscale mode**: In this mode, you can automatically and instantly scale the throughput (RU/s) of your database or container based on it's usage, without impacting the availability, latency, throughput, or performance of the workload. This mode is well suited for mission-critical workloads that have variable or unpredictable traffic patterns, and require SLAs on high performance and scale. To learn more, see the autoscale throughput article.

Request Unit considerations

While you estimate the number of RUs consumed by your workload, consider the following factors:

- **Item size**: As the size of an item increases, the number of RUs consumed to read or write the item also increases.

- **Item indexing**: By default, each item is automatically indexed. Fewer RUs are consumed if you choose not to index some of your items in a container.

- **Item property count**: Assuming the default indexing is on all properties, the number of RUs consumed to write an item increases as the item property count increases.

- **Indexed properties**: An index policy on each container determines which properties are indexed by default. To reduce the RU consumption for write operations, limit the number of indexed properties.

- **Data consistency**: The strong and bounded staleness consistency levels consume approximately two times more RUs while performing read operations when compared to that of other relaxed consistency levels.

- **Type of reads**: Point reads cost significantly fewer RUs than queries.

- **Query patterns**: The complexity of a query affects how many RUs are consumed for an operation. Factors that affect the cost of query operations include:

  - The number of query results
  - The number of predicates
  - The nature of the predicates
  - The number of user-defined functions
  - The size of the source data
  - The size of the result set
  - Projections

The same query on the same data will always costs the same number of RUs on repeated executions.

- **Script usage**: As with queries, stored procedures and triggers consume RUs based on the complexity of the operations that are performed. As you develop your application, inspect the request charge header to better understand how much RU capacity each operation consumes.

Request units and multiple regions

If you provision *'R'* RUs on a Cosmos container (or database), Cosmos DB ensures that *'R'* RUs are available in *each* region associated with your Cosmos account. You can't selectively assign RUs to a specific region. The RUs provisioned on a Cosmos container (or database) are provisioned in all the regions associated with your Cosmos account.

Assuming that a Cosmos container is configured with *'R'* RUs and there are *'N'* regions associated with the Cosmos account, the total RUs available globally on the container = *R* x *N*.

Your choice of consistency model also affects the throughput. You can get approximately 2x read throughput for the more relaxed consistency levels (e.g., *session*, *consistent prefix* and *eventual* consistency) compared to stronger consistency levels (e.g., *bounded staleness* or *strong* consistency).

Next steps

- Learn more about how to provision throughput on Azure Cosmos containers and databases.
- Learn more about serverless on Azure Cosmos DB.
- Learn more about logical partitions.
- Learn how to provision throughput on an Azure Cosmos container.
- Learn how to provision throughput on an Azure Cosmos database.
- Learn how to find the request unit charge for an operation.
- Learn how to optimize provisioned throughput cost in Azure Cosmos DB.
- Learn how to optimize reads and writes cost in Azure Cosmos DB.
- Learn how to optimize query cost in Azure Cosmos DB.
- Learn how to use metrics to monitor throughput.

Introduction to provisioned throughput in Azure Cosmos DB

12/4/2020 • 10 minutes to read • Edit Online

APPLIES TO: SQL API Cassandra API Gremlin API Table API Azure Cosmos DB API for MongoDB

Azure Cosmos DB allows you to set provisioned throughput on your databases and containers. There are two types of provisioned throughput, standard (manual) or autoscale. This article gives an overview of how provisioned throughput works.

An Azure Cosmos database is a unit of management for a set of containers. A database consists of a set of schema-agnostic containers. An Azure Cosmos container is the unit of scalability for both throughput and storage. A container is horizontally partitioned across a set of machines within an Azure region and is distributed across all Azure regions associated with your Azure Cosmos account.

With Azure Cosmos DB, you can provision throughput at two granularities:

- Azure Cosmos containers
- Azure Cosmos databases

Set throughput on a container

The throughput provisioned on an Azure Cosmos container is exclusively reserved for that container. The container receives the provisioned throughput all the time. The provisioned throughput on a container is financially backed by SLAs. To learn how to configure standard (manual) throughput on a container, see Provision throughput on an Azure Cosmos container. To learn how to configure autoscale throughput on a container, see Provision autoscale throughput.

Setting provisioned throughput on a container is the most frequently used option. You can elastically scale throughput for a container by provisioning any amount of throughput by using Request Units (RUs).

The throughput provisioned for a container is evenly distributed among its physical partitions, and assuming a good partition key that distributes the logical partitions evenly among the physical partitions, the throughput is also distributed evenly across all the logical partitions of the container. You cannot selectively specify the throughput for logical partitions. Because one or more logical partitions of a container are hosted by a physical partition, the physical partitions belong exclusively to the container and support the throughput provisioned on the container.

If the workload running on a logical partition consumes more than the throughput that was allocated to the underlying physical partition, it's possible that your operations will be rate-limited. What is known as a *hot partition* occurs when one logical partition has disproportionately more requests than other partition key values.

When rate-limiting occurs, you can either increase the provisioned throughput for the entire container or retry the operations. You also should ensure that you choose a partition key that evenly distributes storage and request volume. For more information about partitioning, see Partitioning and horizontal scaling in Azure Cosmos DB.

We recommend that you configure throughput at the container granularity when you want predictable performance for the container.

The following image shows how a physical partition hosts one or more logical partitions of a container:

Set throughput on a database

When you provision throughput on an Azure Cosmos database, the throughput is shared across all the containers (called shared database containers) in the database. An exception is if you specified a provisioned throughput on specific containers in the database. Sharing the database-level provisioned throughput among its containers is analogous to hosting a database on a cluster of machines. Because all containers within a database share the resources available on a machine, you naturally do not get predictable performance on any specific container. To learn how to configure provisioned throughput on a database, see Configure provisioned throughput on an Azure Cosmos database. To learn how to configure autoscale throughput on a database, see Provision autoscale throughput.

Because all containers within the database share the provisioned throughput, Azure Cosmos DB doesn't provide any predictable throughput guarantees for a particular container in that database. The portion of the throughput that a specific container can receive is dependent on:

- The number of containers.
- The choice of partition keys for various containers.
- The distribution of the workload across various logical partitions of the containers.

We recommend that you configure throughput on a database when you want to share the throughput across multiple containers, but don't want to dedicate the throughput to any particular container.

The following examples demonstrate where it's preferred to provision throughput at the database level:

- Sharing a database's provisioned throughput across a set of containers is useful for a multitenant application. Each user can be represented by a distinct Azure Cosmos container.

- Sharing a database's provisioned throughput across a set of containers is useful when you migrate a NoSQL database, such as MongoDB or Cassandra, hosted on a cluster of VMs or from on-premises physical servers to Azure Cosmos DB. Think of the provisioned throughput configured on your Azure Cosmos database as a logical equivalent, but more cost-effective and elastic, to that of the compute capacity of your MongoDB or Cassandra cluster.

All containers created inside a database with provisioned throughput must be created with a partition key. At any given point in time, the throughput allocated to a container within a database is distributed across all the logical partitions of that container. When you have containers that share provisioned throughput configured on a database, you can't selectively apply the throughput to a specific container or a logical partition.

If the workload on a logical partition consumes more than the throughput that's allocated to a specific logical partition, your operations are rate-limited. When rate-limiting occurs, you can either increase the throughput for the entire database or retry the operations. For more information on partitioning, see Logical partitions.

Containers in a shared throughput database share the throughput (RU/s) allocated to that database. With standard (manual) provisioned throughput, you can have up to 25 containers with a minimum of 400 RU/s on the database. With autoscale provisioned throughput, you can have up to 25 containers in a database with autoscale max 4000 RU/s (scales between 400 - 4000 RU/s).

> **NOTE**
>
> In February 2020, we introduced a change that allows you to have a maximum of 25 containers in a shared throughput database, which better enables throughput sharing across the containers. After the first 25 containers, you can add more containers to the database only if they are provisioned with dedicated throughput, which is separate from the shared throughput of the database.
> If your Azure Cosmos DB account already contains a shared throughput database with >=25 containers, the account and all other accounts in the same Azure subscription are exempt from this change. Please contact product support if you have feedback or questions.

If your workloads involve deleting and recreating all the collections in a database, it is recommended that you drop the empty database and recreate a new database prior to collection creation. The following image shows how a physical partition can host one or more logical partitions that belong to different containers within a database:

Set throughput on a database and a container

You can combine the two models. Provisioning throughput on both the database and the container is allowed. The following example shows how to provision standard (manual) provisioned throughput on an Azure Cosmos database and a container:

- You can create an Azure Cosmos database named *Z* with standard (manual) provisioned throughput of *"K"* RUs.
- Next, create five containers named *A*, *B*, *C*, *D*, and *E* within the database. When creating container B, make sure to enable **Provision dedicated throughput for this container** option and explicitly configure *"P"* RUs of provisioned throughput on this container. You can configure shared and dedicated throughput only when creating the database and container.

- The *"K"* RUs throughput is shared across the four containers *A*, *C*, *D*, and *E*. The exact amount of throughput available to *A*, *C*, *D*, or *E* varies. There are no SLAs for each individual container's throughput.
- The container named *B* is guaranteed to get the *"P"* RUs throughput all the time. It's backed by SLAs.

> **NOTE**
>
> A container with provisioned throughput cannot be converted to shared database container. Conversely a shared database container cannot be converted to have a dedicated throughput.

Update throughput on a database or a container

After you create an Azure Cosmos container or a database, you can update the provisioned throughput. There is no limit on the maximum provisioned throughput that you can configure on the database or the container.

Current provisioned throughput

You can retrieve the provisioned throughput of a container or a database in the Azure portal or by using the SDKs:

- Container.ReadThroughputAsync on the .NET SDK.
- CosmosContainer.readThroughput on the Java SDK.

The response of those methods also contains the minimum provisioned throughput for the container or database:

- ThroughputResponse.MinThroughput on the .NET SDK.
- ThroughputResponse.getMinThroughput() on the Java SDK.

The actual minimum RU/s may vary depending on your account configuration. But generally it's the maximum of:

- 400 RU/s
- Current storage in GB * 10 RU/s (unless your container or database contains more than 1 TB of data, see our high storage / low throughput program)
- Highest RU/s provisioned on the database or container / 100

Changing the provisioned throughput

You can scale the provisioned throughput of a container or a database through the Azure portal or by using the SDKs:

- Container.ReplaceThroughputAsync on the .NET SDK.
- CosmosContainer.replaceThroughput on the Java SDK.

If you are **reducing the provisioned throughput**, you will be able to do it up to the minimum.

If you are **increasing the provisioned throughput**, most of the time, the operation is instantaneous. There are however, cases where the operation can take longer time due to the system tasks to provision the required resources. In this case, an attempt to modify the provisioned throughput while this operation is in progress will yield an HTTP 423 response with an error message explaining that another scaling operation is in progress.

> **NOTE**
>
> If you are planning for a very large ingestion workload that will require a big increase in provisioned throughput, keep in mind that the scaling operation has no SLA and, as mentioned in the previous paragraph, it can take a long time when the increase is large. You might want to plan ahead and start the scaling before the workload starts and use the below methods to check progress.

You can programmatically check the scaling progress by reading the current provisioned throughput and using:

- ThroughputResponse.IsReplacePending on the .NET SDK.
- ThroughputResponse.isReplacePending() on the Java SDK.

You can use Azure Monitor metrics to view the history of provisioned throughput (RU/s) and storage on a resource.

High storage / low throughput program

As described in the Current provisioned throughput section above, the minimum throughput you can provision on a container or database depends on a number of factors. One of them is the amount of data currently stored, as Azure Cosmos DB enforces a minimum throughput of 10 RU/s per GB of storage.

This can be a concern in situations where you need to store large amounts of data, but have low throughput requirements in comparison. To better accommodate these scenarios, Azure Cosmos DB has introduced a **"high storage / low throughput" program** that decreases the RU/s per GB constraint on eligible accounts.

You currently need to have at least 1 container or shared-throughput database containing more than 1 TB of data in your account to be eligible. To join this program and assess your full eligibility, all you have to do is to fill this survey. The Azure Cosmos DB team will then follow up and proceed with your onboarding.

Comparison of models

This table shows a comparison between provisioning standard (manual) throughput on a database vs. on a container.

PARAMETER	STANDARD (MANUAL) THROUGHPUT ON A DATABASE	STANDARD (MANUAL) THROUGHPUT ON A CONTAINER	AUTOSCALE THROUGHPUT ON A DATABASE	AUTOSCALE THROUGHPUT ON A CONTAINER

PARAMETER	STANDARD (MANUAL) THROUGHPUT ON A DATABASE	STANDARD (MANUAL) THROUGHPUT ON A CONTAINER	AUTOSCALE THROUGHPUT ON A DATABASE	AUTOSCALE THROUGHPUT ON A CONTAINER
Entry point (minimum RU/s)	400 RU/s. Can have up to 25 containers with no RU/s minimum per container.	400	Autoscale between 400 - 4000 RU/s. Can have up to 25 containers with no RU/s minimum per container.	Autoscale between 400 - 4000 RU/s.
Minimum RU/s per container	--	400	--	Autoscale between 400 - 4000 RU/s
Maximum RUs	Unlimited, on the database.	Unlimited, on the container.	Unlimited, on the database.	Unlimited, on the container.
RUs assigned or available to a specific container	No guarantees. RUs assigned to a given container depend on the properties. Properties can be the choice of partition keys of containers that share the throughput, the distribution of the workload, and the number of containers.	All the RUs configured on the container are exclusively reserved for the container.	No guarantees. RUs assigned to a given container depend on the properties. Properties can be the choice of partition keys of containers that share the throughput, the distribution of the workload, and the number of containers.	All the RUs configured on the container are exclusively reserved for the container.
Maximum storage for a container	Unlimited.	Unlimited	Unlimited	Unlimited
Maximum throughput per logical partition of a container	10K RU/s	10K RU/s	10K RU/s	10K RU/s
Maximum storage (data + index) per logical partition of a container	20 GB	20 GB	20 GB	20 GB

Next steps

- Learn more about logical partitions.
- Learn how to provision standard (manual) on an Azure Cosmos container.
- Learn how to provision standard (manual) throughput on an Azure Cosmos database.
- Learn how to provision autoscale throughput on an Azure Cosmos database or container.

Create Azure Cosmos containers and databases with autoscale throughput

12/4/2020 • 5 minutes to read • Edit Online

APPLIES TO: SQL API Cassandra API Gremlin API Table API Azure Cosmos DB API for MongoDB

In Azure Cosmos DB, you can configure either standard (manual) or autoscale provisioned throughput on your databases and containers. Autoscale provisioned throughput in Azure Cosmos DB allows you to **scale the throughput (RU/s) of your database or container automatically and instantly**. The throughput is scaled based on the usage, without impacting the availability, latency, throughput, or performance of the workload.

Autoscale provisioned throughput is well suited for mission-critical workloads that have variable or unpredictable traffic patterns, and require SLAs on high performance and scale. This article describes the benefits and use cases of autoscale provisioned throughput.

Benefits of autoscale

Azure Cosmos databases and containers that are configured with autoscale provisioned throughput have the following benefits:

- **Simple:** Autoscale removes the complexity of managing RU/s with custom scripting or manually scaling capacity.
- **Scalable:** Databases and containers automatically scale the provisioned throughput as needed. There is no disruption to client connections, applications, or impact to Azure Cosmos DB SLAs.
- **Cost-effective:** Autoscale helps optimize your RU/s usage and cost usage by scaling down when not in use. You only pay for the resources that your workloads need on a per-hour basis. Of all hours in a month, if you set autoscale max RU/s(Tmax) and use the full amount Tmax for 66% of the hours or less, you'll save with autoscale. To learn more, see the how to choose between standard (manual) and autoscale provisioned throughput article.
- **Highly available:** Databases and containers using autoscale use the same globally distributed, fault-tolerant, highly available Azure Cosmos DB backend to ensure data durability and high availability.

Use cases of autoscale

The use cases of autoscale include:

- **Variable or unpredictable workloads:** When your workloads have variable or unpredictable spikes in usage, autoscale helps by automatically scaling up and down based on usage. Examples include retail websites that have different traffic patterns depending on seasonality; IOT workloads that have spikes at various times during the day; line of business applications that see peak usage a few times a month or year, and more. With autoscale, you no longer need to manually provision for peak or average capacity.
- **New applications:** If you're developing a new application and not sure about the throughput (RU/s) you need, autoscale makes it easy to get started. You can start with the autoscale entry point of 400 - 4000 RU/s, monitor your usage, and determine the right RU/s over time.
- **Infrequently used applications:** If you have an application that's only used for a few hours several times a day, week, or month — such as a low-volume application/web/blog site — autoscale adjusts the

capacity to handle peak usage and scales down when it's over.

- **Development and test workloads:** If you or your team use Azure Cosmos databases and containers during work hours, but don't need them on nights or weekends, autoscale helps save cost by scaling down to a minimum when not in use.
- **Scheduled production workloads/queries:** If you have a series of scheduled requests, operations, or queries that you want to run during idle periods, you can do that easily with autoscale. When you need to run the workload, the throughput will automatically scale to what's needed and scale down afterward.

Building a custom solution to these problems not only requires an enormous amount of time, but also introduces complexity in your application's configuration or code. Autoscale enables the above scenarios out of the box and removes the need for custom or manual scaling of capacity.

How autoscale provisioned throughput works

When configuring containers and databases with autoscale, you specify the maximum throughput `Tmax` required. Azure Cosmos DB scales the throughput `T` such `0.1*Tmax <= T <= Tmax`. For example, if you set the maximum throughput to 20,000 RU/s, the throughput will scale between 2000 to 20,000 RU/s. Because scaling is automatic and instantaneous, at any point in time, you can consume up to the provisioned `Tmax` with no delay.

Each hour, you will be billed for the highest throughput `T` the system scaled to within the hour.

The entry point for autoscale maximum throughput `Tmax` starts at 4000 RU/s, which scales between 400 - 4000 RU/s. You can set `Tmax` in increments of 1000 RU/s and change the value at any time.

Enable autoscale on existing resources

Use the Azure portal to enable autoscale on an existing database or container. You can switch between autoscale and standard (manual) provisioned throughput at any time. See this documentation for more information. Currently, for all APIs, you can only use the Azure portal to enable autoscale on existing resources.

Throughput and storage limits for autoscale

For any value of `Tmax`, the database or container can store a total of `0.01 * Tmax GB`. After this amount of storage is reached, the maximum RU/s will be automatically increased based on the new storage value, with no impact to your application.

For example, if you start with a maximum RU/s of 50,000 RU/s (scales between 5000 - 50,000 RU/s), you can store up to 500 GB of data. If you exceed 500 GB - e.g. storage is now 600 GB, the new maximum RU/s will be 60,000 RU/s (scales between 6000 - 60,000 RU/s).

When you use database level throughput with autoscale, you can have the first 25 containers share an autoscale maximum RU/s of 4000 (scales between 400 - 4000 RU/s), as long as you don't exceed 40 GB of storage. See this documentation for more information.

Comparison – containers configured with manual vs autoscale throughput

For more detail, see this documentation on how to choose between standard (manual) and autoscale throughput.

	CONTAINERS WITH STANDARD (MANUAL) THROUGHPUT	CONTAINERS WITH AUTOSCALE THROUGHPUT
Provisioned throughput (RU/s)	Manually provisioned.	Automatically and instantaneously scaled based on the workload usage patterns.
Rate-limiting of requests/operations (429)	May happen, if consumption exceeds provisioned capacity.	Will not happen if you consume RU/s within the autoscale throughput range that you've set.
Capacity planning	You have to do capacity planning and provision the exact throughput you need.	The system automatically takes care of capacity planning and capacity management.
Pricing	You pay for the manually provisioned RU/s per hour, using the standard (manual) RU/s per hour rate.	You pay per hour for the highest RU/s the system scaled up to within the hour. For single write region accounts, you pay for the RU/s used on an hourly basis, using the autoscale RU/s per hour rate. For accounts with multiple write regions, there is no extra charge for autoscale. You pay for the throughput used on hourly basis using the same multi-region write RU/s per hour rate.
Best suited for workload types	Predictable and stable workloads	Unpredictable and variable workloads

Next steps

- Review the autoscale FAQ.
- Learn how to choose between manual and autoscale throughput.
- Learn how to provision autoscale throughput on an Azure Cosmos database or container.
- Learn more about partitioning in Azure Cosmos DB.

Frequently asked questions about autoscale provisioned throughput in Azure Cosmos DB

11/2/2020 • 11 minutes to read • Edit Online

APPLIES TO: SQL API Cassandra API Gremlin API Table API Azure Cosmos DB API for MongoDB

With autoscale provisioned throughput, Azure Cosmos DB will automatically manage and scale the RU/s of your database or container based on usage. This article answers commonly asked questions about autoscale.

Frequently asked questions

What is the difference between "autopilot" and "autoscale" in Azure Cosmos DB?

"Autoscale" or "autoscale provisioned throughput" is the updated name for the feature, previously known as "autopilot." With the current release of autoscale, we've added new features, including the ability to set custom max RU/s and programmatic support.

What happens to databases or containers created in the previous autopilot tier model?

Resources that were created with the previous tier model are automatically supported with the new autoscale custom maximum RU/s model. The upper bound of the tier becomes the new maximum RU/s, which results in the same scale range.

For example, if you previously selected the tier that scaled between 400 to 4000 RU/s, the database or container will now show as having a maximum RU/s of 4000 RU/s, which scales between 400 to 4000 RU/s. From here, you can change the maximum RU/s to a custom value to suit your workload.

How quickly will autoscale scale up based on spikes in traffic?

With autoscale, the system scales the throughput (RU/s) `T` up or down within the `0.1 * Tmax` and `Tmax` range, based on incoming traffic. Because the scaling is automatic and instantaneous, at any point in time, you can consume up to the provisioned `Tmax` with no delay.

How do I determine what RU/s the system is currently scaled to?

Use Azure Monitor metrics to monitor both the provisioned autoscale max RU/s and the current throughput (RU/s) the system is scaled to.

What is the pricing for autoscale?

Each hour, you will be billed for the highest throughput `T` the system scaled to within the hour. If your resource had no requests during the hour or did not scale beyond `0.1 * Tmax`, you will be billed for the minimum of `0.1 * Tmax`. Refer to the Azure Cosmos DB pricing page for details.

How does autoscale show up on my bill?

In single write region accounts, the autoscale rate per 100 RU/s is 1.5x the rate of standard (manual) provisioned throughput. On your bill, you will see the existing standard provisioned throughput meter. The quantity of this meter will be multiplied by 1.5. For example, if the highest RU/s the system scaled to within an hour was 6000 RU/s, you'd be billed 60 * 1.5 = 90 units of the meter for that hour.

In accounts with multiple write regions, the autoscale rate per 100 RU/s is the same as the rate for standard (manual) provisioned multiple write region throughput. On your bill, you will see the existing multiple write regions meter. Since the rates are the same, if you use autoscale, you'll see the same quantity as with standard throughput.

Does autoscale work with reserved capacity?

Yes. When you purchase reserved capacity for accounts with multiple write regions, the reservation discount for autoscale resources is applied to your meter usage at a ratio of 1.5 * the ratio of the specific region.

Multi-write region reserved capacity works the same for autoscale and standard (manual) provisioned throughput. See Azure Cosmos DB reserved capacity

Does autoscale work with free tier?

Yes. In free tier, you can use autoscale throughput on a container. Support for autoscale shared throughput databases with custom max RU/s is not yet available. See how free tier billing works with autoscale.

Is autoscale supported for all APIs?

Yes, autoscale is supported for all APIs: Core (SQL), Gremlin, Table, Cassandra, and API for MongoDB.

Is autoscale supported for multi-region write accounts?

Yes. The max RU/s are available in each region that is added to the Azure Cosmos DB account.

How do I enable autoscale on new databases or containers?

See this article on how to enable autoscale.

Can I enable autoscale on an existing database or a container?

Yes. You can also switch between autoscale and standard (manual) provisioned throughput as needed. Currently, for all APIs, you can only use the Azure portal to do these operations.

How does the migration between autoscale and standard (manual) provisioned throughput work?

Conceptually, changing the throughput type is a two-stage process. First, you send a request to change the throughput settings to use either autoscale or manual provisioned throughput. In both cases, the system will automatically determine and set an initial RU/s value, based on the current throughput settings and storage. During this step, no user-provided RU/s value will be accepted. Then, after the update is complete, you can change the RU/s to suit your workload.

Migration from standard (manual) provisioned throughput to autoscale

For a container, use the following formula to estimate the initial autoscale max RU/s: `MAX(4000, current manual provisioned RU/s, maximum RU/s ever provisioned / 10, storage in GB * 100)`, rounded to the nearest 1000 RU/s. The actual initial autoscale max RU/s may vary depending on your account configuration.

Example #1: Suppose you have a container with 10,000 RU/s manual provisioned throughput, and 25 GB of storage. When you enable autoscale, the initial autoscale max RU/s will be: 10,000 RU/s, which will scale between 1000 - 10,000 RU/s.

Example #2: Suppose you have a container with 50,000 RU/s manual provisioned throughput, and 2500 GB of storage. When you enable autoscale, the initial autoscale max RU/s will be: 250,000 RU/s, which will scale between 25,000 - 250,000 RU/s.

Migration from autoscale to standard (manual) provisioned throughput

The initial manual provisioned throughput will be equal to the current autoscale max RU/s.

Example: Suppose you have an autoscale database or container with max RU/s of 20,000 RU/s (scales between 2000 - 20,000 RU/s). When you update to use manual provisioned throughput, the initial throughput will be 20,000 RU/s.

Is there Azure CLI or PowerShell support to manage databases or containers with autoscale?

Currently, you can only create and manage resources with autoscale from the Azure portal, .NET V3 SDK, Java V4 SDK, and Azure Resource Manager. Support in Azure CLI, PowerShell, and other SDKs is not yet available.

Is autoscale supported for shared throughput databases?

Yes, autoscale is supported for shared throughput databases. To enable this feature, select autoscale and the **Provision throughput** option when creating the database.

What is the number of allowed containers per shared throughput database when autoscale is enabled?

Azure Cosmos DB enforces a maximum of 25 containers in a shared throughput database, which applies to databases with autoscale or standard (manual) throughput.

What is the impact of autoscale on database consistency level?

There is no impact of the autoscale on consistency level of the database. See the consistency levels article for more information regarding available consistency levels.

What is the storage limit associated with each max RU/s option?

The storage limit in GB for each max RU/s is: Max RU/s of database or container / 100. For example, if the max RU/s is 20,000 RU/s, the resource can support 200 GB of storage. See the autoscale limits article for the available max RU/s and storage options.

What happens if I exceed the storage limit associated with my max throughput?

If the storage limit associated with the max throughput of the database or container is exceeded, Azure Cosmos DB will automatically increase the max throughput to the next highest RU/s that can support that level of storage.

For example, if you start with a max RU/s of 50,000 RU/s (scales between 5000 - 50,000 RU/s), you can store up to 500 GB of data. If you exceed 500 GB - e.g. storage is now 600 GB, the new maximum RU/s will be 60,000 RU/s (scales between 6000 - 60,000 RU/s).

Can I change the max RU/s on the database or container?

Yes. See this article on how to change the max RU/s. When you change the max RU/s, depending on the requested value, this can be an asynchronous operation that may take some time to complete (may be up to 4-6 hours, depending on the RU/s selected)

Increasing the max RU/s

When you send a request to increase the max RU/s `Tmax`, depending on the max RU/s selected, the service provisions more resources to support the higher max RU/s. While this is happening, your existing workload and operations will not be affected. The system will continue to scale your database or container between the previous `0.1*Tmax` to `Tmax` until the new scale range of `0.1*Tmax_new` to `Tmax_new` is ready.

Lowering the max RU/s

When you lower the max RU/s, the minimum value you can set it to is: `MAX(4000, highest max RU/s ever provisioned / 10, current storage in GB * 100)`, rounded to the nearest 1000 RU/s.

Example #1: Suppose you have an autoscale container with max RU/s of 20,000 RU/s (scales between 2000 - 20,000 RU/s) and 50 GB of storage. The lowest, minimum value you can set max RU/s to is: MAX(4000, 20,000 / 10, **50 * 100**) = 5000 RU/s (scales between 500 - 5000 RU/s).

Example #2: Suppose you have an autoscale container with max RU/s of 100,000 RU/s and 100 GB of storage. Now, you scale max RU/s up to 150,000 RU/s (scales between 15,000 - 150,000 RU/s). The lowest, minimum value you can now set max RU/s to is: MAX(4000, **150,000 / 10**, 100 * 100) = 15,000 RU/s (scales between 1500 - 15,000 RU/s).

For a shared throughput database, when you lower the max RU/s, the minimum value you can set it to is:

```
MAX(4000, highest max RU/s ever provisioned / 10, current storage in GB * 100, 4000 + (MAX(Container count - 25, 0) * 1000))
```

, rounded to the nearest 1000 RU/s.

The above formulas and examples relate to the minimum autoscale max RU/s you can set, and is distinct from the `0.1 * Tmax` to `Tmax` range the system automatically scales between. No matter what the max RU/s is, the system

will always scale between `0.1 * Tmax` to `Tmax`.

How does TTL work with autoscale?

With autoscale, TTL operations do not affect the scaling of RU/s. Any RUs consumed due to TTL are not part of the billed RU/s of the autoscale container.

For example, suppose you have an autoscale container with 400 – 4000 RU/s:

- Hour 1: T=0: The container has no usage (no TTL or workload requests). The billable RU/s is 400 RU/s.
- Hour 1: T=1: TTL is enabled.
- Hour 1: T=2: The container starts getting requests, which consume 1000 RU in 1 second. There are also 200 RUs worths of TTL that need to happen. The billable RU/s is still 1000 RU/s. Regardless of when the TTLs occur, they will not affect the autoscale scaling logic.

What is the mapping between the max RU/s and physical partitions?

When you first select the max RU/s, Azure Cosmos DB will provision: Max RU/s / 10,000 RU/s = # of physical partitions. Each physical partition can support up to 10,000 RU/s and 50 GB of storage. As storage size grows, Azure Cosmos DB will automatically split the partitions to add more physical partitions to handle the storage increase, or increase the max RU/s if storage exceeds the associated limit.

The max RU/s of the database or container is divided evenly across all physical partitions. So, the total throughput any single physical partition can scale to is: Max RU/s of database or container / # physical partitions.

What happens if incoming requests exceed the max RU/s of the database or container?

If the overall consumed RU/s exceeds the max RU/s of the database or container, requests that exceed the max RU/s will be throttled and return a 429 status code. Requests that result in over 100% normalized utilization will also be throttled. Normalized utilization is defined as the max of the RU/s utilization across all physical partitions. For example, suppose your max throughput is 20,000 RU/s and you have two physical partitions, P_1 and P_2, each capable of scaling to 10,000 RU/s. In a given second, if P_1 has used 6000 RUs, and P_2 8000 RUs, the normalized utilization is MAX(6000 RU / 10,000 RU, 8000 RU / 10,000 RU) = 0.8.

> **NOTE**
>
> The Azure Cosmos DB client SDKs and data import tools (Azure Data Factory, bulk executor library) automatically retry on 429s, so occasional 429s are fine. A sustained high number of 429s may indicate you need to increase the max RU/s or review your partitioning strategy for a hot partition.

Is it still possible to see 429s (throttling/rate limiting) when autoscale is enabled?

Yes. It is possible to see 429s in two scenarios. First, when the overall consumed RU/s exceeds the max RU/s of the database or container, the service will throttle requests accordingly.

Second, if there is a hot partition, i.e. a logical partition key value that has a disproportionately higher amount of requests compared to other partition key values, it is possible for the underlying physical partition to exceed its RU/s budget. As a best practice, to avoid hot partitions, choose a good partition key that results in an even distribution of both storage and throughput.

For example, if you select the 20,000 RU/s max throughput option and have 200 GB of storage, with four physical partitions, each physical partition can be autoscaled up to 5000 RU/s. If there was a hot partition on a particular logical partition key, you will see 429s when the underlying physical partition it resides in exceeds 5000 RU/s, i.e. exceeds 100% normalized utilization.

Next steps

- Learn how to enable autoscale on an Azure Cosmos DB database or container.

- Learn about the benefits of provisioned throughput with autoscale.
- Learn more about logical and physical partitions.

Azure Cosmos DB serverless (Preview)

12/4/2020 • 4 minutes to read • Edit Online

APPLIES TO: SQL API Cassandra API Gremlin API Table API Azure Cosmos DB API for MongoDB

> **IMPORTANT**
> Azure Cosmos DB serverless is currently in preview. This preview version is provided without a Service Level Agreement and is not recommended for production workloads. For more information, see Supplemental terms of use for Microsoft Azure previews.

Azure Cosmos DB serverless lets you use your Azure Cosmos account in a consumption-based fashion where you are only charged for the Request Units consumed by your database operations and the storage consumed by your data. There is no minimum charge involved when using Azure Cosmos DB in serverless mode.

> **IMPORTANT**
> Do you have any feedback about serverless? We want to hear it! Feel free to drop a message to the Azure Cosmos DB serverless team: azurecosmosdbserverless@service.microsoft.com.

When using Azure Cosmos DB, every database operation has a cost expressed in Request Units. How you are charged for this cost depends on the type of Azure Cosmos account you are using:

- In provisioned throughput mode, you have to commit to a certain amount of throughput (expressed in Request Units per second) that is provisioned on your databases and containers. The cost of your database operations is then deducted from the number of Request Units available every second. At the end of your billing period, you get billed for the amount of throughput you have provisioned.
- In serverless mode, you don't have to provision any throughput when creating containers in your Azure Cosmos account. At the end of your billing period, you get billed for the number of Request Units that were consumed by your database operations.

Use-cases

Azure Cosmos DB serverless best fits scenarios where you expect:

- **Low, intermittent and unpredictable traffic**: Because provisioning capacity in such situations isn't required and may be cost-prohibitive
- **Moderate performance**: Because serverless containers have specific performance characteristics

For these reasons, Azure Cosmos DB serverless should be considered for the following types of workload:

- Development
- Testing
- Prototyping
- Proof of concept
- Non-critical application with light traffic

See the how to choose between provisioned throughput and serverless article for more guidance on how to

choose the offer that best fits your use-case.

Using serverless resources

Serverless is a new Azure Cosmos account type, which means that you have to choose between **provisioned throughput** and **serverless** when creating a new account. You must create a new serverless account to get started with serverless. During the preview release, the only supported way to create a new serverless account is by using the Azure portal. Migrating existing accounts to/from serverless mode is not currently supported.

Any container that is created in a serverless account is a serverless container. Serverless containers expose the same capabilities as containers created in provisioned throughput mode, so you read, write and query your data the exact same way. However serverless accounts and containers also have specific characteristics:

> **IMPORTANT**
>
> Some of these limitations may be eased or removed when serverless becomes generally available and **your feedback** will help us decide! Reach out and tell us more about your serverless experience: azurecosmosdbserverless@service.microsoft.com.

- A serverless account can only run in a single Azure region. It is not possible to add additional Azure regions to a serverless account after you create it.
- It is not possible to enable the Synapse Link preview feature on a serverless account.
- Provisioning throughput is not required on serverless containers, so the following statements are applicable:
  - You can't pass any throughput when creating a serverless container and doing so returns an error.
  - You can't read or update the throughput on a serverless container and doing so returns an error.
  - You can't create a shared throughput database in a serverless account and doing so returns an error.
- Serverless containers can store a maximum of 50 GB of data and indexes.

Monitoring your consumption

If you have used Azure Cosmos DB in provisioned throughput mode before, you will find that serverless is more cost-effective when your traffic doesn't justify provisioned capacity. The trade-off is that your costs will become less predictable because you are billed based on the number of requests your database has processed. Because of that, it's important to keep an eye on your current consumption.

When browsing the **Metrics** pane of your account, you will find a chart named **Request Units consumed** under the **Overview** tab. This chart shows how many Request Units your account has consumed:

You can find the same chart when using Azure Monitor, as described here. Note that Azure Monitor lets you

setup alerts, which can be used to notify you when your Request Unit consumption has passed a certain threshold.

Performance

Serverless resources yield specific performance characteristics that are different from what provisioned throughput resources deliver:

- **Availability**: After the serverless offer becomes generally available, the availability of serverless containers will be covered by a Service Level Agreement (SLA) of 99.9% when Availability Zones (zone redundancy) aren't used. The SLA is 99.99% when Availability Zones are used.
- **Latency**: After the serverless offer becomes generally available, the latency of serverless containers will be covered by a Service Level Objective (SLO) of 10 milliseconds or less for point-reads and 30 milliseconds or less for writes. A point-read operation consists in fetching a single item by its ID and partition key value.
- **Burstability**: After the serverless offer becomes generally available, the burstability of serverless containers will be covered by a Service Level Objective (SLO) of 95%. This means the maximum burstability can be attained at least 95% of the time.

> **NOTE**
>
> As any Azure preview, Azure Cosmos DB serverless is excluded from Service Level Agreements (SLA). The performance characteristics mentioned above are provided as a preview of what this offer will deliver when generally available.

Next steps

Get started with serverless with the following articles:

- Request Units in Azure Cosmos DB
- Choose between provisioned throughput and serverless
- Pricing model in Azure Cosmos DB

How to choose between standard (manual) and autoscale provisioned throughput

11/2/2020 • 10 minutes to read • Edit Online

APPLIES TO: SQL API Cassandra API Gremlin API Table API Azure Cosmos DB API for MongoDB

Azure Cosmos DB supports two types or offers of provisioned throughput: standard (manual) and autoscale. Both throughput types are suitable for mission-critical workloads that require high performance and scale, and are backed by the same Azure Cosmos DB SLAs on throughput, availability, latency, and consistency.

This article describes how to choose between standard (manual) and autoscale provisioned throughput for your workload.

Overview of provisioned throughput types

Before diving into the difference between standard (manual) and autoscale, it's important to first understand how provisioned throughput works in Azure Cosmos DB.

When you use provisioned throughput, you set the throughput, measured in request units per second (RU/s) required for your workload. The service provisions the capacity needed to support the throughput requirements. Database operations against the service, such as reads, writes, and queries consume some amount of request units (RUs). Learn more about request units.

The following table shows a high-level comparison between standard (manual) and autoscale.

DESCRIPTION	STANDARD (MANUAL)	AUTOSCALE
Best suited for	Workloads with steady or predictable traffic	Workloads with variable or unpredictable traffic. See use cases of autoscale.
How it works	You provision a set amount of RU/s `T` that is static over time, unless you manually change them. Each second, you can use up to `T` RU/s throughput. For example, if you set standard (manual) 400 RU/s, the throughput will stay at 400 RU/s.	You set the highest, or maximum RU/s `Tmax` you don't want the system to exceed. The system automatically scales the throughput `T` such that `0.1* Tmax <= T <= Tmax`. For example, if you set autoscale maximum RU/s of 4000 RU/s, the system will scale between 400 - 4000 RU/s.

DESCRIPTION	STANDARD (MANUAL)	AUTOSCALE
When to use it	You want to manually manage your throughput capacity (RU/s) and scale yourself. You have high, consistent utilization of provisioned RU/s. Of all hours in a month, if you set provisioned RU/s `T` and use the full amount for 66% of the hours or more, it's estimated you'll save with standard (manual) provisioned RU/s. This is based on a comparison between setting `T` in standard (manual) and the same amount `Tmax` in autoscale.	You want Azure Cosmos DB to manage your throughput capacity (RU/s) and scale, based on usage. You have RU/s usage that is variable or hard to predict. Of all hours in a month, if you set autoscale max RU/s `Tmax` and use the full amount `Tmax` for 66% of the hours or less, it's estimated you'll save with autoscale. This is based on a comparison between setting autoscale `Tmax` and the same amount `T` in standard (manual) throughput.
Billing model	Billing is done on a per-hour basis for the RU/s provisioned, regardless of how many RUs were consumed. Example: • Provision 400 RU/s • Hour 1: no requests • Hour 2: 400 RU/s worth of requests For both hours 1 and 2, you'll be billed 400 RU/s for both hours at the standard (manual) rates.	Billing is done on a per-hour basis, for the highest RU/s the system scaled to in the hour. Example: • Provision autoscale max RU/s of 4000 RU/s (scales between 400 - 4000 RU/s) • Hour 1: system scaled up to highest value of 3500 RU/s • Hour 2: system scaled down to minimum of 400 RU/s (always 10% of `Tmax`), due to no usage You will be billed for 3500 RU/s in hour 1, and 400 RU/s in hour 2 at the autoscale provisioned throughput rates. The autoscale rate per RU/s is 1.5 * the standard (manual) rate.
What happens if you exceed provisioned RU/s	The RU/s remain static at what is provisioned. Any requests that consume beyond the provisioned RUs in a second will be rate-limited, with a response that recommends a time to wait before retrying. You can manually increase or decrease the RU/s if needed.	The system will scale the RU/s up to the autoscale max RU/s. Any requests that consume beyond the autoscale max RU/s in a second will be rate-limited, with a response that recommends a time to wait before retrying.

Understand your traffic patterns

New applications

If you are building a new application and do not know your traffic pattern yet, you may want to start at the entry point RU/s (or minimum RU/s) to avoid over-provisioning in the beginning. Or, if you have a small application that doesn't need high scale, you may want to provision just the minimum entry point RU/s to optimize cost. For small applications with a low expected traffic, you can also consider the serverless capacity mode.

Whether you plan to use standard (manual) or autoscale, here's what you should consider:

If you provision standard (manual) RU/s at the entry point of 400 RU/s, you won't be able to consume above 400 RU/s, unless you manually change the throughput. You'll be billed for 400 RU/s at the standard (manual)

provisioned throughput rate, per hour.

If you provision autoscale throughput at the entry point of max RU/s of 4000 RU/s, the resource will scale between 400 to 4000 RU/s. Since the autoscale throughput billing rate per RU/s is 1.5x of the standard (manual) rate, for hours where the system has scaled down to the minimum of 400 RU/s, your bill will be higher than if you provisioned 400 RU/s manually. However, with autoscale, at any time, if your application traffic spikes, you can consume up to 4000 RU/s with no user action required. In general, you should weigh the benefit of being able to consume up to the max RU/s at any time with the 1.5x rate of autoscale.

Use the Azure Cosmos DB capacity calculator to estimate your throughput requirements.

Existing applications

If you have an existing application using standard (manual) provisioned throughput, you can use Azure Monitor metrics to determine if your traffic pattern is suitable for autoscale.

First, find the normalized request unit consumption metric of your database or container. Normalized utilization is a measure of how much you are currently using your standard (manual) provisioned throughput. The closer the number is to 100%, the more you are fully using your provisioned RU/s. Learn more about the metric.

Next, determine how the normalized utilization varies over time. Find the highest normalized utilization for each hour. Then, calculate the average normalized utilization across all hours. If you see that your average utilization is less than 66%, consider enabling autoscale on your database or container. In contrast, if the average utilization is greater than 66%, it's recommended to remain on standard (manual) provisioned throughput.

> **TIP**
> If your account is configured to use multi-region writes and has more than one region, the rate per 100 RU/s is the same for both manual and autoscale. This means that enabling autoscale incurs no additional cost regardless of utilization. As a result, it is always recommended to use autoscale with multi-region writes when you have more than one region, to take advantage of the savings from paying only for the RU/s your application scales to. If you have multi-region writes and one region, use the average utilization to determine if autoscale will result in cost savings.

Examples

Let's take a look at two different example workloads and analyze if they are suitable for manual or autoscale throughput. To illustrate the general approach, we'll analyze three hours of history to determine the cost difference between using manual and autoscale. For production workloads, it's recommended to use 7 to 30 days of history (or longer if available) to establish a pattern of RU/s usage.

> **NOTE**
> All the examples shown in this doc are based on the price for an Azure Cosmos account deployed in a non-government region in the US. The pricing and calculation vary depending on the region you are using, see the Azure Cosmos DB pricing page for the latest pricing information.

Assumptions:

- Suppose we currently have manual throughput of 30,000 RU/s.
- Our region is configured with single-region writes, with one region. If we had multiple regions, we would multiply the hourly cost by the number of regions.
- Use public pricing rates for manual ($0.008 USD per 100 RU/s per hour) and autoscale throughput ($0.012 USD per 100 RU/s per hour) in single-region write accounts. See pricing page for details.

Example 1: Variable workload (autoscale recommended)

First, we look at the normalized RU consumption. This workload has variable traffic, with normalized RU consumption ranging from 6% to 100%. There are occasional spikes to 100% that are hard to predict, but many

hours with low utilization.

Let's compare the cost of provisioning 30,000 RU/s manual throughput, versus setting autoscale max RU/s to 30,000 (scales between 3000 - 30,000 RU/s).

Now, let's analyze the history. Suppose we have the utilization described in the following table. The average utilization across these three hours is 39%. Because the normalized RU consumption averages to less than 66%, we save by using autoscale.

Note that in hour 1, when there is 6% usage, autoscale will bill RU/s for 10% of the max RU/s, which is the minimum per hour. Though the cost of autoscale may be higher than manual throughput in certain hours, as long as the average utilization is less than 66% across all hours, autoscale will be cheaper overall.

TIME PERIOD	UTILIZATION	BILLED AUTOSCALE RU/S	OPTION 1: MANUAL 30,000 RU/S	OPTION 2: AUTOSCALE BETWEEN 3000 - 30,000 RU/S
Hour 1	6%	3000	30,000 * 0.008 / 100 = $2.40	3000 * 0.012 / 100 = $0.36
Hour 2	100%	30,000	30,000 * 0.008 / 100 = $2.40	30,000 * 0.012 / 100 = $3.60
Hour 3	11%	3300	30,000 * 0.008 / 100 = $2.40	3300 * 0.012 / 100 = $0.40
Total			$7.20	$4.36 (39% savings)

Example 2: Steady workload (manual throughput recommended)

This workload has steady traffic, with normalized RU consumption ranging from 72% to 100%. With 30,000 RU/s provisioned, this means that we are consuming between 21,600 to 30,000 RU/s.

Let's compare the cost of provisioning 30,000 RU/s manual throughput, versus setting autoscale max RU/s to 30,000 (scales between 3000 - 30,000 RU/s).

Suppose we have the utilization history as described in the table. Our average utilization across these three hours is 88%. Because the normalized RU consumption averages to greater than 66%, we save by using manual throughput.

In general, if the average utilization across all 730 hours in one month is greater than 66%, then we'll save by using manual throughput.

TIME PERIOD	UTILIZATION	BILLED AUTOSCALE RU/S	OPTION 1: MANUAL 30,000 RU/S	OPTION 2: AUTOSCALE BETWEEN 3000 - 30,000 RU/S
Hour 1	72%	21,600	30,000 * 0.008 / 100 = $2.40	21600 * 0.012 / 100 = $2.59
Hour 2	93%	28,000	30,000 * 0.008 / 100 = $2.40	28,000 * 0.012 / 100 = $3.36
Hour 3	100%	30,000	30,000 * 0.008 / 100 = $2.40	30,000 * 0.012 / 100 = $3.60
Total			$7.20	$9.55

TIP

With standard (manual) throughput, you can use the normalized utilization metric to estimate the actual RU/s you may use if you switch to autoscale. Multiply the normalized utilization at a point in time by the currently provisioned standard (manual) RU/s. For example, if you have provisioned 5000 RU/s, and the normalized utilization is 90%, the RU/s usage is 0.9 * 5000 = 4500 RU/s. If you see that your traffic pattern is variable, but you are over or under provisioned, you may want to enable autoscale and then change the autoscale max RU/s setting accordingly.

How to calculate average utilization

Autoscale bills for the highest RU/s scaled to in an hour. When analyzing the normalized RU consumption over time, it is important to use the highest utilization per hour when calculating the average.

To calculate the average of the highest utilization across all hours:

1. Set the **Aggregation** on the Noramlized RU Consumption metric to **Max**.
2. Select the **Time granularity** to 1 hour.
3. Navigate to **Chart options**.
4. Select the bar chart option.
5. Under **Share**, select the **Download to Excel** option. From the generated spreadsheet, calculate the average utilization across all hours.

Measure and monitor your usage

Over time, after you've chosen the throughput type, you should monitor your application and make adjustments as needed.

When using autoscale, use Azure Monitor to see the provisioned autoscale max RU/s (**Autoscale Max Throughput**) and the RU/s the system is currently scaled to (**Provisioned Throughput**). Below is an example of a variable or unpredictable workload using autoscale. Note when there isn't any traffic, the system scales the RU/s to the minimum of 10% of the max RU/s, which in this case is 5000 RU/s and 50,000 RU/s, respectively.

> **NOTE**
>
> When you use standard (manual) provisioned throughput, the **Provisioned Throughput** metric refers to what you as a user have set. When you use autoscale throughput, this metric refers to the RU/s the system is currently scaled to.

Next steps

- Use RU calculator to estimate throughput for new workloads.
- Use Azure Monitor to monitor your existing workloads.
- Learn how to provision autoscale throughput on an Azure Cosmos database or container.
- Review the autoscale FAQ.

How to choose between provisioned throughput and serverless

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: SQL API Cassandra API Gremlin API Table API Azure Cosmos DB API for MongoDB

Azure Cosmos DB is available in two different capacity modes: provisioned throughput and serverless. You can perform the exact same database operations in both modes, but the way you get billed for these operations is radically different. The following video explains the core differences between these modes and how they fit different types of workloads:

Detailed comparison

CRITERIA	PROVISIONED THROUGHPUT	SERVERLESS
Status	Generally available	In preview
Best suited for	Mission-critical workloads requiring predictable performance	Small-to-medium non-critical workloads with light and intermittent traffic
How it works	For each of your containers, you provision some amount of throughput expressed in Request Units per second. Every second, this amount of Request Units is available for your database operations. Provisioned throughput can be updated manually or adjusted automatically with autoscale.	You run your database operations against your containers without having to provision any capacity.
Geo-distribution	Available (unlimited number of Azure regions)	Unavailable (serverless accounts can only run in 1 Azure region)
Maximum storage per container	Unlimited	50 GB
Performance	99.99% to 99.999% availability covered by SLA < 10 ms latency for point-reads and writes covered by SLA 99.99% guaranteed throughput covered by SLA	99.9% to 99.99% availability covered by SLA < 10 ms latency for point-reads and < 30 ms for writes covered by SLO 95% burstability covered by SLO
Billing model	Billing is done on a per-hour basis for the RU/s provisioned, regardless of how many RUs were consumed.	Billing is done on a per-hour basis for the amount of RUs consumed by your database operations.

> **IMPORTANT**
> Some of the serverless limitations may be eased or removed when serverless becomes generally available and **your feedback** will help us decide! Reach out and tell us more about your serverless experience: azurecosmosdbserverless@service.microsoft.com.

Estimating your expected consumption

In some situations, it may be unclear whether provisioned throughput or serverless should be chosen for a given workload. To help with this decision, you can estimate your overall **expected consumption**, that is what's the total number of RUs you may consume over a month (you can estimate this with the help of the table shown here)

Example 1: a workload is expected to burst to a maximum of 500 RU/s and consume a total of 20,000,000 RUs over a month.

- In provisioned throughput mode, you would provision a container with 500 RU/s for a monthly cost of: $0.008 * 5 * 730 = **$29.20**
- In serverless mode, you would pay for the consumed RUs: $0.25 * 20 = **$5.00**

Example 2: a workload is expected to burst to a maximum of 500 RU/s and consume a total of 250,000,000 RUs over a month.

- In provisioned throughput mode, you would provision a container with 500 RU/s for a monthly cost of: $0.008 * 5 * 730 = **$29.20**
- In serverless mode, you would pay for the consumed RUs: $0.25 * 250 = **$62.50**

(these examples are not accounting for the storage cost, which is the same between the two modes)

> **NOTE**
> The costs shown in the previous example are for demonstration purposes only. See the pricing page for the latest pricing information.

Next steps

- Read more about provisioning throughput on Azure Cosmos DB
- Read more about Azure Cosmos DB serverless
- Get familiar with the concept of Request Units

Security in Azure Cosmos DB - overview

12/4/2020 • 8 minutes to read • Edit Online

APPLIES TO: SQL API Cassandra API Gremlin API Table API Azure Cosmos DB API for MongoDB

This article discusses database security best practices and key features offered by Azure Cosmos DB to help you prevent, detect, and respond to database breaches.

What's new in Azure Cosmos DB security

Encryption at rest is now available for documents and backups stored in Azure Cosmos DB in all Azure regions. Encryption at rest is applied automatically for both new and existing customers in these regions. There is no need to configure anything; and you get the same great latency, throughput, availability, and functionality as before with the benefit of knowing your data is safe and secure with encryption at rest.

How do I secure my database

Data security is a shared responsibility between you, the customer, and your database provider. Depending on the database provider you choose, the amount of responsibility you carry can vary. If you choose an on-premises solution, you need to provide everything from end-point protection to physical security of your hardware - which is no easy task. If you choose a PaaS cloud database provider such as Azure Cosmos DB, your area of concern shrinks considerably. The following image, borrowed from Microsoft's Shared Responsibilities for Cloud Computing white paper, shows how your responsibility decreases with a PaaS provider like Azure Cosmos DB.

The preceding diagram shows high-level cloud security components, but what items do you need to worry about specifically for your database solution? And how can you compare solutions to each other?

We recommend the following checklist of requirements on which to compare database systems:

- Network security and firewall settings
- User authentication and fine grained user controls
- Ability to replicate data globally for regional failures
- Ability to fail over from one data center to another
- Local data replication within a data center
- Automatic data backups
- Restoration of deleted data from backups
- Protect and isolate sensitive data
- Monitoring for attacks
- Responding to attacks
- Ability to geo-fence data to adhere to data governance restrictions
- Physical protection of servers in protected data centers
- Certifications

And although it may seem obvious, recent large-scale database breaches remind us of the simple but critical importance of the following requirements:

- Patched servers that are kept up-to-date
- HTTPS by default/TLS encryption
- Administrative accounts with strong passwords

How does Azure Cosmos DB secure my database

Let's look back at the preceding list - how many of those security requirements does Azure Cosmos DB provide? Every single one.

Let's dig into each one in detail.

SECURITY REQUIREMENT	AZURE COSMOS DB'S SECURITY APPROACH
Network security	Using an IP firewall is the first layer of protection to secure your database. Azure Cosmos DB supports policy driven IP-based access controls for inbound firewall support. The IP-based access controls are similar to the firewall rules used by traditional database systems, but they are expanded so that an Azure Cosmos database account is only accessible from an approved set of machines or cloud services. Learn more in Azure Cosmos DB firewall support article. Azure Cosmos DB enables you to enable a specific IP address (168.61.48.0), an IP range (168.61.48.0/8), and combinations of IPs and ranges. All requests originating from machines outside this allowed list are blocked by Azure Cosmos DB. Requests from approved machines and cloud services then must complete the authentication process to be given access control to the resources. You can use virtual network service tags to achieve network isolation and protect your Azure Cosmos DB resources from the general Internet. Use service tags in place of specific IP addresses when you create security rules. By specifying the service tag name (for example, AzureCosmosDB) in the appropriate source or destination field of a rule, you can allow or deny the traffic for the corresponding service.

SECURITY REQUIREMENT	AZURE COSMOS DB'S SECURITY APPROACH
Authorization	Azure Cosmos DB uses hash-based message authentication code (HMAC) for authorization. Each request is hashed using the secret account key, and the subsequent base-64 encoded hash is sent with each call to Azure Cosmos DB. To validate the request, the Azure Cosmos DB service uses the correct secret key and properties to generate a hash, then it compares the value with the one in the request. If the two values match, the operation is authorized successfully and the request is processed, otherwise there is an authorization failure and the request is rejected. You can use either a primary key, or a resource token allowing fine-grained access to a resource such as a document. Learn more in Securing access to Azure Cosmos DB resources.
Users and permissions	Using the primary key for the account, you can create user resources and permission resources per database. A resource token is associated with a permission in a database and determines whether the user has access (read-write, read-only, or no access) to an application resource in the database. Application resources include container, documents, attachments, stored procedures, triggers, and UDFs. The resource token is then used during authentication to provide or deny access to the resource. Learn more in Securing access to Azure Cosmos DB resources.
Active directory integration (Azure RBAC)	You can also provide or restrict access to the Cosmos account, database, container, and offers (throughput) using Access control (IAM) in the Azure portal. IAM provides role-based access control and integrates with Active Directory. You can use built in roles or custom roles for individuals and groups. See Active Directory integration article for more information.
Global replication	Azure Cosmos DB offers turnkey global distribution, which enables you to replicate your data to any one of Azure's world-wide datacenters with the click of a button. Global replication lets you scale globally and provide low-latency access to your data around the world. In the context of security, global replication ensures data protection against regional failures. Learn more in Distribute data globally.
Regional failovers	If you have replicated your data in more than one data center, Azure Cosmos DB automatically rolls over your operations should a regional data center go offline. You can create a prioritized list of failover regions using the regions in which your data is replicated. Learn more in Regional Failovers in Azure Cosmos DB.

SECURITY REQUIREMENT	AZURE COSMOS DB'S SECURITY APPROACH
Local replication	Even within a single data center, Azure Cosmos DB automatically replicates data for high availability giving you the choice of consistency levels. This replication guarantees a99.99% availability SLA for all single region accounts and all multi-region accounts with relaxed consistency, and 99.999% read availability on all multi-region database accounts.
Automated online backups	Azure Cosmos databases are backed up regularly and stored in a geo redundant store. Learn more in Automatic online backup and restore with Azure Cosmos DB.
Restore deleted data	The automated online backups can be used to recover data you may have accidentally deleted up to ~30 days after the event. Learn more in Automatic online backup and restore with Azure Cosmos DB
Protect and isolate sensitive data	All data in the regions listed in What's new? is now encrypted at rest. Personal data and other confidential data can be isolated to specific container and read-write, or read-only access can be limited to specific users.
Monitor for attacks	By using audit logging and activity logs, you can monitor your account for normal and abnormal activity. You can view what operations were performed on your resources, who initiated the operation, when the operation occurred, the status of the operation, and much more as shown in the screenshot following this table.
Respond to attacks	Once you have contacted Azure support to report a potential attack, a 5-step incident response process is kicked off. The goal of the 5-step process is to restore normal service security and operations as quickly as possible after an issue is detected and an investigation is started. Learn more in Microsoft Azure Security Response in the Cloud.
Geo-fencing	Azure Cosmos DB ensures data governance for sovereign regions (for example, Germany, China, US Gov).
Protected facilities	Data in Azure Cosmos DB is stored on SSDs in Azure's protected data centers. Learn more in Microsoft global datacenters
HTTPS/SSL/TLS encryption	All connections to Azure Cosmos DB support HTTPS. Azure Cosmos DB also supports TLS 1.2. It is possible to enforce a minimum TLS version server-side. To do so, please contact azurecosmosdbtls@service.microsoft.com.

SECURITY REQUIREMENT	AZURE COSMOS DB'S SECURITY APPROACH
Encryption at rest	All data stored into Azure Cosmos DB is encrypted at rest. Learn more in Azure Cosmos DB encryption at rest
Patched servers	As a managed database, Azure Cosmos DB eliminates the need to manage and patch servers, that's done for you, automatically.
Administrative accounts with strong passwords	It's hard to believe we even need to mention this requirement, but unlike some of our competitors, it's impossible to have an administrative account with no password in Azure Cosmos DB. Security via TLS and HMAC secret based authentication is baked in by default.
Security and data protection certifications	For the most up-to-date list of certifications see the overall Azure Compliance site as well as the latest Azure Compliance Document with all certifications (search for Cosmos). For a more focused read check out the April 25, 2018 post [Azure #CosmosDB: Secure, private, compliant that includes SOCS 1/2 Type 2, HITRUST, PCI DSS Level 1, ISO 27001, HIPAA, FedRAMP High, and many others.

The following screenshot shows how you can use audit logging and activity logs to monitor your account:

Primary keys

Primary keys provide access to all the administrative resources for the database account. Primary keys:

- Provide access to accounts, databases, users, and permissions.
- Cannot be used to provide granular access to containers and documents.

- Are created during the creation of an account.
- Can be regenerated at any time.

Each account consists of two primary keys: a primary key and secondary key. The purpose of dual keys is so that you can regenerate, or roll keys, providing continuous access to your account and data.

In addition to the two primary keys for the Cosmos DB account, there are two read-only keys. These read-only keys only allow read operations on the account. Read-only keys do not provide access to read permissions resources.

Primary, secondary, read only, and read-write primary keys can be retrieved and regenerated using the Azure portal. For instructions, see View, copy, and regenerate access keys.

Next steps

For more information about primary keys and resource tokens, see Securing access to Azure Cosmos DB data.

For more information about audit logging, see Azure Cosmos DB diagnostic logging.

For more information about Microsoft certifications, see Azure Trust Center.

Data encryption in Azure Cosmos DB

11/2/2020 • 3 minutes to read • Edit Online

APPLIES TO: SQL API Cassandra API Gremlin API Table API Azure Cosmos DB API for MongoDB

Encryption at rest is a phrase that commonly refers to the encryption of data on nonvolatile storage devices, such as solid state drives (SSDs) and hard disk drives (HDDs). Cosmos DB stores its primary databases on SSDs. Its media attachments and backups are stored in Azure Blob storage, which is generally backed up by HDDs. With the release of encryption at rest for Cosmos DB, all your databases, media attachments, and backups are encrypted. Your data is now encrypted in transit (over the network) and at rest (nonvolatile storage), giving you end-to-end encryption.

As a PaaS service, Azure Cosmos DB is very easy to use. Because all user data stored in Azure Cosmos DB is encrypted at rest and in transport, you don't have to take any action. Another way to put this is that encryption at rest is "on" by default. There are no controls to turn it off or on. Azure Cosmos DB uses AES-256 encryption on all regions where the account is running. We provide this feature while we continue to meet our availability and performance SLAs. Data stored in your Azure Cosmos account is automatically and seamlessly encrypted with keys managed by Microsoft (service-managed keys). Optionally, you can choose to add a second layer of encryption with your own keys as described in the customer-managed keys article.

Implementation of encryption at rest for Azure Cosmos DB

Encryption at rest is implemented by using a number of security technologies, including secure key storage systems, encrypted networks, and cryptographic APIs. Systems that decrypt and process data have to communicate with systems that manage keys. The diagram shows how storage of encrypted data and the management of keys is separated.

The basic flow of a user request is as follows:

- The user database account is made ready, and storage keys are retrieved via a request to the Management Service Resource Provider.
- A user creates a connection to Cosmos DB via HTTPS/secure transport. (The SDKs abstract the details.)
- The user sends a JSON document to be stored over the previously created secure connection.
- The JSON document is indexed unless the user has turned off indexing.
- Both the JSON document and index data are written to secure storage.
- Periodically, data is read from the secure storage and backed up to the Azure Encrypted Blob Store.

Frequently asked questions

Q: How much more does Azure Storage cost if Storage Service Encryption is enabled?

A: There is no additional cost.

Q: Who manages the encryption keys?

A: The keys are managed by Microsoft.

Q: How often are encryption keys rotated?

A: Microsoft has a set of internal guidelines for encryption key rotation, which Cosmos DB follows. The specific guidelines are not published. Microsoft does publish the Security Development Lifecycle (SDL), which is seen as a subset of internal guidance and has useful best practices for developers.

Q: Can I use my own encryption keys?

A: Yes, this feature is now available for new Azure Cosmos DB accounts and this should be done at the time of account creation. Please go through Customer-managed Keys document for more information.

Q: What regions have encryption turned on?

A: All Azure Cosmos DB regions have encryption turned on for all user data.

Q: Does encryption affect the performance latency and throughput SLAs?

A: There is no impact or changes to the performance SLAs now that encryption at rest is enabled for all existing and new accounts. You can read more on the SLA for Cosmos DB page to see the latest guarantees.

Q: Does the local emulator support encryption at rest?

A: The emulator is a standalone dev/test tool and does not use the key management services that the managed Cosmos DB service uses. Our recommendation is to enable BitLocker on drives where you are storing sensitive emulator test data. The emulator supports changing the default data directory as well as using a well-known location.

Next steps

- You can choose to add a second layer of encryption with your own keys, to learn more, see the customer-managed keys article.
- For an overview of Cosmos DB security and the latest improvements, see Azure Cosmos database security.
- For more information about Microsoft certifications, see the Azure Trust Center.

Virtual network service tags

12/4/2020 • 9 minutes to read • Edit Online

A service tag represents a group of IP address prefixes from a given Azure service. Microsoft manages the address prefixes encompassed by the service tag and automatically updates the service tag as addresses change, minimizing the complexity of frequent updates to network security rules.

You can use service tags to define network access controls on network security groups or Azure Firewall. Use service tags in place of specific IP addresses when you create security rules. By specifying the service tag name, such as **ApiManagement**, in the appropriate *source* or *destination* field of a rule, you can allow or deny the traffic for the corresponding service.

You can use service tags to achieve network isolation and protect your Azure resources from the general Internet while accessing Azure services that have public endpoints. Create inbound/outbound network security group rules to deny traffic to/from **Internet** and allow traffic to/from **AzureCloud** or other available service tags of specific Azure services.

Network Security Group (NSG)				
Action	Name	Source	Destination	Port
Allow	AllowStorage	VirtualNetwork	**Storage**	Any
Allow	AllowSQL	VirtualNetwork	**Sql.EastUS**	Any
Deny	DenyAllOutBound	Any	Any	Any

Available service tags

The following table includes all the service tags available for use in network security group rules.

The columns indicate whether the tag:

- Is suitable for rules that cover inbound or outbound traffic.
- Supports regional scope.
- Is usable in Azure Firewall rules.

By default, service tags reflect the ranges for the entire cloud. Some service tags also allow more granular control by restricting the corresponding IP ranges to a specified region. For example, the service tag **Storage** represents Azure Storage for the entire cloud, but **Storage.WestUS** narrows the range to only the storage IP address ranges from the WestUS region. The following table indicates whether each service tag supports such regional scope.

TAG	PURPOSE	CAN USE INBOUND OR OUTBOUND?	CAN BE REGIONAL?	CAN USE WITH AZURE FIREWALL?
ActionGroup	Action Group.	Inbound	No	No
ApiManagement	Management traffic for Azure API Management-dedicated deployments. *Note:* This tag represents the Azure API Management service endpoint for control plane per region. This enables customers to perform management operations on the APIs, Operations, Policies, NamedValues configured on the API Management service.	Inbound	Yes	Yes
ApplicationInsightsAvailability	Application Insights Availability.	Inbound	No	No
AppConfiguration	App Configuration.	Outbound	No	No
AppService	Azure App Service. This tag is recommended for outbound security rules to web apps and Function apps.	Outbound	Yes	Yes
AppServiceManagement	Management traffic for deployments dedicated to App Service Environment.	Both	No	Yes
AzureActiveDirectory	Azure Active Directory.	Outbound	No	Yes
AzureActiveDirectoryDomainServices	Management traffic for deployments dedicated to Azure Active Directory Domain Services.	Both	No	Yes
AzureAdvancedThreatProtection	Azure Advanced Threat Protection.	Outbound	No	No

TAG	PURPOSE	CAN USE INBOUND OR OUTBOUND?	CAN BE REGIONAL?	CAN USE WITH AZURE FIREWALL?
AzureBackup	Azure Backup. *Note:* This tag has a dependency on the **Storage** and **AzureActiveDirectory** tags.	Outbound	No	Yes
AzureBotService	Azure Bot Service.	Outbound	No	No
AzureCloud	All datacenter public IP addresses.	Outbound	Yes	Yes
AzureCognitiveSearch	Azure Cognitive Search. This tag or the IP addresses covered by this tag can be used to grant indexers secure access to data sources. Refer to the indexer connection documentation for more details. *Note*: The IP of the search service is not included in the list of IP ranges for this service tag and **also needs to be added** to the IP firewall of data sources.	Inbound	No	No
AzureConnectors	Azure Logic Apps connectors for probe/back-end connections.	Inbound	Yes	Yes
AzureContainerRegistry	Azure Container Registry.	Outbound	Yes	Yes
AzureCosmosDB	Azure Cosmos DB.	Outbound	Yes	Yes
AzureDatabricks	Azure Databricks.	Both	No	No
AzureDataExplorerManagement	Azure Data Explorer Management.	Inbound	No	No
AzureDataLake	Azure Data Lake Storage Gen1.	Outbound	No	Yes
AzureDevSpaces	Azure Dev Spaces.	Outbound	No	No

TAG	PURPOSE	CAN USE INBOUND OR OUTBOUND?	CAN BE REGIONAL?	CAN USE WITH AZURE FIREWALL?
AzureDevOps	Azure Dev Ops. *Note: This tag is not currently configurable via Azure Portal*	Inbound	No	Yes
AzureDigitalTwins	Azure Digital Twins. *Note:* This tag or the IP addresses covered by this tag can be used to restrict access to endpoints configured for event routes. *This tag is not currently configurable via Azure Portal*	Inbound	No	Yes
AzureEventGrid	Azure Event Grid.	Both	No	No
AzureFrontDoor.Frontend **AzureFrontDoor.Backend** **AzureFrontDoor.FirstParty**	Azure Front Door.	Both	No	No
AzureInformationProtection	Azure Information Protection. *Note:* This tag has a dependency on the **AzureActiveDirectory**, **AzureFrontDoor.Frontend** and **AzureFrontDoor.FirstParty** tags.	Outbound	No	No
AzureIoTHub	Azure IoT Hub.	Outbound	No	No
AzureKeyVault	Azure Key Vault. *Note:* This tag has a dependency on the **AzureActiveDirectory** tag.	Outbound	Yes	Yes

TAG	PURPOSE	CAN USE INBOUND OR OUTBOUND?	CAN BE REGIONAL?	CAN USE WITH AZURE FIREWALL?
AzureLoadBalancer	The Azure infrastructure load balancer. The tag translates to the virtual IP address of the host (168.63.129.16) where the Azure health probes originate. This only includes probe traffic, not real traffic to your backend resource. If you're not using Azure Load Balancer, you can override this rule.	Both	No	No
AzureMachineLearning	Azure Machine Learning.	Both	No	Yes
AzureMonitor	Log Analytics, Application Insights, AzMon, and custom metrics (GiG endpoints). *Note:* For Log Analytics, this tag has a dependency on the **Storage** tag.	Outbound	No	Yes
AzureOpenDatasets	Azure Open Datasets. *Note:* This tag has a dependency on the **AzureFrontDoor.Frontend** and **Storage** tag.	Outbound	No	No
AzurePlatformDNS	The basic infrastructure (default) DNS service. You can use this tag to disable the default DNS. Be cautious when you use this tag. We recommend that you read Azure platform considerations. We also recommend that you perform testing before you use this tag.	Outbound	No	No

TAG	PURPOSE	CAN USE INBOUND OR OUTBOUND?	CAN BE REGIONAL?	CAN USE WITH AZURE FIREWALL?
AzurePlatformIMDS	Azure Instance Metadata Service (IMDS), which is a basic infrastructure service. You can use this tag to disable the default IMDS. Be cautious when you use this tag. We recommend that you read Azure platform considerations. We also recommend that you perform testing before you use this tag.	Outbound	No	No
AzurePlatformLKM	Windows licensing or key management service. You can use this tag to disable the defaults for licensing. Be cautious when you use this tag. We recommend that you read Azure platform considerations. We also recommend that you perform testing before you use this tag.	Outbound	No	No
AzureResourceManager	Azure Resource Manager.	Outbound	No	No
AzureSignalR	Azure SignalR.	Outbound	No	No
AzureSiteRecovery	Azure Site Recovery. *Note:* This tag has a dependency on the **AzureActiveDirectory**, **AzureKeyVault**, **EventHub**,**GuestAndHybridManagement** and **Storage** tags.	Outbound	No	No

TAG	PURPOSE	CAN USE INBOUND OR OUTBOUND?	CAN BE REGIONAL?	CAN USE WITH AZURE FIREWALL?
AzureTrafficManager	Azure Traffic Manager probe IP addresses. For more information on Traffic Manager probe IP addresses, see Azure Traffic Manager FAQ.	Inbound	No	Yes
BatchNodeManagement	Management traffic for deployments dedicated to Azure Batch.	Both	No	Yes
CognitiveServicesManagement	The address ranges for traffic for Azure Cognitive Services.	Both	No	No
DataFactory	Azure Data Factory	Both	No	No
DataFactoryManagement	Management traffic for Azure Data Factory.	Outbound	No	No
Dynamics365ForMarketingEmail	The address ranges for the marketing email service of Dynamics 365.	Outbound	Yes	No
EventHub	Azure Event Hubs.	Outbound	Yes	Yes
GatewayManager	Management traffic for deployments dedicated to Azure VPN Gateway and Application Gateway.	Inbound	No	No
GuestAndHybridManagement	Azure Automation and Guest Configuration.	Outbound	No	Yes
HDInsight	Azure HDInsight.	Inbound	Yes	No
Internet	The IP address space that's outside the virtual network and reachable by the public internet. The address range includes the Azure-owned public IP address space.	Both	No	No
LogicApps	Logic Apps.	Both	No	No

TAG	PURPOSE	CAN USE INBOUND OR OUTBOUND?	CAN BE REGIONAL?	CAN USE WITH AZURE FIREWALL?
LogicAppsManagement	Management traffic for Logic Apps.	Inbound	No	No
MicrosoftCloudAppSecurity	Microsoft Cloud App Security.	Outbound	No	No
MicrosoftContainerRegistry	Container registry for Microsoft container images. *Note:* This tag has a dependency on the **AzureFrontDoor.FirstParty** tag.	Outbound	Yes	Yes
PowerQueryOnline	Power Query Online.	Both	No	No
ServiceBus	Azure Service Bus traffic that uses the Premium service tier.	Outbound	Yes	Yes
ServiceFabric	Azure Service Fabric. *Note:* This tag represents the Service Fabric service endpoint for control plane per region. This enables customers to perform management operations for their Service Fabric clusters from their VNET (endpoint eg. https://westus.servicefabric.azure.com)	Both	No	No

TAG	PURPOSE	CAN USE INBOUND OR OUTBOUND?	CAN BE REGIONAL?	CAN USE WITH AZURE FIREWALL?
Sql	Azure SQL Database, Azure Database for MySQL, Azure Database for PostgreSQL, and Azure Synapse Analytics. *Note:* This tag represents the service, but not specific instances of the service. For example, the tag represents the Azure SQL Database service, but not a specific SQL database or server. This tag does not apply to SQL managed instance.	Outbound	Yes	Yes
SqlManagement	Management traffic for SQL-dedicated deployments.	Both	No	Yes
Storage	Azure Storage. *Note:* This tag represents the service, but not specific instances of the service. For example, the tag represents the Azure Storage service, but not a specific Azure Storage account.	Outbound	Yes	Yes
StorageSyncService	Storage Sync Service.	Both	No	No
WindowsVirtualDesktop	Windows Virtual Desktop.	Both	No	Yes

TAG	PURPOSE	CAN USE INBOUND OR OUTBOUND?	CAN BE REGIONAL?	CAN USE WITH AZURE FIREWALL?
VirtualNetwork	The virtual network address space (all IP address ranges defined for the virtual network), all connected on-premises address spaces, peered virtual networks, virtual networks connected to a virtual network gateway, the virtual IP address of the host, and address prefixes used on user-defined routes. This tag might also contain default routes.	Both	No	No

NOTE

In the classic deployment model (before Azure Resource Manager), a subset of the tags listed in the previous table are supported. These tags are spelled differently:

CLASSIC SPELLING	EQUIVALENT RESOURCE MANAGER TAG
AZURE_LOADBALANCER	AzureLoadBalancer
INTERNET	Internet
VIRTUAL_NETWORK	VirtualNetwork

NOTE

Service tags of Azure services denote the address prefixes from the specific cloud being used. For example, the underlying IP ranges that correspond to the **Sql** tag value on the Azure Public cloud will be different from the underlying ranges on the Azure China cloud.

NOTE

If you implement a virtual network service endpoint for a service, such as Azure Storage or Azure SQL Database, Azure adds a route to a virtual network subnet for the service. The address prefixes in the route are the same address prefixes, or CIDR ranges, as those of the corresponding service tag.

Service tags on-premises

You can obtain the current service tag and range information to include as part of your on-premises firewall configurations. This information is the current point-in-time list of the IP ranges that correspond to each service tag. You can obtain the information programmatically or via a JSON file download, as described in the following sections.

Use the Service Tag Discovery API (public preview)

You can programmatically retrieve the current list of service tags together with IP address range details:

- REST
- Azure PowerShell
- Azure CLI

> **NOTE**
> While it's in public preview, the Discovery API might return information that's less current than information returned by the JSON downloads. (See the next section.)

Discover service tags by using downloadable JSON files

You can download JSON files that contain the current list of service tags together with IP address range details. These lists are updated and published weekly. Locations for each cloud are:

- Azure Public
- Azure US Government
- Azure China
- Azure Germany

The IP address ranges in these files are in CIDR notation.

> **NOTE**
> A subset of this information has been published in XML files for Azure Public, Azure China, and Azure Germany. These XML downloads will be deprecated by June 30, 2020 and will no longer be available after that date. You should migrate to using the Discovery API or JSON file downloads as described in the previous sections.

Tips

- You can detect updates from one publication to the next by noting increased *changeNumber* values in the JSON file. Each subsection (for example, **Storage.WestUS**) has its own *changeNumber* that's incremented as changes occur. The top level of the file's *changeNumber* is incremented when any of the subsections is changed.
- For examples of how to parse the service tag information (for example, get all address ranges for Storage in WestUS), see the Service Tag Discovery API PowerShell documentation.

Next steps

- Learn how to create a network security group.

Azure role-based access control in Azure Cosmos DB

12/4/2020 • 4 minutes to read • Edit Online

APPLIES TO: SQL API Cassandra API Gremlin API Table API Azure Cosmos DB API for MongoDB

Azure Cosmos DB provides built-in Azure role-based access control (Azure RBAC) for common management scenarios in Azure Cosmos DB. An individual who has a profile in Azure Active Directory can assign these Azure roles to users, groups, service principals, or managed identities to grant or deny access to resources and operations on Azure Cosmos DB resources. Role assignments are scoped to control-plane access only, which includes access to Azure Cosmos accounts, databases, containers, and offers (throughput).

Built-in roles

The following are the built-in roles supported by Azure Cosmos DB:

BUILT-IN ROLE	DESCRIPTION
DocumentDB Account Contributor	Can manage Azure Cosmos DB accounts.
Cosmos DB Account Reader	Can read Azure Cosmos DB account data.
Cosmos Backup Operator	Can submit restore request for an Azure Cosmos database or a container. Cannot access any data or use Data Explorer.
Cosmos DB Operator	Can provision Azure Cosmos accounts, databases, and containers. Cannot access any data or use Data Explorer.

IMPORTANT

Azure RBAC support in Azure Cosmos DB applies to control plane operations only. Data plane operations are secured using primary keys or resource tokens. To learn more, see Secure access to data in Azure Cosmos DB

Identity and access management (IAM)

The **Access control (IAM)** pane in the Azure portal is used to configure Azure role-based access control on Azure Cosmos resources. The roles are applied to users, groups, service principals, and managed identities in Active Directory. You can use built-in roles or custom roles for individuals and groups. The following screenshot shows Active Directory integration (Azure RBAC) using access control (IAM) in the Azure portal:

Custom roles

In addition to the built-in roles, users may also create custom roles in Azure and apply these roles to service principals across all subscriptions within their Active Directory tenant. Custom roles provide users a way to create Azure role definitions with a custom set of resource provider operations. To learn which operations are available for building custom roles for Azure Cosmos DB see, Azure Cosmos DB resource provider operations

> **TIP**
>
> Custom roles that need to access data stored within Cosmos DB or use Data Explorer in the Azure portal must have `Microsoft.DocumentDB/databaseAccounts/listKeys/*` action.

Preventing changes from the Azure Cosmos DB SDKs

The Azure Cosmos DB resource provider can be locked down to prevent any changes to resources from a client connecting using the account keys (that is applications connecting via the Azure Cosmos SDK). This also includes changes made from the Azure portal. This feature may be desirable for users who want higher degrees of control and governance for production environments. Preventing changes from the SDK also enables features such as resource locks and diagnostic logs for control plane operations. The clients connecting from Azure Cosmos DB SDK will be prevented from changing any property for the Azure Cosmos accounts, databases, containers, and throughput. The operations involving reading and writing data to Cosmos containers themselves are not impacted.

When this feature is enabled, changes to any resource can only be made from a user with the right Azure role and Azure Active Directory credentials including Managed Service Identities.

> **WARNING**
>
> Enabling this feature can have impact on your application. Make sure that you understand the impact before enabling it.

Check list before enabling

This setting will prevent any changes to any Cosmos resource from any client connecting using account keys including any Cosmos DB SDK, any tools that connect via account keys, or from the Azure portal. To prevent issues or errors from applications after enabling this feature, check if applications or Azure portal users perform any of the following actions before enabling this feature, including:

- Any change to the Cosmos account including any properties or adding or removing regions.
- Creating, deleting child resources such as databases and containers. This includes resources for other APIs

such as Cassandra, MongoDB, Gremlin, and table resources.

- Updating throughput on database or container level resources.
- Modifying container properties including index policy, TTL and unique keys.
- Modifying stored procedures, triggers or user-defined functions.

If your applications (or users via Azure portal) perform any of these actions they will need to be migrated to execute via ARM Templates, PowerShell, Azure CLI, REST, or Azure Management Library. Note that Azure Management is available in multiple languages.

Set via ARM Template

To set this property using an ARM template, update your existing template or export a new template for your current deployment, then include the `"disableKeyBasedMetadataWriteAccess": true` to the properties for the `databaseAccounts` resources. Below is a basic example of an Azure Resource Manager template with this property setting.

```
{
    {
      "type": "Microsoft.DocumentDB/databaseAccounts",
      "name": "[variables('accountName')]",
      "apiVersion": "2020-04-01",
      "location": "[parameters('location')]",
      "kind": "GlobalDocumentDB",
      "properties": {
        "consistencyPolicy": "[variables('consistencyPolicy')[parameters('defaultConsistencyLevel')]]",
        "locations": "[variables('locations')]",
        "databaseAccountOfferType": "Standard",
        "disableKeyBasedMetadataWriteAccess": true
        }
    }
}
```

> **IMPORTANT**
>
> Make sure you include the other properties for your account and child resources when redploying with this property. Do not deploy this template as is or it will reset all of your account properties.

Set via Azure CLI

To enable using Azure CLI, use the command below:

```
az cosmosdb update  --name [CosmosDBAccountName] --resource-group [ResourceGroupName]  --disable-key-based-metadata-write-access true
```

Set via PowerShell

To enable using Azure PowerShell, use the command below:

```
Update-AzCosmosDBAccount -ResourceGroupName [ResourceGroupName] -Name [CosmosDBAccountName] -DisableKeyBasedMetadataWriteAccess true
```

Next steps

- What is Azure role-based access control (Azure RBAC)

- Azure custom roles
- Azure Cosmos DB resource provider operations

Use Azure Policy to implement governance and controls for Azure Cosmos DB resources

11/2/2020 • 4 minutes to read • Edit Online

APPLIES TO: SQL API Cassandra API Gremlin API Table API Azure Cosmos DB API for MongoDB

Azure Policy helps to enforce organizational governance standards, assess resource compliance, and implement automatic remediation. Common use cases include security, cost management, and configuration consistency.

Azure Policy provides built-in policy definitions. You can create custom policy definitions for scenarios that are not addressed by the built-in policy definitions. See the Azure Policy documentation for more details.

> **IMPORTANT**
>
> Azure Policy is enforced at the resource provider level for Azure services. Cosmos DB SDKs can perform most management operations on database, container and throughput resources that bypass Cosmos DB's resource provider, thus ignoring any policies created using Azure Policy. To ensure enforcement of policies see, Preventing changes from the Azure Cosmos DB SDKs

Assign a built-in policy definition

Policy definitions describe resource compliance conditions and the effect to take if a condition is met. Policy *assignments* are created from policy *definitions*. You can use built-in or custom policy definitions for your Azure Cosmos DB resources. Policy assignments are scoped to an Azure management group, an Azure subscription, or a resource group and they are applied to the resources within the selected scope. Optionally, you can exclude specific resources from the scope.

You can create policy assignments with the Azure portal, Azure PowerShell, Azure CLI, or ARM template.

To create a policy assignment from a built-in policy definition for Azure Cosmos DB, use the steps in create a policy assignment with the Azure portal article.

At the step to select a policy definition, enter `Cosmos DB` in the Search field to filter the list of available built-in policy definitions. Select one of the available built-in policy definitions, and then choose **Select** to continue creating the policy assignment.

> **TIP**
>
> You can also use the built-in policy definition names shown in the **Available Definitions** pane with Azure PowerShell, Azure CLI, or ARM templates to create policy assignments.

Create a custom policy definition

For specific scenarios that are not addressed by built-in policies, you can create a custom policy definition. Later you create a Policy *assignment* from your custom policy *definition*.

Property types and property aliases in policy rules

Use the custom policy definition steps to identify the resource properties and property aliases, which are required to create policy rules.

To identify Azure Cosmos DB specific property aliases, use the namespace `Microsoft.DocumentDB` with one of the methods shown in the custom policy definition steps article.

Use the Azure CLI:

```
# Login first with az login if not using Cloud Shell

# Get Azure Policy aliases for namespace Microsoft.DocumentDB
az provider show --namespace Microsoft.DocumentDB --expand "resourceTypes/aliases" --query
"resourceTypes[].aliases[].name"
```

Use Azure PowerShell:

```
# Login first with Connect-AzAccount if not using Cloud Shell

# Use Get-AzPolicyAlias to list aliases for Microsoft.DocumentDB namespace
(Get-AzPolicyAlias -NamespaceMatch 'Microsoft.DocumentDB').Aliases
```

These commands output the list of property alias names for Azure Cosmos DB property. The following is an excerpt from the output:

```
[
  "Microsoft.DocumentDB/databaseAccounts/sku.name",
  "Microsoft.DocumentDB/databaseAccounts/virtualNetworkRules[*]",
  "Microsoft.DocumentDB/databaseAccounts/virtualNetworkRules[*].id",
  "Microsoft.DocumentDB/databaseAccounts/isVirtualNetworkFilterEnabled",
  "Microsoft.DocumentDB/databaseAccounts/consistencyPolicy.defaultConsistencyLevel",
  "Microsoft.DocumentDB/databaseAccounts/enableAutomaticFailover",
  "Microsoft.DocumentDB/databaseAccounts/Locations",
  "Microsoft.DocumentDB/databaseAccounts/Locations[*]",
  "Microsoft.DocumentDB/databaseAccounts/Locations[*].locationName",
  "..."
]
```

You can use any of these property alias names in the custom policy definition rules.

The following is an example policy definition that checks if an Azure Cosmos DB account is configured for multiple write locations. The custom policy definition includes two rules: one to check for the specific type of property alias, and the second one for the specific property of the type, in this case the field that stores the multiple write location setting. Both rules use the alias names.

```
"policyRule": {
  "if": {
    "allOf": [
      {
        "field": "type",
        "equals": "Microsoft.DocumentDB/databaseAccounts"
      },
      {
        "field": "Microsoft.DocumentDB/databaseAccounts/enableMultipleWriteLocations",
        "notEquals": true
      }
    ]
  },
  "then": {
    "effect": "Audit"
  }
}
```

Custom policy definitions can be used to create policy assignments just like the built-in policy definitions are used.

Policy compliance

After the policy assignments are created, Azure Policy evaluates the resources in the assignment's scope. Each resource's *compliance* with the policy is assessed. The *effect* specified in the policy is then applied to non-compliant resources.

You can review the compliance results and remediation details in the Azure portal or via the Azure CLI or the Azure Monitor logs.

The following screenshot shows two example policy assignments.

One assignment is based on a built-in policy definition, which checks that the Azure Cosmos DB resources are deployed only to the allowed Azure regions. Resource compliance shows policy evaluation outcome (compliant or non-compliant) for in-scope resources.

The other assignment is based on a custom policy definition. This assignment checks that Cosmos DB accounts are configured for multiple write locations.

After the policy assignments are deployed, the compliance dashboard shows evaluation results. Note that this can take up to 30 minutes after deploying a policy assignment. Additionally, policy evaluation scans can be started on-demand immediately after creating policy assignments.

The screenshot shows the following compliance evaluation results for in-scope Azure Cosmos DB accounts:

- Zero of two accounts are compliant with a policy that Virtual Network (VNet) filtering must be configured.
- Zero of two accounts are compliant with a policy that requires the account to be configured for multiple write locations
- Zero of two accounts are compliant with a policy that resources were deployed to allowed Azure regions.

Name	Scope	Compliance state	Resource compliance
Cosmos DB | VNet filtering	[redacted]	Non-compliant	0% (0 out of 2)
Cosmos DB | Multiple Write Locations | Deny	[redacted]	Non-compliant	0% (0 out of 2)
Built-in: Azure Cosmos DB allowed locations	[redacted]	Non-compliant	0% (0 out of 2)

To remediate the non-compliant resources, see how to remediate resources with Azure Policy.

Next steps

- Review sample custom policy definitions for Azure Cosmos DB, including for the multiple write location and VNet filtering policies shown above.
- Create a policy assignment in the Azure portal
- Review Azure Policy built-in policy definitions for Azure Cosmos DB

Azure Policy Regulatory Compliance controls for Azure Cosmos DB

12/4/2020 • 3 minutes to read • Edit Online

APPLIES TO: SQL API Cassandra API Gremlin API Table API Azure Cosmos DB API for MongoDB

Regulatory Compliance in Azure Policy provides Microsoft created and managed initiative definitions, known as *built-ins*, for the **compliance domains** and **security controls** related to different compliance standards. This page lists the **compliance domains** and **security controls** for Azure Cosmos DB. You can assign the built-ins for a **security control** individually to help make your Azure resources compliant with the specific standard.

The title of each built-in policy definition links to the policy definition in the Azure portal. Use the link in the **Policy Version** column to view the source on the Azure Policy GitHub repo.

> **IMPORTANT**
>
> Each control below is associated with one or more Azure Policy definitions. These policies may help you assess compliance with the control; however, there often is not a one-to-one or complete match between a control and one or more policies. As such, **Compliant** in Azure Policy refers only to the policies themselves; this doesn't ensure you're fully compliant with all requirements of a control. In addition, the compliance standard includes controls that aren't addressed by any Azure Policy definitions at this time. Therefore, compliance in Azure Policy is only a partial view of your overall compliance status. The associations between controls and Azure Policy Regulatory Compliance definitions for these compliance standards may change over time.

Azure Security Benchmark

The Azure Security Benchmark provides recommendations on how you can secure your cloud solutions on Azure. To see how this service completely maps to the Azure Security Benchmark, see the Azure Security Benchmark mapping files.

To review how the available Azure Policy built-ins for all Azure services map to this compliance standard, see Azure Policy Regulatory Compliance - Azure Security Benchmark.

DOMAIN	CONTROL ID	CONTROL TITLE	POLICY (AZURE PORTAL)	POLICY VERSION (GITHUB)
Network Security	1.1	Protect resources using Network Security Groups or Azure Firewall on your Virtual Network	Cosmos DB should use a virtual network service endpoint	1.0.0

HIPAA HITRUST 9.2

To review how the available Azure Policy built-ins for all Azure services map to this compliance standard, see Azure Policy Regulatory Compliance - HIPAA HITRUST 9.2. For more information about this compliance standard, see HIPAA HITRUST 9.2.

DOMAIN	CONTROL ID	CONTROL TITLE	POLICY (AZURE PORTAL)	POLICY VERSION (GITHUB)
Segregation in Networks	0805.01m1Organizational.12 - 01.m	The organization's security gateways (e.g. firewalls) enforce security policies and are configured to filter traffic between domains, block unauthorized access, and are used to maintain segregation between internal wired, internal wireless, and external network segments (e.g., the Internet) including DMZs and enforce access control policies for each of the domains.	Cosmos DB should use a virtual network service endpoint	1.0.0
Segregation in Networks	0806.01m2Organizational.12356 - 01.m	The organizations network is logically and physically segmented with a defined security perimeter and a graduated set of controls, including subnetworks for publicly accessible system components that are logically separated from the internal network, based on organizational requirements; and traffic is controlled based on functionality required and classification of the data/systems based on a risk assessment and their respective security requirements.	Cosmos DB should use a virtual network service endpoint	1.0.0
Segregation in Networks	0894.01m2Organizational.7 - 01.m	Networks are segregated from production-level networks when migrating physical servers, applications or data to virtualized servers.	Cosmos DB should use a virtual network service endpoint	1.0.0

DOMAIN	CONTROL ID	CONTROL TITLE	POLICY	POLICY VERSION
Network Controls	0864.09m2Organizational.12 - 09.m	Usage restrictions and implementation guidance are formally defined for VoIP, including the authorization and monitoring of the service.	Cosmos DB should use a virtual network service endpoint	1.0.0

Next steps

- Learn more about Azure Policy Regulatory Compliance.
- See the built-ins on the Azure Policy GitHub repo.

Azure Security Baseline for Cosmos DB

12/4/2020 • 33 minutes to read • Edit Online

APPLIES TO: SQL API Cassandra API Gremlin API Table API Azure Cosmos DB API for MongoDB

The Azure Security Baseline for Cosmos DB contains recommendations that will help you improve the security posture of your deployment.

The baseline for this service is drawn from the Azure Security Benchmark version 1.0, which provides recommendations on how you can secure your cloud solutions on Azure with our best practices guidance.

For more information, see Azure Security Baselines overview.

Network Security

For more information, see Security Control: Network Security.

1.1: Protect resources using Network Security Groups or Azure Firewall on your Virtual Network

Guidance: By using Azure Private Link, you can connect to an Azure Cosmos account via a Private Endpoint. Traffic between your virtual network and the service traverses over the Microsoft backbone network, eliminating exposure from the public Internet. You can also reduce the risk of data exfiltration by configuring a strict set of outbound rules on a network security group (NSG) and associating that NSG with your subnet.

You can also use Service Endpoints to secure your Azure Cosmos account. By enabling a Service Endpoint, you can configure Azure Cosmos accounts to allow access from only a specific subnet of an Azure virtual network. Once the Azure Cosmos DB Service Endpoint is enabled, you can limit access to an Azure Cosmos account with connections from a subnet in a virtual network.

You can also secure the data stored in your Azure Cosmos account by using IP firewalls. Azure Cosmos DB supports IP-based access controls for inbound firewall support. You can set an IP firewall on the Azure Cosmos account by using the Azure portal, Azure Resource Manager templates, or through the Azure CLI or Azure PowerShell.

Azure Private Link Overview: https://docs.microsoft.com/azure/private-link/private-link-overview

How to configure a Private Endpoint for Azure Cosmos DB: https://docs.microsoft.com/azure/cosmos-db/how-to-configure-private-endpoints

How to create a Network Security Group with a Security Config: https://docs.microsoft.com/azure/virtual-network/tutorial-filter-network-traffic

How to configure IP firewall in Cosmos DB: https://docs.microsoft.com/azure/cosmos-db/how-to-configure-firewall

Azure Security Center monitoring: Yes

Responsibility: Customer

1.2: Monitor and log the configuration and traffic of Vnets, Subnets, and NICs

Guidance: Use Azure Security Center and follow network protection recommendations to help secure network resources related to your Azure Cosmos account.

When virtual machines are deployed in the same virtual network as your Azure Cosmos account, you can use a

network security group (NSG) to reduce the risk of data exfiltration. Enable NSG flow logs and send logs into an Azure Storage Account for traffic audits. You may also send NSG flow logs to a Log Analytics workspace and use Traffic Analytics to provide insights into traffic flow in your Azure cloud. Some advantages of Traffic Analytics are the ability to visualize network activity and identify hot spots, identify security threats, understand traffic flow patterns, and pinpoint network misconfigurations.

Understand Network Security provided by Azure Security Center: https://docs.microsoft.com/azure/security-center/security-center-network-recommendations

How to Enable NSG Flow Logs: https://docs.microsoft.com/azure/network-watcher/network-watcher-nsg-flow-logging-portal

How to Enable and use Traffic Analytics: https://docs.microsoft.com/azure/network-watcher/traffic-analytics

Azure Security Center monitoring: Yes

Responsibility: Customer

1.3: Protect critical web applications

Guidance: Use the Cross-Origin Resource Sharing (CORS) feature to enable a web application running under one domain to access resources in another domain. Web browsers implement a security restriction known as same-origin policy that prevents a web page from calling APIs in a different domain. However, CORS provides a secure way to allow the origin domain to call APIs in another domain. After you enable the CORS support for your Azure Cosmos account, only authenticated requests are evaluated to determine whether they are allowed according to the rules you have specified.

Configure Cross-Origin Resource Sharing: https://docs.microsoft.com/azure/cosmos-db/how-to-configure-cross-origin-resource-sharing

Azure Security Center monitoring: Yes

Responsibility: Customer

1.4: Deny communications with known malicious IP addresses

Guidance: Use Advanced Threat Protection (ATP) for Azure Cosmos DB . ATP for Azure Cosmos DB provides an additional layer of security intelligence that detects unusual and potentially harmful attempts to access or exploit Azure Cosmos accounts. This layer of protection allows you to address threats and integrate them with central security monitoring systems.

Enable DDoS Protection Standard on the Virtual Networks associated with your Azure Cosmos DB instances to guard against DDoS attacks. Use Azure Security Center Integrated Threat Intelligence to deny communications with known malicious or unused Internet IP addresses.

How to configure Azure Cosmos DB Advanced Threat Protection: https://docs.microsoft.com/azure/cosmos-db/cosmos-db-advanced-threat-protection

How to configure DDoS protection: https://docs.microsoft.com/azure/virtual-network/manage-ddos-protection

Understand Azure Security Center Integrated Threat Intelligence: https://docs.microsoft.com/azure/security-center/security-center-alerts-service-layer

Azure Security Center monitoring: Yes

Responsibility: Customer

1.5: Record network packets and flow logs

Guidance: Enable network security group (NSG) flow logs and send logs into a storage account for traffic audit. You can send NSG flow logs to a Log Analytics workspace and use Traffic Analytics to provide insights into traffic flow in your Azure cloud. Some advantages of Traffic Analytics are the ability to visualize network activity and

identify hot spots, identify security threats, understand traffic flow patterns, and pinpoint network misconfigurations.

How to Enable NSG Flow Logs: https://docs.microsoft.com/azure/network-watcher/network-watcher-nsg-flow-logging-portal

How to Enable and use Traffic Analytics: https://docs.microsoft.com/azure/network-watcher/traffic-analytics

Azure Security Center monitoring: Yes

Responsibility: Customer

1.6: Deploy network based intrusion detection/intrusion prevention systems (IDS/IPS)

Guidance: Use Advanced Threat Protection (ATP) for Azure Cosmos DB. ATP for Azure Cosmos DB provides an additional layer of security intelligence that detects unusual and potentially harmful attempts to access or exploit Azure Cosmos accounts. This layer of protection allows you to address threats and integrate them with central security monitoring systems.

How to configure Cosmos DB Advanced Threat Protection: https://docs.microsoft.com/azure/cosmos-db/cosmos-db-advanced-threat-protection

Azure Security Center monitoring: Yes

Responsibility: Customer

1.7: Manage traffic to web applications

Guidance: Not applicable; recommendation is intended for web applications running on Azure App Service or compute resources.

Azure Security Center monitoring: Not applicable

Responsibility: Not applicable

1.8: Minimize complexity and administrative overhead of network security rules

Guidance: For resources that need access to your Azure Cosmos account, use Virtual Network service tags to define network access controls on network security groups or Azure Firewall. You can use service tags in place of specific IP addresses when creating security rules. By specifying the service tag name (e.g., AzureCosmosDB) in the appropriate source or destination field of a rule, you can allow or deny the traffic for the corresponding service. Microsoft manages the address prefixes encompassed by the service tag and automatically updates the service tag as addresses change.

For more information about using service tags: https://docs.microsoft.com/azure/virtual-network/service-tags-overview

Azure Security Center monitoring: Not applicable

Responsibility: Customer

1.9: Maintain standard security configurations for network devices

Guidance: Define and implement standard security configurations for network resources with Azure Policy. Use Azure Policy aliases in the "Microsoft.DocumentDB" and "Microsoft.Network" namespaces to create custom policies to audit or enforce the network configuration of your Azure Cosmos DB instances. You may also make use of built-in policy definitions for Azure Cosmos DB, such as:

- Deploy Advanced Threat Protection for Cosmos DB Accounts
- Cosmos DB should use a virtual network service endpoint

You may also use Azure Blueprints to simplify large-scale Azure deployments by packaging key environment

artifacts, such as Azure Resource Manager templates, Azure role-based access control (Azure RBAC), and policies in a single blueprint definition. You can easily apply the blueprint to new subscriptions, environments, and fine-tune control and management through versioning.

How to configure and manage Azure Policy: https://docs.microsoft.com/azure/governance/policy/tutorials/create-and-manage

How to create an Azure Blueprint: https://docs.microsoft.com/azure/governance/blueprints/create-blueprint-portal

Azure Security Center monitoring: Not applicable

Responsibility: Customer

1.10: Document traffic configuration rules

Guidance: Use tags for network resources associated with your Azure Cosmos DB deployment in order to logically organize them into a taxonomy.

How to create and use tags: https://docs.microsoft.com/azure/azure-resource-manager/resource-group-using-tags

Azure Security Center monitoring: Not applicable

Responsibility: Customer

1.11: Use automated tools to monitor network resource configurations and detect changes

Guidance: Use Azure Activity Log to monitor network resource configurations and detect changes for network resources related to your Azure Cosmos DB instances. Create alerts within Azure Monitor that will trigger when changes to critical network resources take place.

How to view and retrieve Azure Activity Log events: https://docs.microsoft.com/azure/azure-monitor/platform/activity-log-view

How to create alerts in Azure Monitor: https://docs.microsoft.com/azure/azure-monitor/platform/alerts-activity-log

Azure Security Center monitoring: Not applicable

Responsibility: Customer

Logging and Monitoring

For more information, see Security Control: Logging and Monitoring.

2.1: Use approved time synchronization sources

Guidance: Microsoft maintains the time source used for Azure resources such as Azure Cosmos DB for timestamps in the logs.

Azure Security Center monitoring: Not applicable

Responsibility: Microsoft

2.2: Configure central security log management

Guidance: Ingest logs via Azure Monitor to aggregate security data generated by Azure Cosmos DB. Within the Azure Monitor, use Log Analytics workspace(s) to query and perform analytics, and use storage accounts for long-term/archival storage. Alternatively, you may enable, and on-board data to Azure Sentinel or a third-party Security Incident and Event Management (SIEM).

How to enable diagnostic logs for Azure Cosmos DB: https://docs.microsoft.com/azure/cosmos-db/logging

How to onboard Azure Sentinel: https://docs.microsoft.com/azure/sentinel/quickstart-onboard

Azure Security Center monitoring: Currently not available

Responsibility: Customer

2.3: Enable audit logging for Azure resources

Guidance: Enable diagnostic settings for Azure Cosmos DB and send the logs to a Log Analytics workspace or storage account. Diagnostic settings in Azure Cosmos DB are used to collect resource logs. These logs are captured per request and they are also referred to as "data plane logs". Some examples of the data plane operations include delete, insert, and read. You may also enable Azure Activity Log Diagnostic Settings and send them to the same Log Analytics Workspace.

How to enable Diagnostic Settings for Azure Cosmos DB: https://docs.microsoft.com/azure/cosmos-db/logging

How to enable Diagnostic Settings for Azure Activity Log: https://docs.microsoft.com/azure/azure-monitor/platform/diagnostic-settings-legacy

Azure Security Center monitoring: Currently not available

Responsibility: Customer

2.4: Collect security logs from operating systems

Guidance: Not applicable; this recommendation is intended for compute resources.

Azure Security Center monitoring: Not applicable

Responsibility: Not applicable

2.5: Configure security log storage retention

Guidance: In Azure Monitor, set the log retention period for Log Analytics workspaces associated with your Azure Cosmos DB instances according to your organization's compliance regulations.

How to set log retention parameters: https://docs.microsoft.com/azure/azure-monitor/platform/manage-cost-storage#change-the-data-retention-period

Azure Security Center monitoring: Not applicable

Responsibility: Customer

2.6: Monitor and review Logs

Guidance: You can perform queries in Log Analytics a workspace to search terms, identify trends, analyze patterns, and provide many other insights based on the Azure Cosmos DB logs that you gathered.

How to perform queries for Azure Cosmos DB in Log Analytics Workspaces: https://docs.microsoft.com/azure/cosmos-db/monitor-cosmos-db

Azure Security Center monitoring: Not applicable

Responsibility: Customer

2.7: Enable alerts for anomalous activity

Guidance: In Azure Security Center, enable Advanced Threat Protection for Azure Cosmos DB to monitor anomalous activities in security logs and events. Enable diagnostic settings in Azure Cosmos DB and send logs to a Log Analytics workspace.

You can also onboard your Log Analytics workspace to Azure Sentinel as it provides a security orchestration automated response (SOAR) solution. This allows for playbooks (automated solutions) to be created and used to remediate security issues. Additionally, you can create custom log alerts in your Log Analytics workspace using Azure Monitor.

List of threat protection alerts for Azure Cosmos DB: https://docs.microsoft.com/azure/security-center/alerts-

reference#alerts-azurecosmos

How to onboard Azure Sentinel: https://docs.microsoft.com/azure/sentinel/quickstart-onboard

Create, view, and manage log alerts using Azure Monitor: https://docs.microsoft.com/azure/azure-monitor/platform/alerts-log

Azure Security Center monitoring: Yes

Responsibility: Customer

2.8: Centralize anti-malware logging

Guidance: Not applicable; Azure Cosmos DB does not process or produce anti-malware related logs.

Azure Security Center monitoring: Not applicable

Responsibility: Not applicable

2.9: Enable DNS query logging

Guidance: Not applicable; Azure Cosmos DB does not process or produce DNS-related logs.

Azure Security Center monitoring: Not applicable

Responsibility: Not applicable

2.10: Enable command-line audit logging

Guidance: Not applicable; this recommendation is intended for compute resources.

Azure Security Center monitoring: Not applicable

Responsibility: Not applicable

Identity and Access Control

For more information, see Security Control: Identity and Access Control.

3.1: Maintain an inventory of administrative accounts

Guidance: You can use the Identity and Access control (IAM) pane in the Azure portal to configure Azure role-based access control (Azure RBAC) and maintain inventory on Azure Cosmos DB resources. The roles are applied to users, groups, service principals, and managed identities in Active Directory. You can use built-in roles or custom roles for individuals and groups.

Azure Cosmos DB provides built-in Azure RBAC for common management scenarios in Azure Cosmos DB. An individual who has a profile in Azure Active Directory (AD) can assign these Azure roles to users, groups, service principals, or managed identities to grant or deny access to resources and operations on Azure Cosmos DB resources.

You can also use the Azure AD PowerShell module to perform adhoc queries to discover accounts that are members of administrative groups.

Additionally, some actions in Azure Cosmos DB can be controlled with Azure Active Directory and account-specific primary keys. Use the 'disableKeyBasedMetadataWriteAccess' account setting to control key access.

Understand Azure role-based access control in Azure Cosmos DB: https://docs.microsoft.com/azure/cosmos-db/role-based-access-control

Build your own custom roles using Azure Cosmos DB Actions (Microsoft.DocumentDB namespace): https://docs.microsoft.com/azure/role-based-access-control/resource-provider-operations#microsoftdocumentdb

Create a new role in Azure Active Directory: https://docs.microsoft.com/azure/role-based-access-control/custom-

roles

How to get a directory role in Azure Active Directory with PowerShell: https://docs.microsoft.com/powershell/module/azuread/get-azureaddirectoryrole?view=azureadps-2.0&preserve-view=true

How to get members of a directory role in Azure Active Directory with PowerShell: https://docs.microsoft.com/powershell/module/azuread/get-azureaddirectoryrolemember?view=azureadps-2.0&preserve-view=true

Restrict user access to data operations only: https://docs.microsoft.com/azure/cosmos-db/how-to-restrict-user-data

Azure Security Center monitoring: Yes

Responsibility: Customer

3.2: Change default passwords where applicable

Guidance: The concept of default or blank passwords does not exist in relation to Azure AD or Azure Cosmos DB. Instead, Azure Cosmos DB uses two types of keys to authenticate users and provide access to its data and resources; primary keys and resource tokens. The keys can be regenerated at any time.

Understanding secure access to data in Azure Cosmos DB: https://docs.microsoft.com/azure/cosmos-db/secure-access-to-data

How to regenerate Azure Cosmos DB Keys: https://docs.microsoft.com/azure/cosmos-db/manage-with-powershell#regenerate-keys

How to programmatically access keys using Azure Active Directory: https://docs.microsoft.com/azure/cosmos-db/certificate-based-authentication

Azure Security Center monitoring: Not applicable

Responsibility: Shared

3.3: Use dedicated administrative accounts

Guidance: Not applicable; Azure Cosmos DB does not support administrator accounts. All access is integrated with Azure Active Directory and Azure role-based access control (Azure RBAC).

Azure Security Center monitoring: Not applicable

Responsibility: Customer

3.4: Use single sign-on (SSO) with Azure Active Directory

Guidance: Azure Cosmos DB uses two types of keys to authorize users and does not support Single Sign-On (SSO) at the data plane level. Access to the control plane for Cosmos DB is available via REST API and supports SSO. To authenticate, set the Authorization header for your requests to a JSON Web Token that you obtain from Azure Active Directory.

Understand Azure Database for Cosmos DB REST API: https://docs.microsoft.com/rest/api/cosmos-db/

Understand SSO with Azure Active Directory: https://docs.microsoft.com/azure/active-directory/manage-apps/what-is-single-sign-on

Azure Security Center monitoring: Not applicable

Responsibility: Customer

3.5: Use multi-factor authentication for all Azure Active Directory based access

Guidance: Enable Azure Active Directory Multi-Factor Authentication and follow Azure Security Center Identity

and Access Management recommendations.

How to enable MFA in Azure: https://docs.microsoft.com/azure/active-directory/authentication/howto-mfa-getstarted

How to monitor identity and access within Azure Security Center: https://docs.microsoft.com/azure/security-center/security-center-identity-access

Azure Security Center monitoring: Yes

Responsibility: Customer

3.6: Use dedicated machines (Privileged Access Workstations) for all administrative tasks

Guidance: Use Privileged Access Workstations (PAW) with Multi-Factor Authentication configured to log into and configure Azure resources.

Learn about Privileged Access Workstations: https://docs.microsoft.com/windows-server/identity/securing-privileged-access/privileged-access-workstations

How to enable MFA in Azure: https://docs.microsoft.com/azure/active-directory/authentication/howto-mfa-getstarted

Azure Security Center monitoring: Not applicable

Responsibility: Customer

3.7: Log and alert on suspicious activity from administrative accounts

Guidance: Use Advanced Threat Protection (ATP) for Azure Cosmos DB. ATP for Azure Cosmos DB provides an additional layer of security intelligence that detects unusual and potentially harmful attempts to access or exploit Azure Cosmos accounts. This layer of protection allows you to address threats and integrate them with central security monitoring systems.

In addition, you may use Azure Active Directory (AD) Privileged Identity Management (PIM) for generation of logs and alerts when suspicious or unsafe activity occurs in the environment.

Use Azure AD Risk Detections to view alerts and reports on risky user behavior.

How to deploy Privileged Identity Management (PIM): https://docs.microsoft.com/azure/active-directory/privileged-identity-management/pim-deployment-plan

Understand Azure AD risk detections: https://docs.microsoft.com/azure/active-directory/reports-monitoring/concept-risk-events

Azure Security Center monitoring: Yes

Responsibility: Customer

3.8: Manage Azure resources from only approved locations

Guidance: Configure the location condition of a Conditional Access policy and manage your named locations. With named locations, you can create logical groupings of IP address ranges or countries and regions. You can restrict access to sensitive resources, such as your Azure Cosmos DB instances, to your configured named locations.

How to configure Named Locations in Azure: https://docs.microsoft.com/azure/active-directory/reports-monitoring/quickstart-configure-named-locations

Azure Security Center monitoring: Not applicable

Responsibility: Customer

3.9: Use Azure Active Directory

Guidance: Use Azure Active Directory (AD) as the central authentication and authorization system. Azure AD protects data by using strong encryption for data at rest and in transit. Azure AD also salts, hashes, and securely stores user credentials.

How to create and configure an Azure Active Directory instance: https://docs.microsoft.com/azure/active-directory-domain-services/tutorial-create-instance

How to configure and manage Azure Active Directory authentication with Azure SQL: https://docs.microsoft.com/azure/sql-database/sql-database-aad-authentication-configure

Azure Security Center monitoring: Not applicable

Responsibility: Customer

3.10: Regularly review and reconcile user access

Guidance: Azure Active Directory provides logs to help discover stale accounts. In addition, you can use Azure Identity Access Reviews to efficiently manage group memberships, access to enterprise applications, and role assignments. User's access can be reviewed on a regular basis to make sure only the right Users have continued access.

How to use Azure Identity Access Reviews: https://docs.microsoft.com/azure/active-directory/governance/access-reviews-overview

Azure Security Center monitoring: Yes

Responsibility: Customer

3.11: Monitor attempts to access deactivated accounts

Guidance: You can create diagnostic settings for Azure Active Directory user accounts, sending the audit logs and sign-in logs to a Log Analytics workspace where you can configure desired alerts.

How to integrate Azure Activity Logs into Azure Monitor: https://docs.microsoft.com/azure/active-directory/reports-monitoring/howto-integrate-activity-logs-with-log-analytics

Azure Security Center monitoring: Currently not available

Responsibility: Customer

3.12: Alert on account login behavior deviation

Guidance: Use Advanced Threat Protection (ATP) for Azure Cosmos DB. ATP for Azure Cosmos DB provides an additional layer of security intelligence that detects unusual and potentially harmful attempts to access or exploit Azure Cosmos accounts. This layer of protection allows you to address threats and integrate them with central security monitoring systems.

You can also use Azure AD Identity Protection and risk detections feature to configure automated responses to detected suspicious actions related to user identities. Additionally, you can ingest logs into Azure Sentinel for further investigation.

How to view Azure Active Directory risky sign-ins: https://docs.microsoft.com/azure/active-directory/reports-monitoring/concept-risky-sign-ins

How to configure and enable Identity Protection risk policies: https://docs.microsoft.com/azure/active-directory/identity-protection/howto-identity-protection-configure-risk-policies

How to onboard Azure Sentinel: https://docs.microsoft.com/azure/sentinel/quickstart-onboard

Azure Security Center monitoring: Yes

Responsibility: Customer

3.13: Provide Microsoft with access to relevant customer data during support scenarios

Guidance: Customer Lockbox is not applicable for Cosmos DB. Microsoft employees cannot access data resources even with customer permissions.

List of Customer Lockbox supported services: https://docs.microsoft.com/azure/security/fundamentals/customer-lockbox-overview#supported-services-and-scenarios-in-general-availability

Azure Security Center monitoring: Not applicable

Responsibility: Not applicable

Data Protection

For more information, see Security Control: Data Protection.

4.1: Maintain an inventory of sensitive Information

Guidance: Use tags to assist in tracking Azure Cosmos DB instances that store or process sensitive information.

How to create and use tags: https://docs.microsoft.com/azure/azure-resource-manager/resource-group-using-tags

Azure Security Center monitoring: Not applicable

Responsibility: Customer

4.2: Isolate systems storing or processing sensitive information

Guidance: Implement separate subscriptions and/or management groups for development, test, and production. Azure Cosmos DB instances are separated by virtual network/subnet, tagged appropriately, and secured within a network security group (NSG) or Azure Firewall. Azure Cosmos DB instances storing sensitive data should be isolated. By using Azure Private Link, you can connect to an Azure Cosmos DB instance account via a private endpoint. The private endpoint is a set of private IP addresses in a subnet within your virtual network. You can then limit access to the selected private IP addresses.

How to create additional Azure subscriptions: https://docs.microsoft.com/azure/billing/billing-create-subscription

How to create management groups: https://docs.microsoft.com/azure/governance/management-groups/create

How to create and use tags: https://docs.microsoft.com/azure/azure-resource-manager/resource-group-using-tags

How to configure a Private Endpoint for Azure Cosmos DB: https://docs.microsoft.com/azure/cosmos-db/how-to-configure-private-endpoints

How to create a Network Security Group with a Security Config: https://docs.microsoft.com/azure/virtual-network/tutorial-filter-network-traffic

Azure Security Center monitoring: Not applicable

Responsibility: Customer

4.3: Monitor and block unauthorized transfer of sensitive information

Guidance: You can deploy Advanced Threat Protection for Azure Cosmos DB, which will detect anomalous activities indicating unusual and potentially harmful attempts to access or exploit databases. It can currently trigger the following alerts:

- Access from unusual locations
- Unusual data extraction

Additionally, when using virtual machines to access your Azure Cosmos DB instances, make use of Private Link, Firewall, network security groups, and service tags to mitigate the possibility of data exfiltration. Microsoft manages the underlying infrastructure for Azure Cosmos DB and has implemented strict controls to prevent the

loss or exposure of customer data.

How to configure Cosmos DB Advanced Threat Protection: https://docs.microsoft.com/azure/cosmos-db/cosmos-db-advanced-threat-protection

Understand customer data protection in Azure: https://docs.microsoft.com/azure/security/fundamentals/protection-customer-data

Azure Security Center monitoring: Yes

Responsibility: Shared

4.4: Encrypt all sensitive information in transit

Guidance: All connections to Azure Cosmos DB support HTTPS. Azure Cosmos DB also supports TLS1.2. It is possible to enforce a minimum TLS version server-side. To do so, please contact azurecosmosdbtls@service.microsoft.com.

Overview of Cosmos DB Security: https://docs.microsoft.com/azure/cosmos-db/database-security

Azure Security Center monitoring: Not applicable

Responsibility: Microsoft

4.5: Use an active discovery tool to identify sensitive data

Guidance: Automatic data identification, classification, and loss prevention features are not yet available for Azure Cosmos DB. However, you can use the Azure Cognitive Search integration for classification and data analysis. You can also implement a third-party solution if required for compliance purposes.

For the underlying platform which is managed by Microsoft, Microsoft treats all customer content as sensitive and goes to great lengths to guard against customer data loss and exposure. To ensure customer data within Azure remains secure, Microsoft has implemented and maintains a suite of robust data protection controls and capabilities.

Index Azure Cosmos DB data with Azure Cognitive Search: https://docs.microsoft.com/azure/search/search-howto-index-cosmosdb?toc=/azure/cosmos-db/toc.json&bc=/azure/cosmos-db/breadcrumb/toc.json

Understand customer data protection in Azure: https://docs.microsoft.com/azure/security/fundamentals/protection-customer-data

Azure Security Center monitoring: Currently not available

Responsibility: Shared

4.6: Use Azure RBAC to control access to resources

Guidance: Azure Cosmos DB provides built-in Azure role-based access control (Azure RBAC) for common management scenarios in Azure Cosmos DB. An individual who has a profile in Azure Active Directory can assign these Azure roles to users, groups, service principals, or managed identities to grant or deny access to resources and operations on Azure Cosmos DB resources. Role assignments are scoped to control-plane access only, which includes access to Azure Cosmos accounts, databases, containers, and offers (throughput).

How to implement Azure RBAC in Azure Cosmos DB: https://docs.microsoft.com/azure/cosmos-db/role-based-access-control

Azure Security Center monitoring: Not applicable

Responsibility: Customer

4.7: Use host-based data loss prevention to enforce access control

Guidance: Not applicable; this guideline is intended for compute resources.

Microsoft manages the underlying infrastructure for Cosmos DB and has implemented strict controls to prevent the loss or exposure of customer data.

Understand customer data protection in Azure: https://docs.microsoft.com/azure/security/fundamentals/protection-customer-data

Azure Security Center monitoring: Not applicable

Responsibility: Not applicable

4.8: Encrypt sensitive information at rest

Guidance: All user data stored in Cosmos DB is encrypted at rest by default. There are no controls to turn it off. Azure Cosmos DB uses AES-256 encryption on all regions where the account is running.

By default, Microsoft manages the keys that are used to encrypt the data in your Azure Cosmos account. You can optionally choose to add a second layer of encryption with your own keys.

Understanding encryption at rest with Azure Cosmos DB: https://docs.microsoft.com/azure/cosmos-db/database-encryption-at-rest

Understanding key management for encryption at rest with Azure Cosmos DB: https://docs.microsoft.com/azure/cosmos-db/cosmos-db-security-controls

How to configure customer-managed keys for your Azure Cosmos DB account: https://docs.microsoft.com/azure/cosmos-db/how-to-setup-cmk

Azure Security Center monitoring: Not applicable

Responsibility: Shared

4.9: Log and alert on changes to critical Azure resources

Guidance: Use Azure Monitor with the Azure Activity Log to create alerts for when changes take place to production instances of Azure Cosmos DB.

How to create alerts for Azure Activity Log events: https://docs.microsoft.com/azure/azure-monitor/platform/alerts-activity-log

How to create alerts for Azure Activity Log events: https://docs.microsoft.com/azure/azure-monitor/platform/alerts-activity-log

Azure Security Center monitoring: Not applicable

Responsibility: Customer

Vulnerability Management

For more information, see Security Control: Vulnerability Management.

5.1: Run automated vulnerability scanning tools

Guidance: Follow recommendations from Azure Security Center for your Azure Cosmos DB instances.

Microsoft performs system patching and vulnerability management on the underlying hosts that support your Azure Cosmos DB instances. To ensure customer data within Azure remains secure, Microsoft has implemented and maintains a suite of robust data protection controls and capabilities.

Supported features available in Azure Security Center: https://docs.microsoft.com/azure/security-center/security-center-services?tabs=features-windows

Azure Security Center monitoring: Yes

Responsibility: Shared

5.2: Deploy automated operating system patch management solution

Guidance: Not applicable; this guideline is intended for compute resources.

Azure Security Center monitoring: Not applicable

Responsibility: Not applicable

5.3: Deploy automated third-party software patch management solution

Guidance: Not applicable; this guideline is intended for compute resources.

Azure Security Center monitoring: Not applicable

Responsibility: Not applicable

5.4: Compare back-to-back vulnerability scans

Guidance: Not applicable; this guideline is intended for compute resources.

Azure Security Center monitoring: Not applicable

Responsibility: Not applicable

5.5: Use a risk-rating process to prioritize the remediation of discovered vulnerabilities

Guidance: Not applicable; this guideline is intended for compute resources.

Azure Security Center monitoring: Not applicable

Responsibility: Not applicable

Inventory and Asset Management

For more information, see Security Control: Inventory and Asset Management.

6.1: Use Azure Asset Discovery

Guidance: Use the Azure portal or Azure Resource Graph to discover all resources (not limited to Azure Cosmos DB, but also including resources such as compute, other storage, network, ports, and protocols etc.) within your subscription(s). Ensure you have appropriate permissions in your tenant and are able to enumerate all Azure subscriptions as well as resources within your subscriptions.

Although classic Azure resources may be discovered via Resource Graph, it is highly recommended to create and use Azure Resource Manager resources going forward.

How to create queries with Azure Resource Graph: https://docs.microsoft.com/azure/governance/resource-graph/first-query-portal

How to view your Azure Subscriptions: https://docs.microsoft.com/powershell/module/az.accounts/get-azsubscription?view=azps-3.0.0&preserve-view=true

Understanding Azure role-based access control: https://docs.microsoft.com/azure/role-based-access-control/overview

Azure Security Center monitoring: Not applicable

Responsibility: Customer

6.2: Maintain asset metadata

Guidance: Apply tags to your Azure Cosmos DB instances and related resources with metadata to logically organize them into a taxonomy.

How to create and use tags: https://docs.microsoft.com/azure/azure-resource-manager/resource-group-using-tags

Which Azure Cosmos DB resources support tags: https://docs.microsoft.com/azure/azure-resource-manager/management/tag-support#microsoftdocumentdb

Azure Security Center monitoring: Not applicable

Responsibility: Customer

6.3: Delete unauthorized Azure resources

Guidance: Use tagging, management groups, and separate subscriptions, where appropriate, to organize and track assets, including but not limited to Azure Cosmos DB resources. Reconcile inventory on a regular basis and ensure unauthorized resources are deleted from the subscription in a timely manner.

How to create additional Azure subscriptions: https://docs.microsoft.com/azure/billing/billing-create-subscription

How to create Management Groups: https://docs.microsoft.com/azure/governance/management-groups/create

How to create and use Tags: https://docs.microsoft.com/azure/azure-resource-manager/resource-group-using-tags

Azure Security Center monitoring: Not applicable

Responsibility: Customer

6.4: Maintain an inventory of approved Azure resources and software titles

Guidance: Not applicable; this guideline is intended for compute resources and Azure as a whole.

Azure Security Center monitoring: Not applicable

Responsibility: Not applicable

6.5: Monitor for unapproved Azure resources

Guidance: Use Azure Policy to put restrictions on the type of resources that can be created in customer subscription(s) using the following built-in policy definitions:

- Not allowed resource types
- Allowed resource types

In addition, use the Azure Resource Graph to query/discover resources within the subscription(s).

How to configure and manage Azure Policy: https://docs.microsoft.com/azure/governance/policy/tutorials/create-and-manage

How to create queries with Azure Graph: https://docs.microsoft.com/azure/governance/resource-graph/first-query-portal

Azure Security Center monitoring: Not applicable

Responsibility: Customer

6.6: Monitor for unapproved software applications within compute resources

Guidance: Not applicable; this baseline is intended for compute resources.

Azure Security Center monitoring: Not applicable

Responsibility: Not applicable

6.7: Remove unapproved Azure resources and software applications

Guidance: Not applicable; this guideline is intended for compute resources and Azure as a whole.

Azure Security Center monitoring: Not applicable

Responsibility: Not applicable

6.8: Use only approved applications

Guidance: Not applicable; this guideline is intended for compute resources.

Azure Security Center monitoring: Not applicable

Responsibility: Not applicable

6.9: Use only approved Azure services

Guidance: Use Azure Policy to put restrictions on the type of resources that can be created in customer subscription(s) using the following built-in policy definitions:

- Not allowed resource types
- Allowed resource types

How to configure and manage Azure Policy: https://docs.microsoft.com/azure/governance/policy/tutorials/create-and-manage

How to deny a specific resource type with Azure Policy: https://docs.microsoft.com/azure/governance/policy/samples/not-allowed-resource-types

Azure Security Center monitoring: Not applicable

Responsibility: Customer

6.10: Implement approved application list

Guidance: Not applicable; this guideline is intended for compute resources.

Azure Security Center monitoring: Not applicable

Responsibility: Not applicable

6.11: Limit users' ability to interact with AzureResources Manager via scripts

Guidance: Use the Azure Conditional Access to limit users' ability to interact with Azure Resource Manager by configuring "Block access" for the "Microsoft Azure Management" App. This can prevent the creation and changes to resources within a high security environment.

How to configure Conditional Access to block access to Azure Resource Manager: https://docs.microsoft.com/azure/role-based-access-control/conditional-access-azure-management

Azure Security Center monitoring: Not applicable

Responsibility: Customer

6.12: Limit users' ability to execute scripts within compute resources

Guidance: Not applicable; this guideline is intended for compute resources.

Azure Security Center monitoring: Not applicable

Responsibility: Not applicable

6.13: Physically or logically segregate high risk applications

Guidance: Not applicable; this guideline is intended for web applications running on Azure App Service or compute resources.

Azure Security Center monitoring: Not applicable

Responsibility: Not applicable

Secure Configuration

For more information, see Security Control: Secure Configuration.

7.1: Establish secure configurations for all Azure resources

Guidance: Define and implement standard security configurations for your Cosmos DB instances with Azure Policy. Use Azure Policy aliases in the "Microsoft.DocumentDB" namespace to create custom policies to audit or enforce the configuration of your Cosmos DB instances. You may also make use of built-in policy definitions for Azure Cosmos DB, such as:

- Deploy Advanced Threat Protection for Cosmos DB Accounts
- Cosmos DB should use a virtual network service endpoint

How to view available Azure Policy aliases: https://docs.microsoft.com/powershell/module/az.resources/get-azpolicyalias?view=azps-3.3.0&preserve-view=true

How to configure and manage Azure Policy: https://docs.microsoft.com/azure/governance/policy/tutorials/create-and-manage

Azure Security Center monitoring: Not applicable

Responsibility: Customer

7.2: Establish secure operating system configurations

Guidance: Not applicable; this guideline is intended for compute resources.

Azure Security Center monitoring: Not applicable

Responsibility: Not applicable

7.3: Maintain secure Azure resource configurations

Guidance: Use Azure Policy [deny] and [deploy if not exist] to enforce secure settings across your Azure resources.

How to configure and manage Azure Policy: https://docs.microsoft.com/azure/governance/policy/tutorials/create-and-manage

Understand Azure Policy Effects: https://docs.microsoft.com/azure/governance/policy/concepts/effects

Azure Security Center monitoring: Not applicable

Responsibility: Customer

7.4: Maintain secure operating system configurations

Guidance: Not applicable; this guideline is intended for compute resources.

Azure Security Center monitoring: Not applicable

Responsibility: Not applicable

7.5: Securely store configuration of Azure resources

Guidance: If using custom Azure Policy definitions for your Cosmos DB or related resources, use Azure Repos to securely store and manage your code.

Azure Repos Documentation: https://docs.microsoft.com/azure/devops/repos/index?view=azure-devops&preserve-view=true https://docs.microsoft.com/azure/devops/repos/git/gitworkflow?view=azure-devops&preserve-view=true

Azure Security Center monitoring: Not applicable

Responsibility: Customer

7.6: Securely store custom operating system images

Guidance: Not applicable; this guideline is intended for compute resources.

Azure Security Center monitoring: Not applicable

Responsibility: Not applicable

7.7: Deploy system configuration management tools

Guidance: Use Azure Policy aliases in the "Microsoft.DocumentDB" namespace to create custom policies to alert, audit, and enforce system configurations. Additionally, develop a process and pipeline for managing policy exceptions.

How to configure and manage Azure Policy: https://docs.microsoft.com/azure/governance/policy/tutorials/create-and-manage

Azure Security Center monitoring: Not applicable

Responsibility: Customer

7.8: Deploy system configuration management tools for operating systems

Guidance: Not applicable; this guideline is intended for compute resources.

Azure Security Center monitoring: Not applicable

Responsibility: Not applicable

7.9: Implement automated configuration monitoring for Azure services

Guidance: Use Azure Policy aliases in the "Microsoft.DocumentDB" namespace to create custom policies to alert, audit, and enforce system configurations. Use Azure Policy [audit], [deny], and [deploy if not exist] to automatically enforce configurations for your Azure Cosmos DB instances and related resources.

How to configure and manage Azure Policy: https://docs.microsoft.com/azure/governance/policy/tutorials/create-and-manage

Azure Security Center monitoring: Not applicable

Responsibility: Customer

7.10: Implement automated configuration monitoring for operating systems

Guidance: Not applicable; this guideline is intended for compute resources.

Azure Security Center monitoring: Not applicable

Responsibility: Not applicable

7.11: Manage Azure secrets securely

Guidance: For Azure virtual machines or web applications running on Azure App Service being used to access your Azure Cosmos DB instances, use Managed Service Identity in conjunction with Azure Key Vault to simplify and secure Azure Cosmos DB secret management. Ensure Key Vault Soft Delete is enabled.

How to integrate with Azure Managed Identities: https://docs.microsoft.com/azure/azure-app-configuration/howto-integrate-azure-managed-service-identity

How to create a Key Vault: https://docs.microsoft.com/azure/key-vault/general/quick-create-portal

How to authenticate to Key Vault: https://docs.microsoft.com/azure/key-vault/general/authentication

How to assign a Key Vault access policy: https://docs.microsoft.com/azure/key-vault/general/assign-access-policy-portal

Azure Security Center monitoring: Yes

Responsibility: Customer

7.12: Manage identities securely and automatically

Guidance: For Azure virtual machines or web applications running on Azure App Service being used to access your Azure Cosmos DB instances, use Managed Service Identity in conjunction with Azure Key Vault to simplify and secure Azure Cosmos DB secret management.

Use Managed Identities to provide Azure services with an automatically managed identity in Azure Active Directory (AD). Managed Identities allows you to authenticate to any service that supports Azure AD authentication, including Key Vault, without any credentials in your code.

How to configure Managed Identities: https://docs.microsoft.com/azure/active-directory/managed-identities-azure-resources/qs-configure-portal-windows-vm

How to integrate with Azure Managed Identities: https://docs.microsoft.com/azure/azure-app-configuration/howto-integrate-azure-managed-service-identity

Azure Security Center monitoring: Yes

Responsibility: Customer

7.13: Eliminate unintended credential exposure

Guidance: Implement Credential Scanner to identify credentials within code. Credential Scanner will also encourage moving discovered credentials to more secure locations such as Azure Key Vault.

How to set up Credential Scanner: https://secdevtools.azurewebsites.net/helpcredscan.html

Azure Security Center monitoring: Not applicable

Responsibility: Customer

Malware Defense

For more information, see Security Control: Malware Defense.

8.1: Use centrally managed anti-malware software

Guidance: Not applicable; this guideline is intended for compute resources. Microsoft Antimalware is enabled on the underlying host that supports Azure services (for example, Azure App Service), however it does not run on customer content.

Azure Security Center monitoring: Not applicable

Responsibility: Not applicable

8.2: Pre-scan files to be uploaded to non-compute Azure resources

Guidance: Microsoft Antimalware is enabled on the underlying host that supports Azure services (for example, Azure App Service), however it does not run on customer content.

It is your responsibility to pre-scan any files being uploaded to non-compute Azure resources, including Azure Cosmos DB. Microsoft cannot access customer data, and therefore cannot conduct anti-malware scans of customer content on your behalf.

Azure Security Center monitoring: Not applicable

Responsibility: Customer

8.3: Ensure anti-malware software and signatures are updated

Guidance: Not applicable; benchmark is intended for compute resources. Microsoft Antimalware is enabled on the underlying host that supports Azure services, however it does not run on customer content.

Azure Security Center monitoring: Not applicable

Responsibility: Not applicable

Data Recovery

For more information, see Security Control: Data Recovery.

9.1: Ensure regular automated back ups

Guidance: Azure Cosmos DB takes snapshots of your data every four hours. All the backups are stored separately in a storage service, and those backups are globally replicated for resiliency against regional disasters. At any given time, only the last two snapshots are retained. However, if the container or database is deleted, Azure Cosmos DB retains the existing snapshots of a given container or database for 30 days. Contact Azure Support to restore from a backup.

Understanding Azure Cosmos DB Automated Backups: https://docs.microsoft.com/azure/cosmos-db/online-backup-and-restore

Azure Security Center monitoring: Not applicable

Responsibility: Microsoft

9.2: Perform complete system backups and backup any customer managed keys

Guidance: Azure Cosmos DB automatically takes backups of your data at regular intervals. If database or container is deleted, you can file a support ticket or call Azure support to restore the data from automatic online backups. Azure support is available for selected plans only such as Standard, Developer, and plans higher than them. To restore a specific snapshot of the backup, Azure Cosmos DB requires that the data is available for the duration of the backup cycle for that snapshot.

If using Key Vault to store credentials for your Cosmos DB instances, ensure regular automated backups of your keys.

Understand Azure Cosmos DB Automated Backups: https://docs.microsoft.com/azure/cosmos-db/online-backup-and-restore

How to restore data in Azure Cosmos DB: https://docs.microsoft.com/azure/cosmos-db/how-to-backup-and-restore

How to backup Key Vault Keys: https://docs.microsoft.com/powershell/module/azurerm.keyvault/backup-azurekeyvaultkey

Azure Security Center monitoring: Currently not available

Responsibility: Shared

9.3: Validate all backups including customer managed keys

Guidance: If database or container is deleted, you can file a support ticket or call Azure support to restore the data from automatic online backups. Azure support is available for selected plans only such as Standard, Developer, and plans higher than them. To restore a specific snapshot of the backup, Azure Cosmos DB requires that the data is available for the duration of the backup cycle for that snapshot.

Test restoration of your secrets stored in Azure Key Vault using PowerShell. The Restore-AzureKeyVaultKey cmdlet creates a key in the specified key vault. This key is a replica of the backed-up key in the input file and has the same name as the original key.

Understand Azure Cosmos DB Automated Backups:

https://docs.microsoft.com/azure/cosmos-db/online-backup-and-restore

How to restore data in Azure Cosmos DB:

https://docs.microsoft.com/azure/cosmos-db/how-to-backup-and-restore

How to restore Azure Key Vault Secrets:

https://docs.microsoft.com/powershell/module/azurerm.keyvault/restore-azurekeyvaultkey?view=azurermps-6.13.0&preserve-view=true

Azure Security Center monitoring: Not applicable

Responsibility: Shared

9.4: Ensure protection of backups and customer managed keys

Guidance: Because all user data stored in Cosmos DB is encrypted at rest and in transport, you don't have to take any action. Another way to put this is that encryption at rest is "on" by default. There are no controls to turn it off or on. Azure Cosmos DB uses AES-256 encryption on all regions where the account is running.

Enable Soft-Delete in Key Vault to protect keys against accidental or malicious deletion.

Understand data encryption in Azure Cosmos DB: https://docs.microsoft.com/azure/cosmos-db/database-encryption-at-rest

How to enable Soft-Delete in Key Vault: https://docs.microsoft.com/azure/storage/blobs/storage-blob-soft-delete?tabs=azure-portal

Azure Security Center monitoring: Yes

Responsibility: Shared

Incident Response

For more information, see Security Control: Incident Response.

10.1: Create an incident response guide

Guidance: Build out an incident response guide for your organization. Ensure that there are written incident response plans that define all roles of personnel as well as phases of incident handling/management from detection to post-incident review.

You may also leverage NIST's Computer Security Incident Handling Guide to aid in the creation of your own incident response plan: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-61r2.pdf

How to configure Workflow Automations within Azure Security Center: https://docs.microsoft.com/azure/security-center/security-center-planning-and-operations-guide

Guidance on building your own security incident response process: https://msrc-blog.microsoft.com/2019/07/01/inside-the-msrc-building-your-own-security-incident-response-process/

Microsoft Security Response Center's Anatomy of an Incident: https://msrc-blog.microsoft.com/2019/07/01/inside-the-msrc-building-your-own-security-incident-response-process/

Azure Security Center monitoring: Not applicable

Responsibility: Customer

10.2: Create an incident scoring and prioritization procedure

Guidance: Security Center assigns a severity to each alert to help you prioritize which alerts should be investigated first. The severity is based on how confident the Security Center is in finding or the analytics used to

issue the alert as well as the confidence level that there was malicious intent behind the activity that led to the alert.

Additionally, clearly mark subscriptions (for ex. production, non-prod) and create a naming system to clearly identify and categorize Azure resources.

Azure Security Center monitoring: Not applicable

Responsibility: Customer

10.3: Test security response procedures

Guidance: Conduct exercises to test your systems' incident response capabilities on a regular cadence. Identify weak points and gaps and revise plan as needed.

Refer to NIST's publication: Guide to Test, Training, and Exercise Programs for IT Plans and Capabilities: https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-84.pdf

Azure Security Center monitoring: Not applicable

Responsibility: Customer

10.4: Provide security incident contact details and configure alert notifications for security incidents

Guidance: Security incident contact information will be used by Microsoft to contact you if the Microsoft Security Response Center (MSRC) discovers that the customer's data has been accessed by an unlawful or unauthorized party. Review incidents after the fact to ensure that issues are resolved.

How to set the Azure Security Center Security Contact: https://docs.microsoft.com/azure/security-center/security-center-provide-security-contact-details

Azure Security Center monitoring: Yes

Responsibility: Customer

10.5: Incorporate security alerts into your incident response system

Guidance: Export your Azure Security Center alerts and recommendations using the Continuous Export feature. Continuous Export allows you to export alerts and recommendations either manually or in an ongoing, continuous fashion. You may use the Azure Security Center data connector to stream the alerts Sentinel.

How to configure continuous export: https://docs.microsoft.com/azure/security-center/continuous-export

How to stream alerts into Azure Sentinel: https://docs.microsoft.com/azure/sentinel/connect-azure-security-center

Azure Security Center monitoring: Not applicable

Responsibility: Customer

10.6: Automate the response to security alerts

Guidance: Use the Workflow Automation feature in Azure Security Center to automatically trigger responses via "Logic Apps" on security alerts and recommendations.

How to configure Workflow Automation and Logic Apps: https://docs.microsoft.com/azure/security-center/workflow-automation

Azure Security Center monitoring: Not applicable

Responsibility: Customer

Penetration Tests and Red Team Exercises

For more information, see Security Control: Penetration Tests and Red Team Exercises.

11.1: Conduct regular penetration testing of your Azure resources and ensure remediation of all critical security

findings within 60 days

Guidance: Follow the Microsoft Rules of Engagement to ensure your Penetration Tests are not in violation of Microsoft policies: https://www.microsoft.com/msrc/pentest-rules-of-engagement?rtc=1

You can find more information on Microsoft's strategy and execution of Red Teaming and live site penetration testing against Microsoft-managed cloud infrastructure, services, and applications, here: https://gallery.technet.microsoft.com/Cloud-Red-Teaming-b837392e

Azure Security Center monitoring: Not applicable

Responsibility: Shared

Next steps

- See the Azure Security Benchmark
- Learn more about Azure Security Baselines

What is Azure Synapse Link for Azure Cosmos DB?

12/4/2020 • 8 minutes to read • Edit Online

APPLIES TO: SQL API Azure Cosmos DB API for MongoDB

> **IMPORTANT**
>
> Synapse serverless SQL pool support for Azure Synapse Link for Azure Cosmos DB is currently in preview. This preview version is provided without a service level agreement, and it's not recommended for production workloads. For more information, see Supplemental terms of use for Microsoft Azure previews.

Azure Synapse Link for Azure Cosmos DB is a cloud-native hybrid transactional and analytical processing (HTAP) capability that enables you to run near real-time analytics over operational data in Azure Cosmos DB. Azure Synapse Link creates a tight seamless integration between Azure Cosmos DB and Azure Synapse Analytics.

Using Azure Cosmos DB analytical store, a fully isolated column store, Azure Synapse Link enables no Extract-Transform-Load (ETL) analytics in Azure Synapse Analytics against your operational data at scale. Business analysts, data engineers and data scientists can now use Synapse Spark or Synapse SQL interchangeably to run near real-time business intelligence, analytics, and machine learning pipelines. You can achieve this without impacting the performance of your transactional workloads on Azure Cosmos DB.

The following image shows the Azure Synapse Link integration with Azure Cosmos DB and Azure Synapse Analytics:

Benefits

To analyze large operational datasets while minimizing the impact on the performance of mission-critical transactional workloads, traditionally, the operational data in Azure Cosmos DB is extracted and processed by Extract-Transform-Load (ETL) pipelines. ETL pipelines require many layers of data movement resulting in much operational complexity, and performance impact on your transactional workloads. It also increases the latency to analyze the operational data from the time of origin.

When compared to the traditional ETL-based solutions, Azure Synapse Link for Azure Cosmos DB offers several advantages such as:

Reduced complexity with No ETL jobs to manage

Azure Synapse Link allows you to directly access Azure Cosmos DB analytical store using Azure Synapse Analytics without complex data movement. Any updates made to the operational data are visible in the analytical store in near real-time with no ETL or change feed jobs. You can run large scale analytics against analytical store, from Azure Synapse Analytics, without additional data transformation.

Near real-time insights into your operational data

You can now get rich insights on your operational data in near real-time, using Azure Synapse Link. ETL-based systems tend to have higher latency for analyzing your operational data, due to many layers needed to extract, transform and load the operational data. With native integration of Azure Cosmos DB analytical store with Azure Synapse Analytics, you can analyze operational data in near real-time enabling new business scenarios.

No impact on operational workloads

With Azure Synapse Link, you can run analytical queries against an Azure Cosmos DB analytical store (a separate column store) while the transactional operations are processed using provisioned throughput for the transactional workload (a row-based transactional store). The analytical workload is served independent of the transactional workload traffic without consuming any of the throughput provisioned for your operational data.

Optimized for large-scale analytics workloads

Azure Cosmos DB analytical store is optimized to provide scalability, elasticity, and performance for analytical workloads without any dependency on the compute run-times. The storage technology is self-managed to optimize your analytics workloads. With built-in support into Azure Synapse Analytics, accessing this storage layer provides simplicity and high performance.

Cost effective

With Azure Synapse Link, you can get a cost-optimized, fully managed solution for operational analytics. It eliminates the extra layers of storage and compute required in traditional ETL pipelines for analyzing operational data.

Azure Cosmos DB analytical store follows a consumption-based pricing model, which is based on data storage and analytical read/write operations and queries executed . It doesn't require you to provision any throughput, as you do today for the transactional workloads. Accessing your data with highly elastic compute engines from Azure Synapse Analytics makes the overall cost of running storage and compute very efficient.

Analytics for locally available, globally distributed, multi-region writes

You can run analytical queries effectively against the nearest regional copy of the data in Azure Cosmos DB. Azure Cosmos DB provides the state-of-the-art capability to run the globally distributed analytical workloads along with transactional workloads in an active-active manner.

Enable HTAP scenarios for your operational data

Synapse Link brings together Azure Cosmos DB analytical store with Azure Synapse analytics runtime support. This integration enables you to build cloud native HTAP (Hybrid transactional/analytical processing) solutions that generate insights based on real-time updates to your operational data over large datasets. It unlocks new business scenarios to raise alerts based on live trends, build near real-time dashboards, and business experiences based on user behavior.

Azure Cosmos DB analytical store

Azure Cosmos DB analytical store is a column-oriented representation of your operational data in Azure Cosmos DB. This analytical store is suitable for fast, cost effective queries on large operational data sets, without copying data and impacting the performance of your transactional workloads.

Analytical store automatically picks up high frequency inserts, updates, deletes in your transactional workloads in near real time, as a fully managed capability ("auto-sync") of Azure Cosmos DB. No change feed or ETL is

required.

If you have a globally distributed Azure Cosmos DB account, after you enable analytical store for a container, it will be available in all regions for that account. For more information on the analytical store, see Azure Cosmos DB Analytical store overview article.

Integration with Azure Synapse Analytics

With Synapse Link, you can now directly connect to your Azure Cosmos DB containers from Azure Synapse Analytics and access the analytical store with no separate connectors. Azure Synapse Analytics currently supports Synapse Link with Synapse Apache Spark and serverless SQL pool.

You can query the data from Azure Cosmos DB analytical store simultaneously, with interop across different analytics run times supported by Azure Synapse Analytics. No additional data transformations are required to analyze the operational data. You can query and analyze the analytical store data using:

- Synapse Apache Spark with full support for Scala, Python, SparkSQL, and C#. Synapse Spark is central to data engineering and data science scenarios
- Serverless SQL pool with T-SQL language and support for familiar BI tools (for example, Power BI Premium, etc.)

> **NOTE**
> From Azure Synapse Analytics, you can access both analytical and transactional stores in your Azure Cosmos DB container. However, if you want to run large-scale analytics or scans on your operational data, we recommend that you use analytical store to avoid performance impact on transactional workloads.

> **NOTE**
> You can run analytics with low latency in an Azure region by connecting your Azure Cosmos DB container to Synapse runtime in that region.

This integration enables the following HTAP scenarios for different users:

- A BI engineer who wants to model and publish a Power BI report to access the live operational data in Azure Cosmos DB directly through Synapse SQL.
- A data analyst who wants to derive insights from the operational data in an Azure Cosmos DB container by querying it with Synapse SQL, read the data at scale and combine those findings with other data sources.
- A data scientist who wants to use Synapse Spark to find a feature to improve their model and train that model without doing complex data engineering. They can also write the results of the model post inference into Azure Cosmos DB for real-time scoring on the data through Spark Synapse.
- A data engineer who wants to make data accessible for consumers, by creating SQL or Spark tables over Azure Cosmos DB containers without manual ETL processes.

For more information on Azure Synapse Analytics runtime support for Azure Cosmos DB, see Azure Synapse Analytics for Cosmos DB support.

When to use Azure Synapse Link for Azure Cosmos DB?

Synapse Link is recommended in the following cases:

- If you are an Azure Cosmos DB customer and you want to run analytics, BI, and machine learning over your operational data. In such cases, Synapse Link provides a more integrated analytics experience without

impacting your transactional store's provisioned throughput. For example:

  - If you are running analytics or BI on your Azure Cosmos DB operational data directly using separate connectors today, or
  - If you are running ETL processes to extract operational data into a separate analytics system.

In such cases, Synapse Link provides a more integrated analytics experience without impacting your transactional store's provisioned throughput.

Synapse Link is not recommended if you are looking for traditional data warehouse requirements such as high concurrency, workload management, and persistence of aggregates across multiple data sources. For more information, see common scenarios that can be powered with Azure Synapse Link for Azure Cosmos DB.

Limitations

- Today Azure Synapse Link for Azure Cosmos DB is supported for SQL API and Azure Cosmos DB API for MongoDB. It is not supported for Gremlin API and Table API. Support for Cassandra API is in private preview, for more information please contact the Azure Synapse Link team.
- Currently, the analytical store can only be enabled for new containers. To use analytical store for existing containers, migrate data from your existing containers to new containers using Azure Cosmos DB migration tools. You can enable Synapse Link on new and existing Azure Cosmos DB accounts.
- For the containers with analytical store turned on, automatic backup and restore of your data in the analytical store is not supported at this time. When Synapse Link is enabled on a database account, Azure Cosmos DB will continue to automatically take backups of your data in the transactional store (only) of containers at scheduled backup interval, as always. It is important to note that when a container with analytical store turned on is restored to a new account, the container will be restored with only transactional store and no analytical store enabled.
- Accessing the Azure Cosmos DB analytics store with Synapse SQL provisioned is currently not available.

Pricing

The billing model of Azure Synapse Link includes the costs incurred by using the Azure Cosmos DB analytical store and the Synapse runtime. To learn more, see the Azure Cosmos DB analytical store pricing and Azure Synapse Analytics pricing articles.

Next steps

To learn more, see the following docs:

- Azure Cosmos DB analytical store overview
- Get started with Azure Synapse Link for Azure Cosmos DB
- What is supported in Azure Synapse Analytics run time
- Frequently asked questions about Azure Synapse Link for Azure Cosmos DB
- Azure Synapse Link for Azure Cosmos DB Use cases

What is Azure Cosmos DB analytical store?

12/4/2020 • 14 minutes to read • Edit Online

APPLIES TO: SQL API Azure Cosmos DB API for MongoDB

Azure Cosmos DB analytical store is a fully isolated column store for enabling large-scale analytics against operational data in your Azure Cosmos DB, without any impact to your transactional workloads.

Azure Cosmos DB transactional store is schema-agnostic, and it allows you to iterate on your transactional applications without having to deal with schema or index management. In contrast to this, Azure Cosmos DB analytical store is schematized to optimize for analytical query performance. This article describes in detailed about analytical storage.

Challenges with large-scale analytics on operational data

The multi-model operational data in an Azure Cosmos DB container is internally stored in an indexed row-based "transactional store". Row store format is designed to allow fast transactional reads and writes in the order-of-milliseconds response times, and operational queries. If your dataset grows large, complex analytical queries can be expensive in terms of provisioned throughput on the data stored in this format. High consumption of provisioned throughput in turn, impacts the performance of transactional workloads that are used by your real-time applications and services.

Traditionally, to analyze large amounts of data, operational data is extracted from Azure Cosmos DB's transactional store and stored in a separate data layer. For example, the data is stored in a data warehouse or data lake in a suitable format. This data is later used for large-scale analytics and analyzed using compute engine such as the Apache Spark clusters. This separation of analytical storage and compute layers from operational data results in additional latency, because the ETL(Extract, Transform, Load) pipelines are run less frequently to minimize the potential impact on your transactional workloads.

The ETL pipelines also become complex when handling updates to the operational data when compared to handling only newly ingested operational data.

Column-oriented analytical store

Azure Cosmos DB analytical store addresses the complexity and latency challenges that occur with the traditional ETL pipelines. Azure Cosmos DB analytical store can automatically sync your operational data into a separate column store. Column store format is suitable for large-scale analytical queries to be performed in an optimized manner, resulting in improving the latency of such queries.

Using Azure Synapse Link, you can now build no-ETL HTAP solutions by directly linking to Azure Cosmos DB analytical store from Azure Synapse Analytics. It enables you to run near real-time large-scale analytics on your operational data.

Features of analytical store

When you enable analytical store on an Azure Cosmos DB container, a new column-store is internally created based on the operational data in your container. This column store is persisted separately from the row-oriented transactional store for that container. The inserts, updates, and deletes to your operational data are automatically synced to analytical store. You don't need the change feed or ETL to sync the data.

Column store for analytical workloads on operational data

Analytical workloads typically involve aggregations and sequential scans of selected fields. By storing the data in a column-major order, the analytical store allows a group of values for each field to be serialized together. This format reduces the IOPS required to scan or compute statistics over specific fields. It dramatically improves the query response times for scans over large data sets.

For example, if your operational tables are in the following format:

ID	Product Name	Product Code	Product Category
1	Product1	123	Books
2	Product2	250	Equipment
3	Product3	380	Equipment

The row store persists the above data in a serialized format, per row, on the disk. This format allows for faster transactional reads, writes, and operational queries, such as, "Return information about Product1". However, as the dataset grows large and if you want to run complex analytical queries on the data it can be expensive. For example, if you want to get "the sales trends for a product under the category named 'Equipment' across different business units and months", you need to run a complex query. Large scans on this dataset can get expensive in terms of provisioned throughput and can also impact the performance of the transactional workloads powering your real-time applications and services.

Analytical store, which is a column store, is better suited for such queries because it serializes similar fields of data together and reduces the disk IOPS.

The following image shows transactional row store vs. analytical column store in Azure Cosmos DB:

Decoupled performance for analytical workloads

There is no impact on the performance of your transactional workloads due to analytical queries, as the analytical store is separate from the transactional store. Analytical store does not need separate request units (RUs) to be allocated.

Auto-Sync

Auto-Sync refers to the fully managed capability of Azure Cosmos DB where the inserts, updates, deletes to operational data are automatically synced from transactional store to analytical store in near real time. Auto-sync latency is usually within 2 minutes. In cases of shared throughput database with a large number of containers, auto-sync latency of individual containers could be higher and take up to 5 minutes. We would like to learn more

how this latency fits your scenarios. For that, please reach out to the Azure Cosmos DB team.

The auto-sync capability along with analytical store provides the following key benefits:

Scalability & elasticity

By using horizontal partitioning, Azure Cosmos DB transactional store can elastically scale the storage and throughput without any downtime. Horizontal partitioning in the transactional store provides scalability & elasticity in auto-sync to ensure data is synced to the analytical store in near real time. The data sync happens regardless of the transactional traffic throughput, whether it is 1000 operations/sec or 1 million operations/sec, and it doesn't impact the provisioned throughput in the transactional store.

Automatically handle schema updates

Azure Cosmos DB transactional store is schema-agnostic, and it allows you to iterate on your transactional applications without having to deal with schema or index management. In contrast to this, Azure Cosmos DB analytical store is schematized to optimize for analytical query performance. With the auto-sync capability, Azure Cosmos DB manages the schema inference over the latest updates from the transactional store. It also manages the schema representation in the analytical store out-of-the-box which, includes handling nested data types.

As your schema evolves, and new properties are added over time, the analytical store automatically presents a unionized schema across all historical schemas in the transactional store.

Schema constraints

The following constraints are applicable on the operational data in Azure Cosmos DB when you enable analytical store to automatically infer and represent the schema correctly:

- You can have a maximum of 200 properties at any nesting level in the schema and a maximum nesting depth of 5.
  - An item with 201 properties at the top level doesn’t satisfy this constraint and hence it will not be represented in the analytical store.
  - An item with more than five nested levels in the schema also doesn’t satisfy this constraint and hence it will not be represented in the analytical store. For example, the following item doesn't satisfy the requirement:

    ```
    {"level1": {"level2":{"level3":{"level4":{"level5":{"too many":12}}}}}}
    ```

- Property names should be unique when compared case insensitively. For example, the following items do not satisfy this constraint and hence will not be represented in the analytical store:

  `{"Name": "fred"} {"name": "john"}` – "Name" and "name" are the same when compared in a case insensitive manner.

Schema representation

There are two modes of schema representation in the analytical store. These modes have tradeoffs between the simplicity of a columnar representation, handling the polymorphic schemas, and simplicity of query experience:

- Well-defined schema representation
- Full fidelity schema representation

> **NOTE**
>
> For SQL (Core) API accounts, when analytical store is enabled, the default schema representation in the analytical store is well-defined. Whereas for Azure Cosmos DB API for MongoDB accounts, the default schema representation in the analytical store is a full fidelity schema representation. If you have scenarios requiring a different schema representation than the default offering for each of these APIs, reach out to the Azure Cosmos DB team to enable it.

Well-defined schema representation

The well-defined schema representation creates a simple tabular representation of the schema-agnostic data in the transactional store. The well-defined schema representation has the following considerations:

- A property always has the same type across multiple items.
  - For example, `{"a":123} {"a": "str"}` does not have a well-defined schema because `"a"` is sometimes a string and sometimes a number. In this case, the analytical store registers the data type of `“a”` as the data type of `“a”` in the first-occurring item in the lifetime of the container. Items where the data type of `“a”` differs will not be included in the analytical store.

    This condition does not apply for null properties. For example, `{"a":123} {"a":null}` is still well defined.
- Array types must contain a single repeated type.
  - For example, `{"a": ["str",12]}` is not a well-defined schema because the array contains a mix of integer and string types.

> **NOTE**
> If the Azure Cosmos DB analytical store follows the well-defined schema representation and the specification above is violated by certain items, those items will not be included in the analytical store.

Full fidelity schema representation

The full fidelity schema representation is designed to handle the full breadth of polymorphic schemas in the schema-agnostic operational data. In this schema representation, no items are dropped from the analytical store even if the well-defined schema constraints (that is no mixed data type fields nor mixed data type arrays) are violated.

This is achieved by translating the leaf properties of the operational data into the analytical store with distinct columns based on the data type of values in the property. The leaf property names are extended with data types as a suffix in the analytical store schema such that they can be queries without ambiguity.

For example, let’s take the following sample document in the transactional store:

```
{
name: "John Doe",
age: 32,
profession: "Doctor",
address: {
  streetNo: 15850,
  streetName: "NE 40th St.",
  zip: 98052
},
salary: 1000000
}
```

The leaf property `streetNo` within the nested object `address` will be represented in the analytical store schema as a column `address.object.streetNo.int32`. The datatype is added as a suffix to the column. This way, if another document is added to the transactional store where the value of leaf property `streetNo` is "123" (note it’s a string), the schema of the analytical store automatically evolves without altering the type of a previously written column. A new column added to the analytical store as `address.object.streetNo.string` where this value of "123" is stored.

Data type to suffix map

Here is a map of all the property data types and their suffix representations in the analytical store:

ORIGINAL DATA TYPE	SUFFIX	EXAMPLE
Double	".float64"	24.99
Array	".array"	["a", "b"]
Binary	".binary"	0
Boolean	".bool"	True
Int32	".int32"	123
Int64	".int64"	255486129307
Null	".null"	null
String	".string"	"ABC"
Timestamp	".timestamp"	Timestamp(0, 0)
DateTime	".date"	ISODate("2020-08-21T07:43:07.375Z")
ObjectId	".objectId"	ObjectId("5f3f7b59330ec25c132623a2")
Document	".object"	{"a": "a"}

Cost-effective archival of historical data

Data tiering refers to the separation of data between storage infrastructures optimized for different scenarios. Thereby improving the overall performance and cost-effectiveness of the end-to-end data stack. With analytical store, Azure Cosmos DB now supports automatic tiering of data from the transactional store to analytical store with different data layouts. With analytical store optimized in terms of storage cost compared to the transactional store, allows you to retain much longer horizons of operational data for historical analysis.

After the analytical store is enabled, based on the data retention needs of the transactional workloads, you can configure the 'Transactional Store Time to Live (Transactional TTL)' property to have records automatically deleted from the transactional store after a certain time period. Similarly, the 'Analytical Store Time To Live (Analytical TTL)' allows you to manage the lifecycle of data retained in the analytical store independent from the transactional store. By enabling analytical store and configuring TTL properties, you can seamlessly tier and define the data retention period for the two stores.

Global Distribution

If you have a globally distributed Azure Cosmos DB account, after you enable analytical store for a container, it will be available in all regions of that account. Any changes to operational data are globally replicated in all regions. You can run analytical queries effectively against the nearest regional copy of your data in Azure Cosmos DB.

Security

Authentication with the analytical store is the same as the transactional store for a given database. You can use primary or read-only keys for authentication. You can leverage linked service in Synapse Studio to prevent pasting the Azure Cosmos DB keys in the Spark notebooks. Access to this Linked Service is available to anyone who has access into the workspace.

Support for multiple Azure Synapse Analytics runtimes

The analytical store is optimized to provide scalability, elasticity, and performance for analytical workloads without any dependency on the compute run-times. The storage technology is self-managed to optimize your analytics workloads without manual efforts.

By decoupling the analytical storage system from the analytical compute system, data in Azure Cosmos DB analytical store can be queried simultaneously from the different analytics runtimes supported by Azure Synapse Analytics. As of today, Azure Synapse Analytics supports Apache Spark and serverless SQL pool with Azure Cosmos DB analytical store.

> **NOTE**
> You can only read from analytical store using Azure Synapse Analytics run time. You can write the data back to your transactional store as a serving layer.

Pricing

Analytical store follows a consumption-based pricing model where you are charged for:

- Storage: the volume of the data retained in the analytical store every month including historical data as defined by Analytical TTL.
- Analytical write operations: the fully managed synchronization of operational data updates to the analytical store from the transactional store (auto-sync)
- Analytical read operations: the read operations performed against the analytical store from Azure Synapse Analytics Spark pool and serverless SQL pool run times.

Analytical store pricing is separate from the transaction store pricing model. There is no concept of provisioned RUs in the analytical store. See Azure Cosmos DB pricing page, for full details on the pricing model for analytical store.

In order to get a high-level cost estimate to enable analytical store on an Azure Cosmos DB container, you can use the Azure Cosmos DB Capacity planner and get an estimate of your analytical storage and write operations costs. Analytical read operations costs depends on the analytics workload characteristics but as a high-level estimate, scan of 1 TB of data in analytical store typically results in 130,000 analytical read operations, and results in a cost of $0.065.

Analytical Time-to-Live (TTL)

Analytical TTL indicates how long data should be retained in your analytical store, for a container.

If analytical store is enabled, inserts, updates, deletes to operational data are automatically synced from transactional store to analytical store, irrespective of the transactional TTL configuration. The retention of this operational data in the analytical store can be controlled by the Analytical TTL value at the container level, as specified below:

Analytical TTL on a container is set using the `AnalyticalStoreTimeToLiveInSeconds` property:

- If the value is set to "0", missing (or set to null): the analytical store is disabled and no data is replicated from transactional store to analytical store
- If present and the value is set to "-1": the analytical store retains all historical data, irrespective of the retention of the data in the transactional store. This setting indicates that the analytical store has infinite retention of your operational data
- If present and the value is set to some positive number "n": items will expire from the analytical store "n" seconds after their last modified time in the transactional store. This setting can be leveraged if you want to

retain your operational data for a limited period of time in the analytical store, irrespective of the retention of the data in the transactional store

Some points to consider:

- After the analytical store is enabled with an analytical TTL value, it can be updated to a different valid value later.
- While transactional TTL can be set at the container or item level, analytical TTL can only be set at the container level currently.
- You can achieve longer retention of your operational data in the analytical store by setting analytical TTL >= transactional TTL at the container level.
- The analytical store can be made to mirror the transactional store by setting analytical TTL = transactional TTL.

When you enable analytical store on a container:

- From the Azure portal, the analytical TTL option is set to the default value of -1. You can change this value to 'n' seconds, by navigating to container settings under Data Explorer.
- From the Azure SDK or PowerShell or CLI, the analytical TTL option can be enabled by setting it to either -1 or 'n'.

To learn more, see how to configure analytical TTL on a container.

Next steps

To learn more, see the following docs:

- Azure Synapse Link for Azure Cosmos DB
- Get started with Azure Synapse Link for Azure Cosmos DB
- Frequently asked questions about Synapse Link for Azure Cosmos DB
- Azure Synapse Link for Azure Cosmos DB Use cases

Azure Synapse Link for Azure Cosmos DB supported features

12/4/2020 • 2 minutes to read • Edit Online

This article describes the functionalities that are currently supported in Azure Synapse Link for Azure Cosmos DB.

Azure Synapse support

There are two types of containers in Azure Cosmos DB:

- HTAP container - A container with Synapse Link enabled. This container has both transactional store and analytical store.
- OLTP container - A container with Synaspe Link not enabled. This container has only transactional store and no analytical store.

> **IMPORTANT**
> Azure Synapse Link for Azure Cosmos DB is currently supported in Synapse workspaces that do not have managed virtual network enabled.

You can connect to an Azure Cosmos DB container without enabling Synapse Link. In this scenario, you can only read/write to the transactional store. What follows is a list of the currently supported features within Synapse Link for Azure Cosmos DB.

CATEGORY	DESCRIPTION	APACHE SPARK POOL	SERVERLESS SQL POOL
Run-time Support	Supported Azure Synapse runtime to access Azure Cosmos DB	✓	Preview
Azure Cosmos DB API support	Supported Azure Cosmos DB API kind	SQL / MongoDB	SQL / MongoDB
Object	Objects such as a table that can be created, pointing directly to Azure Cosmos DB container	Dataframe, View, Table	View
Read	Type of Azure Cosmos DB container that can be read	OLTP / HTAP	HTAP
Write	Can the Azure Synapse runtime be used to write data to an Azure Cosmos DB container	Yes	No

- If you write data into an Azure Cosmos DB container from Spark, this process happens through the transactional store of Azure Cosmos DB. It will impact the transactional performance of Azure Cosmos DB by consuming Request Units.
- Dedicated SQL pool integration through external tables is currently not supported.

Supported code-generated actions for Spark

GESTURE	DESCRIPTION	OLTP	HTAP
Load to DataFrame	Load and read data into a Spark DataFrame	✓	✓
Create Spark table	Create a table pointing to an Azure Cosmos DB container	✓	✓
Write DataFrame to container	Write data into a container	✓	✓
Load streaming DataFrame from container	Stream data using Azure Cosmos DB change feed	✓	✓
Write streaming DataFrame to container	Stream data using Azure Cosmos DB change feed	✓	✓

Supported code-generated actions for serverless SQL pool

GESTURE	DESCRIPTION	OLTP	HTAP
Explore data	Explore data from a container with familiar T-SQL syntax and automatic schema inference	X	✓
Create views and build BI reports	Create a SQL view to have direct access to a container for BI through serverless SQL pool	X	✓
Join disparate data sources along with Cosmos DB data	Store results of query reading data from Cosmos DB containers along with data in Azure Blob Storage or Azure Data Lake Storage using CETAS	X	✓

Next steps

- See how to connect to Synapse Link for Azure Cosmos DB
- Learn how to query the Cosmos DB Analytical Store with Spark

Azure Synapse Link for Azure Cosmos DB: Near real-time analytics use cases

12/4/2020 • 8 minutes to read • Edit Online

APPLIES TO: SQL API Azure Cosmos DB API for MongoDB

Azure Synapse Link for Azure Cosmos DB is a cloud native hybrid transactional and analytical processing (HTAP) capability that enables you to run near real-time analytics over operational data. Synapse Link creates a tight seamless integration between Azure Cosmos DB and Azure Synapse Analytics.

You might be curious to understand what industry use cases can leverage this cloud native HTAP capability for near real-time analytics over operational data. Here are three common use cases for Azure Synapse Link for Azure Cosmos DB:

- Supply chain analytics, forecasting & reporting
- Real-time personalization
- Predictive maintenance, anomaly detection in IOT scenarios

> **NOTE**
> Azure Synapse Link for Azure Cosmos DB targets the scenario where enterprise teams are looking to run near real-time analytics. These analytics are run without ETL over operational data generated across transactional applications built on Azure Cosmos DB. This does not replace the need for a separate data warehouse when there are traditional data warehouse requirements such as workload management, high concurrency, persistence aggregates across multiple data sources.

Supply chain analytics, forecasting & reporting

Research studies show that embedding big data analytics in supply chain operations leads to improvements in order-to-cycle delivery times and supply chain efficiency.

Manufacturers are onboarding to cloud-native technologies to break out of constraints of legacy Enterprise Resource Planning (ERP) and Supply Chain Management (SCM) systems. With supply chains generating increasing volumes of operational data every minute (order, shipment, transaction data), manufacturers need an operational database. This operational database should scale to handle the data volumes as well as an analytical platform to get to a level of real-time contextual intelligence to stay ahead of the curve.

The following architecture shows the power of leveraging Azure Cosmos DB as the cloud-native operational database and Synapse Link in supply chain analytics:

Based on previous architecture, you can achieve the following use cases with Synapse Link for Azure Cosmos DB:

- **Prepare & train predictive pipeline:** Generate insights over the operational data across the supply chain using machine learning translates. This way you can lower inventory, operations costs, and reduce the order-to-delivery times for customers.

  Synapse Link allows you to analyze the changing operational data in Azure Cosmos DB without any manual ETL processes. It saves you from additional cost, latency, and operational complexity. Synapse Link enables data engineers and data scientists to build robust predictive pipelines:

  - Query operational data from Azure Cosmos DB analytical store by leveraging native integration with Apache Spark pools in Azure Synapse Analytics. You can query the data in an interactive notebook or scheduled remote jobs without complex data engineering.
  - Build Machine Learning (ML) models with Spark ML algorithms and Azure ML integration in Azure Synapse Analytics.
  - Write back the results after model inference into Azure Cosmos DB for operational near-real-time scoring.

- **Operational reporting:** Supply chain teams need flexible and custom reports over real-time, accurate operational data. These reports are required to obtain a snapshot view of supply chain effectiveness, profitability, and productivity. It allows data analysts and other key stakeholders to constantly reevaluate the business and identify areas to tweak to reduce operational costs.

  Synapse Link for Azure Cosmos DB enables rich business intelligence (BI)/reporting scenarios:

  - Query operational data from Azure Cosmos DB analytical store by using native integration with serverless SQL pool and full expressiveness of T-SQL language.
  - Model and publish auto refreshing BI dashboards over Azure Cosmos DB through serverless SQL pool support for familiar BI tools. For example, Azure Analysis Services, Power BI Premium, etc.

The following is some guidance for data integration for batch & streaming data into Azure Cosmos DB:

- **Batch data integration & orchestration:** With supply chains getting more complex, supply chain data platforms need to integrate with variety of data sources and formats. Azure Synapse comes built-in with the same data integration engine and experiences as Azure Data Factory. This integration allows data engineers to create rich data pipelines without a separate orchestration engine:

- Move data from 85+ supported data sources to Azure Cosmos DB with Azure Data Factory.
- Write code-free ETL pipelines to Azure Cosmos DB including relational-to-hierarchical and hierarchical-to-hierarchical mappings with mapping data flows.

- **Streaming data integration & processing:** With the growth of Industrial IoT (sensors tracking assets from 'floor-to-store', connected logistics fleets, etc.), there is an explosion of real-time data being generated in a streaming fashion that needs to be integrated with traditional slow moving data for generating insights. Azure Stream Analytics is a recommended service for streaming ETL and processing on Azure with a wide range of scenarios. Azure Stream Analytics supports Azure Cosmos DB as a native data sink.

Real-time personalization

Retailers today must build secure and scalable e-commerce solutions that meet the demands of both customers and business. These e-commerce solutions need to engage customers through customized products and offers, process transactions quickly and securely, and focus on fulfillment and customer service. Azure Cosmos DB along with the latest Synapse Link for Azure Cosmos DB allows retailers to generate personalized recommendations for customers in real time. They use low-latency and tunable consistency settings for immediate insights as shown in the following architecture:

Synapse Link for Azure Cosmos DB use case:

- **Prepare & train predictive pipeline:** You can generate insights over the operational data across your business units or customer segments using Synapse Spark and machine learning models. This translates to personalized delivery to target customer segments, predictive end-user experiences and targeted marketing to fit your end-user requirements.

IOT predictive maintenance

Industrial IOT innovations have drastically reduced downtimes of machinery and increased overall efficiency across all fields of industry. One of such innovations is predictive maintenance analytics for machinery at the edge of the cloud.

The following is an architecture leveraging the cloud native HTAP capabilities of Azure Synapse Link for Azure Cosmos DB in IoT predictive maintenance:

Synapse Link for Azure Cosmos DB use cases:

- **Prepare & train predictive pipeline:** The historical operational data from IoT device sensors could be used to train predictive models such as anomaly detectors. These anomaly detectors are then deployed back to the edge for real-time monitoring. Such a virtuous loop allows for continuous retraining of the predictive models.
- **Operational reporting:** With the growth of digital twin initiatives, companies are collecting vast amounts of operational data from large number of sensors to build a digital copy of each machine. This data powers BI needs to understand trends over historical data in addition to real-time applications over recent hot data.

Sample scenario: HTAP for Azure Cosmos DB

For nearly a decade, Azure Cosmos DB has been used by thousands of customers for mission critical applications that require elastic scale, turnkey global distribution, multi-region write replication for low latency and high availability of both reads & writes in their transactional workloads.

The following list shows an overview of the various workload patterns that are supported with operational data using Azure Cosmos DB:

- Real-time apps & services
- Event stream processing
- BI dashboards
- Big data analytics
- Machine learning

Azure Synapse Link enables Azure Cosmos DB to not just power transactional workloads but also perform near real-time analytical workloads over historical operational data. It happens with no ETL requirements and guaranteed performance isolation from the transactional workloads.

The following image shows workload patterns using Azure Cosmos DB:

Let us take the example of an e-commerce company CompanyXYZ with global operations across 20 countries/regions to illustrate the benefits of choosing Azure Cosmos DB as the single real-time database powering both transactional and analytical requirements of an inventory management platform.

- CompanyXYZ's core business depends on the inventory management system – hence availability & reliability are core pillar requirements. Benefits of using Azure Cosmos DB:
  - By virtue of deep integration with Azure infrastructure, and transparent multi-region writes, global replication, Azure Cosmos DB provides industry-leading 99.999% high availability against regional outages.
- CompanyXYZ's supply chain partners may be in separate geographic locations but they may have to see a single view of the product inventory across the globe to support their local operations. This includes the need to be able to read updates made by other supply chain partners in real time. As well as being able to make updates without worrying about conflicts with other partners at high throughput. Benefits of using Azure Cosmos DB:
  - With its unique multi-region writes replication protocol and latch-free, write-optimized transactional store, Azure Cosmos DB guarantees less than 10-ms latencies for both indexed reads and writes at the 99th percentile globally.
  - High throughput ingestion of both batch & streaming data feeds with real-time indexing in transactional store.
  - Azure Cosmos DB transactional store provides three more options than the two extremes of strong and eventual consistency levels to achieve the availability vs performance tradeoffs closest to the business need.
- CompanyXYZ's supply chain partners have highly fluctuating traffic patterns ranging from hundreds to millions of requests/s and thus the inventory management platform needs to deal with unexpected burstiness in traffic. Benefits of using Azure Cosmos DB:
  - Azure Cosmos DB's transactional store supports elastic scalability of storage and throughput using horizontal partitioning. Containers and databases configured in Autopilot mode can automatically and instantly scale the provisioned throughput based on the application needs without impacting the availability, latency, throughput, or performance of the workload globally.
- CompanyXYZ needs to establish a secure analytics platform to house system-wide historical inventory data to enable analytics and insights across supply chain partner, business units and functions. The analytics platform needs to enable collaboration across the system, traditional BI/reporting use cases, advanced analytics use cases and predictive intelligent solutions over the operational inventory data. Benefits of using

Synapse Link for Azure Cosmos DB:

- By using Azure Cosmos DB analytical store, a fully isolated column store, Synapse Link enables no Extract-Transform-Load (ETL) analytics in Azure Synapse Analytics against globally distributed operational data at scale. Business analysts, data engineers and data scientists can now use Synapse Spark or Synapse SQL in an interoperable manner to run near real-time business intelligence, analytics, and machine learning pipelines without impacting the performance of their transactional workloads on Azure Cosmos DB. See the benefits of Synapse Link in Azure Cosmos DB for more details.

Next steps

To learn more, see the following docs:

- Azure Synapse Link for Azure Cosmos DB
- Azure Cosmos DB Analytical Store
- Working with Azure Synapse Link for Azure Cosmos DB
- Frequently asked questions about Azure Synapse Link for Azure Cosmos DB
- Apache Spark in Azure Synapse Analytics
- Serverless SQL pool runtime support in Azure Synapse Analytics

Frequently asked questions about Azure Synapse Link for Azure Cosmos DB

12/4/2020 • 8 minutes to read • Edit Online

APPLIES TO: SQL API Azure Cosmos DB API for MongoDB

Azure Synapse Link for Azure Cosmos DB creates a tight integration between Azure Cosmos DB and Azure Synapse Analytics. It enables customers to run near real-time analytics over their operational data with full performance isolation from their transactional workloads and without an ETL pipeline. This article answers commonly asked questions about Synapse Link for Azure Cosmos DB.

General FAQ

Is Azure Synapse Link supported for all Azure Cosmos DB APIs?

Azure Synapse Link is supported for the Azure Cosmos DB SQL (Core) API and for the Azure Cosmos DB API for MongoDB.

Is Azure Synapse Link supported for multi-region Azure Cosmos DB accounts?

Yes, for multi-region Azure Cosmos accounts, the data stored in the analytical store is also globally distributed. Regardless of single write region or multiple write regions, analytical queries performed from Azure Synapse Analytics can be served from the closest local region.

When planning to configure a multi-region Azure Cosmos DB account with analytical store support, it is recommended to have all the necessary regions added at time of account creation.

Can I choose to enable Azure Synapse Link for only certain region and not all regions in a multi-region account set-up?

When Azure Synapse Link is enabled for a multi-region account, the analytical store is created in all regions. The underlying data is optimized for throughput and transactional consistency in the transactional store.

Is backup and restore supported for Azure Synapse Link enabled accounts?

For the containers with analytical store turned on, automatic backup and restore of your data in the analytical store is not supported at this time.

When Synapse Link is enabled on a database account, Azure Cosmos DB will continue to automatically take backups of your data in the transactional store (only) of containers at scheduled backup interval, as always. It is important to note that when a container with analytical store turned on is restored to a new account, the container will be restored with only transactional store and no analytical store enabled.

Can I disable the Azure Synapse Link feature for my Azure Cosmos DB account?

Currently, after the Synapse Link capability is enabled at the account level, you cannot disable it. Understand that you will not have any billing implications if the Synapse Link capability is enabled at the account level and there is no analytical store enabled containers.

If you need to turn off the capability, you have 2 options. The first one is to delete and re-create a new Azure Cosmos DB account, migrating the data if necessary. The second option is to open a support ticket, to get help on a data migration to another account.

Does analytical store have any impact on Cosmos DB transactional SLAs?

No, there is no impact.

Azure Cosmos DB analytical store

Can I enable analytical store on existing containers?

Currently, the analytical store can only be enabled for new containers (both in new and existing accounts).

Can I disable analytical store on my Azure Cosmos DB containers after enabling it during container creation?

Currently, the analytical store cannot be disabled on an Azure Cosmos DB container after it is enabled during container creation.

Is analytical store supported for Azure Cosmos DB containers with autoscale provisioned throughput?

Yes, the analytical store can be enabled on containers with autoscale provisioned throughput.

Is there any effect on Azure Cosmos DB transactional store provisioned RUs?

Azure Cosmos DB guarantees performance isolation between the transactional and analytical workloads. Enabling the analytical store on a container will not impact the RU/s provisioned on the Azure Cosmos DB transactional store. The transactions (read & write) and storage costs for the analytical store will be charged separately. See the pricing for Azure Cosmos DB analytical store for more details.

Are delete and update operations on the transactional store reflected in the analytical store?

Yes, deletes and updates to the data in the transactional store will be reflected in the analytical store. You can configure the Time to Live (TTL) on the container to include historical data so that the analytical store retains all versions of items that satisfy the analytical TTL criteria. See the overview of analytical TTL for more details.

Can I connect to analytical store from analytics engines other than Azure Synapse Analytics?

You can only access and run queries against the analytical store using the various run-times provided by Azure Synapse Analytics. The analytical store can be queried and analyzed using:

- Synapse Spark with full support for Scala, Python, SparkSQL, and C#. Synapse Spark is central to data engineering and science scenarios
- Serverless SQL pool with T-SQL language and support for familiar BI tools (For example, Power BI Premium, etc.)

Can I connect to analytical store from Synapse SQL provisioned?

At this time, the analytical store cannot be accessed from Synapse SQL provisioned.

Can I write back the query aggregation results from Synapse back to the analytical store?

Analytical store is a read-only store in an Azure Cosmos DB container. So, you cannot directly write back the aggregation results to the analytical store, but can write them to the Azure Cosmos DB transactional store of another container, which can later be leveraged as a serving layer.

Is the autosync replication from transactional store to the analytical store asynchronous or synchronous and what are the latencies?

Auto-sync latency is usually within 2 minutes. In cases of shared throughput database with a large number of containers, auto-sync latency of individual containers could be higher and take up to 5 minutes. We would like to learn more how this latency fits your scenarios. For that, please reach out to the Azure Cosmos DB team.

Are there any scenarios where the items from the transactional store are not automatically propagated to the analytical store?

If specific items in your container violate the well-defined schema for analytics, they will not be included in the analytical store. If you have scenarios blocked by well-defined schema for analytics, email the Azure Cosmos DB team for help.

Can I partition the data in analytical store differently from transactional store?

The data in analytical store is partitioned based on the horizontal partitioning of shards in the transactional store. Currently, you cannot choose a different partitioning strategy for the analytical store.

Can I customize or override the way transactional data is transformed into columnar format in the analytical store?

Currently you can't transform the data items when they are automatically propagated from the transactional store to analytical store. If you have scenarios blocked by this limitation, email the Azure Cosmos DB team.

Is analytical store supported by Terraform?

Currently Terraform doesn't support analytical store containers. Please check Terraform GitHub Issues for more information.

Analytical Time to live (TTL)

Is TTl for analytical data supported at both container and item level?

At this time, TTl for analytical data can only be configured at container level and there is no support to set analytical TTL at item level.

After setting the container level analytical TTL on an Azure Cosmos DB container, can I change to a different value later?

Yes, analytical TTL can be updated to any valid value. See the Analytical TTL article for more details about analytical TTL.

Can I update or delete an item from the analytical store after it has been TTL'd out from the transactional store?

All transactional updates and deletes are copied to the analytical store but if the item has been purged from the transactional store, then it cannot be updated in the analytical store. To learn more, see the Analytical TTL article.

Billing

What is the billing model of Azure Synapse Link for Azure Cosmos DB?

The billing model of Azure Synapse Link includes the costs incurred by using the Azure Cosmos DB analytical store and the Synapse runtime. To learn more, see the Azure Cosmos DB analytical store pricing and Azure Synapse Analytics pricing articles.

What is the billing impact if I enable Synapse Link in my Azure Cosmos DB database account?

None. You will only be charged when you create an analytical store enabled container and start to load data.

Security

What are the ways to authenticate with the analytical store?

Authentication with the analytical store is the same as a transactional store. For a given database, you can authenticate with the primary or read-only key. You can leverage linked service in Azure Synapse Studio to prevent pasting the Azure Cosmos DB keys in the Spark notebooks. Access to this Linked Service is available for everyone who has access to the workspace.

Synapse run-times

What are the currently supported Synapse run-times to access Azure Cosmos DB analytical store?

AZURE SYNAPSE RUNTIME	CURRENT SUPPORT
Azure Synapse Spark pools	Read, Write (through transactional store), Table, Temporary View
Azure Synapse serverless SQL pool	Read, View

AZURE SYNAPSE RUNTIME	CURRENT SUPPORT
Azure Synapse SQL Provisioned	Not available

Do my Azure Synapse Spark tables sync with my Azure Synapse serverless SQL pool tables the same way they do with Azure Data Lake?

Currently, this feature is not available.

Can I do Spark structured streaming from analytical store?

Currently Spark structured streaming support for Azure Cosmos DB is implemented using the change feed functionality of the transactional store and it's not yet supported from analytical store.

Azure Synapse Studio

In the Azure Synapse Studio, how do I recognize if I'm connected to an Azure Cosmos DB container with the analytics store enabled?

An Azure Cosmos DB container enabled with analytical store has the following icon:

A transactional store container will be represented with the following icon:

How do you pass Azure Cosmos DB credentials from Azure Synapse Studio?

Currently Azure Cosmos DB credentials are passed while creating the linked service by the user who has access to the Azure Cosmos DB databases. Access to that store is available to other users who have access to the workspace.

Next steps

- Learn about the benefits of Azure Synapse Link
- Learn about the integration between Azure Synapse Link and Azure Cosmos DB.

Plan and manage costs for Azure Cosmos DB

12/4/2020 • 7 minutes to read • Edit Online

APPLIES TO: SQL API Cassandra API Gremlin API Table API Azure Cosmos DB API for MongoDB

This article describes how you can plan and manage costs for Azure Cosmos DB. First, you use the Azure Cosmos DB capacity calculator to estimate your workload cost before you create any resources. Later you can review the estimated cost and start creating your resources.

After you've started using Azure Cosmos DB resources, use the Cost Management features to set budgets and monitor costs. You can also review forecasted costs and identify spending trends to identify areas where you might want to act. The costs for Azure Cosmos DB are only a portion of the monthly costs in your Azure bill. Although this article explains how to plan for and manage costs for Azure Cosmos DB, you're billed for all Azure services and resources used in your Azure subscription, including the third-party services.

Prerequisites

Provisioned throughput or serverless

Azure Cosmos DB supports two types of capacity modes: provisioned throughput and serverless. The way you get charged for your Azure Cosmos DB usage varies a lot between these two modes, so it's important to choose the one that works best for your workload. See the how to choose between provisioned throughput and serverless article for guidance and recommendations on how to make this choice.

Cost analysis

Cost analysis in Cost Management supports most Azure account types, but not all of them. To view the full list of supported account types, see Understand Cost Management data. To view cost data, you need at least read access for an Azure account. For information about assigning access to Azure Cost Management data, see Assign access to data.

Estimating provisioned throughput costs before using Azure Cosmos DB

If you plan to use Azure Cosmos DB in provisioned throughput mode, use the Azure Cosmos DB capacity calculator to estimate costs before you create the resources in an Azure Cosmos account. The capacity calculator is used to get an estimate of the required throughput and cost of your workload. Configuring your Azure Cosmos databases and containers with the right amount of provisioned throughput, or Request Units (RU/s), for your workload is essential to optimize the cost and performance. You have to input details such as API type, number of regions, item size, read/write requests per second, total data stored to get a cost estimate. To learn more about the capacity calculator, see the estimate article.

The following screenshot shows the throughput and cost estimation by using the capacity calculator:

Estimating serverless costs before using Azure Cosmos DB

If you plan to use Azure Cosmos DB in serverless mode, you need to estimate how many Request Units and GB of storage you may consume on a monthly basis. You can estimate the required amount of Request Units by evaluating the number of database operations that would be issued in a month, and multiply their amount by their corresponding RU cost. The following table lists estimated RU charges for common database operations:

OPERATION	ESTIMATED COST	NOTES
Create an item	5 RUs	Average cost for a 1 KB item with less than 5 properties to index
Update an item	10 RUs	Average cost for a 1 KB item with less than 5 properties to index
Read an individual item by its ID and partition key (point-read)	1 RU	Average cost for a 1 KB item
Delete an item	5 RUs	
Execute a query	10 RUs	Average cost for a query that takes full advantage of indexing and returns 100 results or less

> **IMPORTANT**
>
> Pay attention to the Notes from the table above. For a more accurate estimation of the actual costs of your operations, you can use the Azure Cosmos DB Emulator and measure the exact RU cost of your operations. Although the Azure Cosmos DB Emulator doesn't support serverless, it reports a standard RU charge for database operations and can be used for this estimation.

Once you have computed the total number of Request Units and GB of storage you're likely to consume over a month, the following formula will return your cost estimate: **([Number of Request units] / 1,000,000 * $0.25) + ([GB of storage] * $0.25)**.

> **NOTE**
>
> The costs shown in the previous example are for demonstration purposes only. See the pricing page for the latest pricing information.

Review estimated costs in the Azure portal

As you start using Azure Cosmos DB resources from Azure portal, you can see the estimated costs. Use the following steps to review the cost estimate:

1. Sign into the Azure portal and navigate to your Azure Cosmos account.
2. Go to the **Overview** section.
3. Check the **Cost** chart at the bottom. This chart shows an estimate of your current cost over a configurable time period:
4. Create a new container such as a graph container.
5. Input the throughput required for your workload such as 400 RU/s. After you input the throughput value, you can see the pricing estimate as shown in the following screenshot:

If your Azure subscription has a spending limit, Azure prevents you from spending over your credit amount. As you

create and use Azure resources, your credits are used. When you reach your credit limit, the resources that you deployed are disabled for the rest of that billing period. You can't change your credit limit, but you can remove it. For more information about spending limits, see Azure spending limit.

You can pay for Azure Cosmos DB charges with your Azure Enterprise Agreement monetary commitment credit. However, you can't use the monetary commitment credit to pay for charges for third party products and services including those from the Azure Marketplace.

Monitor costs

As you use resources with Azure Cosmos DB, you incur costs. Resource usage unit costs vary by time intervals (seconds, minutes, hours, and days) or by request unit usage. As soon as usage of Azure Cosmos DB starts, costs are incurred and you can see them in the cost analysis pane in the Azure portal.

When you use cost analysis, you can view the Azure Cosmos DB costs in graphs and tables for different time intervals. Some examples are by day, current, prior month, and year. You can also view costs against budgets and forecasted costs. Switching to longer views over time can help you identify spending trends and see where overspending might have occurred. If you’ve created budgets, you can also easily see where they exceeded.

To view Azure Cosmos DB costs in cost analysis:

1. Sign into the Azure portal.
2. Open the scope in the Azure portal and select **Cost analysis** in the menu. For example, go to **Subscriptions**, select a subscription from the list, and then select **Cost analysis** in the menu. Select **Scope** to switch to a different scope in cost analysis.
3. By default, cost for all services are shown in the first donut chart. Select the area in the chart labeled "Azure Cosmos DB".
4. To narrow costs for a single service such as Azure Cosmos DB, select **Add filter** and then select **Service name**. Then, choose **Azure Cosmos DB** from the list. Here’s an example showing costs for just Azure Cosmos DB:

In the preceding example, you see the current cost for Azure Cosmos DB for the month of Feb. The charts also contain Azure Cosmos DB costs by location and by resource group.

Create budgets

You can create budgets to manage costs and create alerts that automatically notify stakeholders of spending anomalies and overspending risks. Alerts are based on spending compared to budget and cost thresholds. Budgets and alerts are created for Azure subscriptions and resource groups, so they're useful as part of an overall cost monitoring strategy.

Budgets can be created with filters for specific resources or services in Azure if you want more granularity present in your monitoring. Filters help ensure that you don't accidentally create new resources that cost you additional money. For more about the filter options when you when create a budget, see Group and filter options.

Export cost data

You can also export your cost data to a storage account. This is helpful when you need or others to do additional data analysis for costs. For example, a finance teams can analyze the data using Excel or Power BI. You can export your costs on a daily, weekly, or monthly schedule and set a custom date range. Exporting cost data is the recommended way to retrieve cost datasets.

Next steps

See the following articles to learn more on how pricing works in Azure Cosmos DB:

- Pricing model in Azure Cosmos DB
- Optimize provisioned throughput cost in Azure Cosmos DB
- Optimize query cost in Azure Cosmos DB
- Optimize storage cost in Azure Cosmos DB

- Learn how to optimize your cloud investment with Azure Cost Management.
- Learn more about managing costs with cost analysis.
- Learn about how to prevent unexpected costs.
- Take the Cost Management guided learning course.

Pricing model in Azure Cosmos DB

11/2/2020 • 5 minutes to read • Edit Online

APPLIES TO: SQL API Cassandra API Gremlin API Table API Azure Cosmos DB API for MongoDB

The pricing model of Azure Cosmos DB simplifies the cost management and planning. With Azure Cosmos DB, you pay for the operations you perform against the database and for the storage consumed by your data.

- **Database operations**: The way you get charged for your database operations depends on the type of Azure Cosmos account you are using.
  - **Provisioned Throughput**: Provisioned throughput (also called reserved throughput) provides high performance at any scale. You specify the throughput that you need in Request Units per second (RU/s), and Azure Cosmos DB dedicates the resources required to provide the configured throughput. You can provision throughput on either a database or a container. Based on your workload needs, you can scale throughput up/down at any time or use autoscale (although there is a minimum throughput required on a database or a container to guarantee the SLAs). You are billed hourly for the maximum provisioned throughput for a given hour.

> **NOTE**
> Because the provisioned throughput model dedicates resources to your container or database, you will be charged for the throughput you have provisioned even if you don't run any workloads.

  - **Serverless**: In serverless mode, you don't have to provision any throughput when creating resources in your Azure Cosmos account. At the end of your billing period, you get billed for the amount of Request Units that has been consumed by your database operations.
- **Storage**: You are billed a flat rate for the total amount of storage (in GBs) consumed by your data and indexes for a given hour. Storage is billed on a consumption basis, so you don't have to reserve any storage in advance. You are billed only for the storage you consume.

The pricing model in Azure Cosmos DB is consistent across all APIs. For more information, see the Azure Cosmos DB pricing page, Understanding your Azure Cosmos DB bill and How Azure Cosmos DB pricing model is cost-effective for customers.

If you deploy your Azure Cosmos DB account to a non-government region in the US, there is a minimum price for both database and container-based throughput in provisioned throughput mode. There is no minimum price in serverless mode. The pricing varies depending on the region you are using, see the Azure Cosmos DB pricing page for latest pricing information.

Try Azure Cosmos DB for free

Azure Cosmos DB offers many options for developers to it for free. These options include:

- **Azure Cosmos DB free tier**: Azure Cosmos DB free tier makes it easy to get started, develop and test your applications, or even run small production workloads for free. When free tier is enabled on an account, you'll get the first 400 RU/s and 5 GB of storage in the account free, for the lifetime of the account. You can have up to one free tier account per Azure subscription and must opt-in when creating the account. To get started, create a new account in Azure portal with free tier enabled or use an ARM Template.
- **Azure free account**: Azure offers a free tier that gives you $200 in Azure credits for the first 30 days and a

limited quantity of free services for 12 months. For more information, see Azure free account. Azure Cosmos DB is a part of Azure free account. Specifically for Azure Cosmos DB, this free account offers 5-GB storage and 400 RU/s of provisioned throughput for the entire year.

- **Try Azure Cosmos DB for free**: Azure Cosmos DB offers a time-limited experience by using try Azure Cosmos DB for free accounts. You can create an Azure Cosmos DB account, create database and collections and run a sample application by using the Quickstarts and tutorials. You can run the sample application without subscribing to an Azure account or using your credit card. Try Azure Cosmos DB for free offers Azure Cosmos DB for one month, with the ability to renew your account any number of times.
- **Azure Cosmos DB emulator**: Azure Cosmos DB emulator provides a local environment that emulates the Azure Cosmos DB service for development purposes. Emulator is offered at no cost and with high fidelity to the cloud service. Using Azure Cosmos DB emulator, you can develop and test your applications locally, without creating an Azure subscription or incurring any costs. You can develop your applications by using the emulator locally before going into production. After you are satisfied with the functionality of the application against the emulator, you can switch to using the Azure Cosmos DB account in the cloud and significantly save on cost. For more information about emulator, see Using Azure Cosmos DB for development and testing article for more details.

Pricing with reserved capacity

Azure Cosmos DB reserved capacity helps you save money when using the provisioned throughput mode by pre-paying for Azure Cosmos DB resources for either one year or three years. You can significantly reduce your costs with one-year or three-year upfront commitments and save between 20-65% discounts when compared to the regular pricing. Azure Cosmos DB reserved capacity helps you lower costs by pre-paying for the provisioned throughput (RU/s) for a period of one year or three years and you get a discount on the throughput provisioned.

Reserved capacity provides a billing discount and does not affect the runtime state of your Azure Cosmos DB resources. Reserved capacity is available consistently to all APIs, which includes MongoDB, Cassandra, SQL, Gremlin, and Azure Tables and all regions worldwide. You can learn more about reserved capacity in Prepay for Azure Cosmos DB resources with reserved capacity article and buy reserved capacity from the Azure portal.

Next steps

You can learn more about optimizing the costs for your Azure Cosmos DB resources in the following articles:

- Learn about Optimizing for development and testing
- Learn more about Understanding your Azure Cosmos DB bill
- Learn more about Optimizing throughput cost
- Learn more about Optimizing storage cost
- Learn more about Optimizing the cost of reads and writes
- Learn more about Optimizing the cost of queries
- Learn more about Optimizing the cost of multi-region Cosmos accounts
- Learn about Azure Cosmos DB reserved capacity
- Learn about Azure Cosmos DB Emulator

Total Cost of Ownership (TCO) with Azure Cosmos DB

11/2/2020 • 7 minutes to read • Edit Online

APPLIES TO: SQL API Cassandra API Gremlin API Table API Azure Cosmos DB API for MongoDB

Azure Cosmos DB is designed with the fine grained multi-tenancy and resource governance. This design allows Azure Cosmos DB to operate at significantly lower cost and help users save. Currently Azure Cosmos DB supports more than 280 customer workloads on a single machine with the density continuously increasing, and thousands of customer workloads within a cluster. It load balances replicas of customers' workloads across different machines in a cluster and across multiple clusters within a data center. To learn more, see Azure Cosmos DB: Pushing the frontier of globally distributed databases. Because of resource-governance, multi-tenancy, and native integration with the rest of Azure infrastructure, Azure Cosmos DB is on average 4 to 6 times cheaper than MongoDB, Cassandra, or other OSS NoSQL running on IaaS and up to 10 times cheaper than the database engines running on premises. See the paper on The total cost of (non) ownership of a NoSQL database cloud service.

The OSS NoSQL database solutions, such as Apache Cassandra, MongoDB, HBase, engines were designed for on-premises. When offered as a managed service they are equivalent to a Resource Manager template with a tenant database for managing the provisioned clusters and monitoring support. OSS NoSQL architectures require significant operational overhead, and the expertise can be difficult and expensive to find. On the other hand, Azure Cosmos DB is a fully managed cloud service, which allows developers to focus on business innovation rather than on managing and maintaining database infrastructure.

Unlike a cloud-native database service Azure Cosmos DB, OSS NoSQL database engines were not designed and built with the resource governance or fine-grained multi-tenancy as the fundamental architectural principles. OSS NoSQL database engines like Cassandra and MongoDB make a fundamental assumption that all the resources of the virtual machine on which they are running are available for their use. Many of these database engines cannot function if the amount of resources drops below a certain threshold. For example, for small VM instances, and they are available with vendor-recommended configurations suggesting typically large-scale VMs with higher cost. So, it is not possible to host an OSS NoSQL or any other on-premises database engine and make it available by using a consumption-based charging model like requests per second or consumed storage.

Total cost of ownership of Azure Cosmos DB

The serverless provisioning model of Azure Cosmos DB eliminates the need to over-provision the database infrastructure. Azure Cosmos DB resources are provided without any need for specialized configurations or licensing. As a result, the Azure Cosmos DB-backed applications can run with as much as a 70 percent Total cost of ownership savings when compared to OSS NoSQL databases. For some real-time examples, see customer use-cases. Other benefits of the Azure Cosmos DB pricing model include:

- **Great value for the price:** Market analysts, customers, and partners have confirmed a greater value of all the features that Azure Cosmos DB offers for a much lower price compared to what customers can get when implementing these solutions on their own or through other vendors. The database features such global distribution, multi-region writes, well-defined and intuitive consistency models, automatic indexing are greatly simplified with Azure Cosmos DB without any complexity, overhead, or downtime.
- **No NoSQL DevOps administration is required:** With Azure Cosmos DB one does not need to employ DevOps to manage deployments, perform maintenance, scale, or patch. You can execute the all the workloads that you would do with OSS NoSQL cluster hosted on-premises or on cloud infrastructure.

- **Ability to elastically scale:** Azure Cosmos DB throughput can be scaled up and down, allowing you to reduce the cost of ownership during non-peak times. OSS NoSQL clusters deployed on cloud infrastructure offer limited elasticity, and on-premises deployments aren't elastic by definition. In Azure Cosmos DB, if you provision more throughput, your throughput is guaranteed to scale linearly. This guarantee is backed up by financial SLAs and at the 99th percentile at any scale.

- **Economies of scale:** A managed service like Azure Cosmos DB operates with a large number of nodes, integrated natively with networking, storage, and computes. Because of Azure Cosmos DB's large scale, standardization you can save costs.

- **Optimized for the cloud:** Azure Cosmos DB is designed from the ground-up with fine-grained multi-tenancy and performance isolation. This allows for optimally placing, executing, and balancing thousands of tenants and their workloads across clusters and data centers. In contrast, the current generation of OSS NoSQL databases operate on-premises with the entire virtual machine assumed to run a single tenant's workload. These databases are also not designed to leverage a cloud provider's infrastructure and hardware to the full extent. For example, an OSS NoSQL database engine isn't aware of the differences between a virtual machine being down Vs a routine image upgrade, or the fact that premium disk is already three-way replicated. It can't take advantage of these benefits and pass on the benefits and savings to customers.

- **You pay by the hour:** For large-scale workloads, that need to scale at any point in time, you are only charged by the hour. The workloads on an application typically vary across times of the year, and by the data that is queried. With Azure Cosmos DB, you can scale up or down as you need and pay only for what you need. With on-premises or IaaS-hosted systems, you can't match this model, because there isn't a way to decommission the hardware every hour. In such cases, you can potentially save between 10 to 14 times on an average with Azure Cosmos DB.

- **You get numerous features for free:** In Azure Cosmos DB, write workloads are substantially cheaper compared to alternative database services. In addition, Azure Cosmos DB offers features such as such as automatic indexing, Time to Live (TTL), Change Feed and others without any additional charges, something that other database services typically charge.

- **Uses unified currency for diverse workloads:** Unlike alternative offerings, in Azure Cosmos DB, you do not need to segment workloads, for example, into reads and writes. Or provision throughput on a per workload type that is read throughput vs. write throughput. In Azure Cosmos DB, provisioned throughput is reserved using a unified and normalized currency in terms of Request Units or RU/sec. Azure Cosmos DB

doesn't force you to assign priority to your workloads, perform capacity planning or pay for each type of capacity separately. Such approach enables you to easily interchange the same RU/s between various operations and workload types.

- **Doesn't require provisioning VMs to scale:** Most operational databases require you to go with large virtual machines to avoid noisy neighbors and for loose resource governance, if you want scale. This puts the burden and the upfront commitment of cost on the customers. With Azure Cosmos DB, you can start small and grow into the large scale workload sizes seamlessly, and without any downtime or impact on data availability.
- **You can utilize provisioned throughput to a maximum limit:** By the virtue of sub-core multiplexing in Azure Cosmos DB, you can saturate the provisioned throughput to a greater extent than IaaS hosted options or third party offers. This method saves a lot more than the alternative solutions.
- **Deep integration of Azure Cosmos DB with other Azure services.** Azure Cosmos DB has a native integration with Networking, Compute, Azure Functions (serverless), Azure IoT, and others Azure services. With this integration, you get the best performance, speed of data replication across the world with robust guarantees. The third party solutions won't be able to match or would typically charge a premium to offer such features.
- **You automatically get high availability, with at least 10-20 fault domains by default:** Azure Cosmos DB supports the distribution of workloads across fault domains, a feature that is critical for high availability. It offers 99.999 high availability for reads and writes at the 99th percentile across anywhere in the world. The cost of implementing something like this on your own or through a third-party solution, would be high.
- **You automatically get all enterprise capabilities, at no additional cost.** Azure Cosmos DB offers the most comprehensive set of compliance certifications, security, and encryption at rest and in motion at no additional cost (compared to our competition). You automatically get regional availability anywhere in the world. You can span your database across any number of Azure regions and add or remove regions at any point.
- **You can save up to 65% of costs with reserved capacity:** Azure Cosmos DB reserved capacity helps you save money by pre-paying for Azure Cosmos DB resources for either one year or three years. You can significantly reduce your costs with one-year or three-year upfront commitments and save between 20-65% discounts when compared to the regular pricing. On your mission-critical workloads you can get better SLAs in terms of provisioning capacity.

Next steps

- Learn more about How Azure Cosmos DB pricing model is cost-effective for customers
- Learn more about Optimizing for development and testing
- Learn more about Optimizing throughput cost
- Learn more about Optimizing storage cost
- Learn more about Optimizing the cost of reads and writes
- Learn more about Optimizing the cost of queries
- Learn more about Optimizing the cost of multi-region Cosmos accounts
- Learn more about The Total Cost of (Non) Ownership of a NoSQL Database Cloud Service

Understand your Azure Cosmos DB bill

12/4/2020 • 19 minutes to read • Edit Online

APPLIES TO: SQL API Cassandra API Gremlin API Table API Azure Cosmos DB API for MongoDB

As a fully managed cloud-native database service, Azure Cosmos DB simplifies billing by charging only for your database operations and consumed storage. There are no additional license fees, hardware, utility costs, or facility costs compared to on-premises or IaaS-hosted alternatives. When you consider the multi region capabilities of Azure Cosmos DB, the database service provides a substantial reduction in costs compared to existing on-premises or IaaS solutions.

- **Database operations**: The way you get charged for your database operations depends on the type of Azure Cosmos account you are using.
  - **Provisioned Throughput**: You are billed hourly for the maximum provisioned throughput for a given hour, in increments of 100 RU/s.
  - **Serverless**: You are billed hourly for the total amount of Request Units consumed by your database operations.
- **Storage**: You are billed a flat rate for the total amount of storage (in GBs) consumed by your data and indexes for a given hour.

See the pricing page for the latest pricing information.

This article uses some examples to help you understand the details you see on the monthly bill. The numbers shown in the examples may be different if your Azure Cosmos containers have a different amount of throughput provisioned, if they span across multiple regions or run for a different for a period over a month. All the examples in this article calculate the bill based on the pricing information shown in the Pricing page.

> **NOTE**
> Billing is for any portion of a wall-clock hour, not a 60 minute duration. All the examples shown in this doc are based on the price for an Azure Cosmos account deployed in a non-government region in the US. The pricing and calculation vary depending on the region you are using, see the Azure Cosmos DB pricing page for latest pricing information.

Billing examples

Billing example - provisioned throughput on a container (full month)

- Let's assume you configure a throughput of 1,000 RU/sec on a container, and it exists for 24 hours * 30 days for the month = 720 hours total.
- 1,000 RU/sec is 10 units of 100 RU/sec per hour for each hour the containers exists (that is, 1,000/100 = 10).
- Multiplying 10 units per hour by the cost of $0.008 (per 100 RU/sec per hour) = $0.08 per hour.
- Multiplying the $0.08 per hour by the number of hours in the month equals $0.08 * 24 hours * 30 days = $57.60 for the month.
- The total monthly bill will show 7,200 units (of 100 RUs), which will cost $57.60.

Billing example - provisioned throughput on a container (partial month)

- Let's assume we create a container with provisioned throughput of 2,500 RU/sec. The container lives for 24 hours over the month (for example, we delete it 24 hours after we create it).
- Then we'll see 600 units on the bill (2,500 RU/sec / 100 RU/sec/unit * 24 hours). The cost will be $4.80 (600 units * $0.008/unit).
- Total bill for the month will be $4.80.

Billing example - serverless container

- Let's assume we create a serverless container.
- Over a month, we issue database requests consuming a total of 500,000 Request Units. The cost will be $0.125 (500,000 * $0.25/million).
- Total bill for the month will be $0.125.

Billing rate if storage size changes

Storage capacity is billed in units of the maximum hourly amount of data stored, in GB, over a monthly period. For example, if you utilized 100 GB of storage for half of the month and 50 GB for the second half of the month, you would be billed for an equivalent of 75 GB of storage during that month.

Billing rate when container or a set of containers are active for less than an hour

You're billed the flat rate for each hour the container or database exists, no matter the usage or if the container or database is active for less than an hour. For example, if you create a container or database and delete it 5 minutes later, your bill will include one hour.

Billing rate when provisioned throughput on a container or database scales up/down

If you increase provisioned throughput at 9:30 AM from 400 RU/sec to 1,000 RU/sec and then lower provisioned throughput at 10:45 AM back to 400 RU/sec, you will be charged for two hours of 1,000 RU/sec.

If you increase provisioned throughput for a container or a set of containers at 9:30 AM from 100-K RU/sec to 200-K RU/sec and then lower provisioned throughput at 10:45 AM back to 100-K RU/sec, you'll be charged for two hours of 200 K RU/sec.

Billing example: multiple containers, each with dedicated provisioned throughput mode

- If you create an Azure Cosmos account in East US 2 with two containers with provisioned throughput of 500 RU/sec and 700 RU/sec, respectively, you would have a total provisioned throughput of 1,200 RU/sec.
- You would be charged 1,200/100 *$0.008=$0.096/hour.
- If your throughput needs changed, and you've increased each container's capacity by 500 RU/sec while also creating a new unlimited container with 20,000 RU/sec, your overall provisioned capacity would be 22,200 RU/sec (1,000 RU/sec + 1,200 RU/sec + 20,000RU/sec).
- Your bill would then change to:$0.008x 222 =$1.776/hour.
- In a month of 720 hours (24 hours * 30 days), if for 500 hours provisioned throughput was 1,200 RU/sec and for the remaining 220 hours provisioned throughput was 22,200 RU/sec, your monthly bill shows: 500 x$0.096/hour + 220 x$1.776/hour =$438.72/month.

Billing example: containers with shared (provisioned) throughput mode

- If you create an Azure Cosmos account in East US 2 with two Azure Cosmos databases (with a set of containers sharing the throughput at the database level) with the provisioned throughput of 50-K RU/sec and 70-K RU/sec, respectively, you would have a total provisioned throughput of 120 K RU/sec.
- You would be charged 1200 x$0.008=$9.60/hour.
- If your throughput needs changed and you increased each database's provisioned throughput by 10K RU/sec for each database, and you add a new container to the first database with dedicated throughput mode of 15-K RU/sec to your shared throughput database, your overall provisioned capacity would be 155-K RU/sec (60 K RU/sec + 80 K RU/sec + 15 K RU/sec).
- Your bill would then change to: 1,550 *$0.008=$12.40/hour.
- In a month of 720 hours, if for 300 hours provisioned throughput was 120-K RU/sec and for the remaining 420 hours provisioned throughput was 155-K RU/sec, your monthly bill will show: 300 x$9.60/hour + 420 x$12.40/hour =$2,880+$5,208=$8,088/month.

Billing examples with geo-replication

You can add/remove Azure regions anywhere in the world to your Azure Cosmos database account at any time. The throughput that you have configured for various Azure Cosmos databases and containers will be reserved in each of the Azure regions associated with your Azure Cosmos database account. If the sum of provisioned throughput (RU/sec) configured across all the databases and containers within your Azure Cosmos database account (provisioned per hour) is T and the number of Azure regions associated with your database account is N, then the total provisioned throughput for a given hour, for your Azure Cosmos database account is equal to T x N RU/sec. Provisioned throughput (single write region) costs $0.008/hour per 100 RU/sec and provisioned throughput with multiple writable regions (multi-region writes config) costs $0.016/per hour per 100 RU/sec (see the Pricing page). Whether its single write region, or multiple write regions, Azure Cosmos DB allows you to read data from any region.

Billing example: multi-region Azure Cosmos account, single region writes

Let's assume you have an Azure Cosmos container in West US. The container is created with throughput 10K RU/sec and you store 1 TB of data this month. Let's assume you add three regions (East US, North Europe, and East Asia) to your Azure Cosmos account, each with the same storage and throughput. Your total monthly bill will be (assuming 30 days in a month). Your bill would be as follows:

ITEM	USAGE (MONTH)	RATE	MONTHLY COST
Throughput bill for container in West US	10K RU/sec * 24 * 30	$0.008per 100 RU/sec per hour	$576
Throughput bill for 3 additional regions - East US, North Europe, and East Asia	3 * 10K RU/sec * 24 * 30	$0.008per 100 RU/sec per hour	$1,728
Storage bill for container in West US	250 GB	$0.25/GB	$62.50
Storage bill for 3 additional regions - East US, North Europe, and East Asia	3 * 250 GB	$0.25/GB	$187.50

ITEM	USAGE (MONTH)	RATE	MONTHLY COST
Total			**$2,554**

Let's also assume that you egress 100 GB of data every month from the container in West US to replicate data into East US, North Europe, and East Asia. You're billed for egress as per data transfer rates.

Billing example: multi-region Azure Cosmos account, multi region writes

Let's assume you create an Azure Cosmos container in West US. The container is created with throughput 10K RU/sec and you store 1 TB of data this month. Let's assume you add three regions (East US, North Europe, and East Asia), each with the same storage and throughput and you want the ability to write to the containers in all regions associated with your Azure Cosmos account. Your total monthly bill will be (assuming 30 days in a month) as follows:

ITEM	USAGE (MONTH)	RATE	MONTHLY COST
Throughput bill for container in West US (all regions are writable)	10K RU/sec * 24 * 30	$0.016per 100 RU/sec per hour	$1,152
Throughput bill for 3 additional regions - East US, North Europe, and East Asia (all regions are writable)	(3 + 1) * 10K RU/sec * 24 * 30	$0.016per 100 RU/sec per hour	$4,608
Storage bill for container in West US	250 GB	$0.25/GB	$62.50
Storage bill for 3 additional regions - East US, North Europe, and East Asia	3 * 250 GB	$0.25/GB	$187.50
Total			**$6,010**

Let's also assume that you egress 100 GB of data every month from the container in West US to replicate data into East US, North Europe, and East Asia. You're billed for egress as per data transfer rates.

Billing example: Azure Cosmos account with multi-region writes, database-level throughput including dedicated throughput mode for some containers

Let's consider the following example, where we have a multi-region Azure Cosmos account where all regions are writable (multiple write regions config). For simplicity, we will assume storage size stays constant and doesn't change and omit it here to keep the example simpler. The provisioned throughput during the month varied as follows (assuming 30 days or 720 hours):

[0-100 hours]:

- We created a three region Azure Cosmos account (West US, East US, North Europe), where all regions are writable
- We created a database (D1) with shared throughput 10K RU/sec
- We created a database (D2) with shared throughput 30-K RU/sec and
- We created a container (C1) with dedicated throughput 20 K RU/sec

[101-200 hours]:

- We scaled up database (D1) to 50 K RU/sec
- We scaled up database (D2) to 70 K RU/sec
- We deleted container (C1)

[201-300 hours]:

- We created container (C1) again with dedicated throughput 20 K RU/sec

[301-400 hours]:

- We removed one of the regions from Azure Cosmos account (# of writable regions is now 2)
- We scaled down database (D1) to 10K RU/sec
- We scaled up database (D2) to 80 K RU/sec
- We deleted container (C1) again

[401-500 hours]:

- We scaled down database (D2) to 10K RU/sec
- We created container (C1) again with dedicated throughput 20 K RU/sec

[501-700 hours]:

- We scaled up database (D1) to 20 K RU/sec
- We scaled up database (D2) to 100 K RU/sec
- We deleted container (C1) again

[701-720 hours]:

- We scaled down database (D2) to 50 K RU/sec

Visually the changes in total provisioned throughput during 720 hours for the month are shown in the figure below:

The total monthly bill will be (assuming 30 days/720 hours in a month) will be computed as follows:

HOURS	RU/S	ITEM	USAGE (HOURLY)	COST

[0-100]	D1:10K
	D2:30K
	C1:20K
		Throughput bill for container in West US (all regions are writable)	`D1: 10K RU/sec/100 * $0.016 * 100 hours = $160`
			`D2: 30 K RU/sec/100 * $0.016 * 100 hours = $480`
			`C1: 20 K RU/sec/100 *$0.016 * 100 hours = $320`
				$960

		Throughput bill for 2 additional regions: East US, North Europe (all regions are writable)	`(2 + 1) * (60 K RU/sec /100 * $0.016) * 100 hours = $2,880`	$2,880
[101-200]	D1:50K D2:70K C1: --	Throughput bill for container in West US (all regions are writable)	`D1: 50 K RU/sec/100 * $0.016 * 100 hours = $800` `D2: 70 K RU/sec/100 * $0.016 * 100 hours = $1,120`	$1920
		Throughput bill for 2 additional regions: East US, North Europe (all regions are writable)	`(2 + 1) * (120 K RU/sec /100 * $0.016) * 100 hours = $5,760`	$5,760

[201-300]	D1:50K
	D2:70K
	C1:20K
		Throughput bill for container in West US (all regions are writable)	`D1: 50 K RU/sec/100 * $0.016 * 100 hours = $800`
			`D2: 70 K RU/sec/100 * $0.016 * 100 hours = $1,120`
			`C1: 20 K RU/sec/100 *$0.016 * 100 hours = $320`
				$2,240

		Throughput bill for 2 additional regions: East US, North Europe (all regions are writable)	`(2 + 1) * (140 K RU/sec /100 * $0.016-) * 100 hours = $6,720`	$6,720
[301-400]	D1:10K D2:80K C1: --	Throughput bill for container in West US (all regions are writable)	`D1: 10K RU/sec/100 * $0.016 * 100 hours = $160` `D2: 80 K RU/sec/100 * $0.016 * 100 hours = $1,280`	$1,440

HOURS	RU/S	ITEM	USAGE (HOURLY)	COST
		Throughput bill for 2 additional regions: East US, North Europe (all regions are writable)	`(1 + 1) * (90 K RU/sec /100 * $0.016) * 100 hours = $2,880`	$2,880

[401-500]	D1:10K
	D2:10K
	C1:20K
		Throughput bill for container in West US (all regions are writable)	`D1: 10K RU/sec/100 * $0.016 * 100 hours = $160`
			`D2: 10K RU/sec/100 * $0.016 * 100 hours = $160`
			`C1: 20 K RU/sec/100 *$0.016 * 100 hours = $320`
				$640

		Throughput bill for 2 additional regions: East US, North Europe (all regions are writable)	`(1 + 1) * (40 K RU/sec /100 * $0.016) * 100 hours = $1,280`	$1,280
[501-700]	D1:20K D2:100K C1: --	Throughput bill for container in West US (all regions are writable)	`D1: 20 K RU/sec/100 * $0.016 * 200 hours = $640` `D2: 100 K RU/sec/100 * $0.016 * 200 hours = $3,200`	$3,840
		Throughput bill for 2 additional regions: East US, North Europe (all regions are writable)	`(1 + 1) * (120 K RU/sec /100 * $0.016) * 200 hours = $1,280`	$7,680
[701-720]	D1:20K D2:50K C1: --	Throughput bill for container in West US (all regions are writable)	`D1: 20 K RU/sec/100 *$0.016 * 20 hours = $64` `D2: 50 K RU/sec/100 *$0.016 * 20 hours = $160`	$224
		Throughput bill for 2 additional regions: East US, North Europe (all regions are writable)	`(1 + 1) * (70 K RU/sec /100 * $0.016) * 20 hours = $448`	$224
		Total Monthly Cost		**$38,688**

Billing examples with free tier accounts

With Azure Cosmos DB free tier, you'll get the first 400 RU/s and 5 GB of storage in your account for free, applied at the account level. Any RU/s and storage beyond 400 RU/s and 5 GB will be billed at the regular pricing rates per the pricing page. On the bill, you will not see a charge or line item for the free 400 Ru/s and 5 GB, only the RU/s

and storage beyond what is covered by free tier. The 400 RU/s applies to any type of RU/s - provisioned throughput, autoscale, and multi-region writes.

Billing example - container or database with provisioned throughput

- Let's suppose we create a database or container in a free tier account with 400 RU/s and 5 GB of storage.
- Your bill will not show any charge for this resource. Your hourly and monthly cost will be $0.
- Now, let's suppose in the same account, we add another database or container with 1000 RU/s and 10 GB of storage.
- Your bill will now show a charge for the 1000 RU/s and 10 GB of storage.

Billing example - container with autoscale throughput

- Let's suppose in a free tier account, we create a container with autoscale enabled, with a maximum RU/s of 4000 RU/s. This resource will automatically scale between 400 RU/s - 4000 RU/s.
- Suppose in hour 1 through hour 10, the resource is at the minimum of 400 RU/s. During hour 11, the resource scales up to 1000 RU/s and then back down to 400 RU/s within the hour.
- In hours 1 through 10, you will be billed $0 for throughput, as the 400 RU/s were covered by free tier.
- In hour 11, you will be billed for an effective 1000 RU/s - 400 RU/s = 600 RU/s, as this is the highest RU/s in the hour. This will be 6 units of 100 RU/s for the hour, so the total throughput cost for the hour will be 6 units * $0.012 = $0.072.
- Any storage beyond the first 5 GB will be billed at normal storage rates.

Billing example - multi-region, single write region account

- Let's suppose in a free tier account, we create a database or container with 1200 RU/s and 10 GB of storage. We replicate the account to 3 regions, and we have a single write-region account.
- In total, without free tier, we would be billed for 3 * 1200 RU/s = 3600 RU/s and 3 * 10 GB = 30 GB of storage.
- With the free tier discount, after removing 400 RU/s and 5 GB of storage, we will be billed for an effective 3200 RU/s (32 units) of provisioned throughput at the single write region rate and 25 GB of storage.
- The monthly cost for RU/s would be: 32 units * $0.008 * 24 hours * 31 days = $190.46. The monthly cost for storage would be: 25 GB * 0.25 / GB = $6.25. The total cost would be $190.46 + $6.25 = $196.71.
- Note: if the unit price for RU/s or storage differs in the regions, the free tier 400 RU/s and 5 GB will reflect the rates of the region the account was created in.

Billing example - multi-region, account with multiple write regions

This example reflects pricing for multi-region writes for accounts created after December 1, 2019.

- Let's suppose in a free tier account, we create a database or container with 1200 RU/s and 10 GB of storage. We replicate the account to 3 regions, and we have a multiple write regions account.
- In total, without free tier, we would be billed for 3 * 1200 RU/s = 3600 RU/s and 3 * 10 GB = 30 GB of storage.
- With the free tier discount, after removing 400 RU/s and 5 GB of storage, we will be billed for an effective 3200 RU/s (32 units) of provisioned throughput at the multiple write region rate and 25 GB of storage.
- The monthly cost for RU/s would be: 32 units * $0.016 * 24 hours * 31 days = $380.93. The monthly cost for storage would be: 25 GB * 0.25 / GB = $6.25. The total cost would be $380.93 + $6.25 = $387.18.

Proactively estimating your monthly bill

Let's consider another example, where you want to proactively estimate your bill before the month's end. You can estimate your bill as follows:

Storage cost

- Avg Record Size (KB) = 1
- Number of Records = 100,000,000

- Total Storage (GB) = 100
- Monthly cost per GB = $0.25
- Expected Monthly Cost for Storage = $25.00

Throughput Cost

OPERATION TYPE	REQUESTS/SEC	AVG. RU/REQUEST	RUS NEEDED
Write	100	5	500
Read	400	1	400

Total RU/sec: 500 + 400 = 900 Hourly cost: 900/100 * $0.008 = $0.072 Expected Monthly Cost for Throughput (assuming 31 days): $0.072 * 24 * 31 = $53.57

Total Monthly Cost

Total Monthly Cost = Monthly Cost for Storage + Monthly Cost for Throughput Total Monthly Cost = $25.00 + $53.57 = $78.57

Pricing may vary by region. For up-to-date pricing, see the Pricing page.

Billing with Azure Cosmos DB reserved capacity

Azure Cosmos DB reserved capacity enables you to purchase provisioned throughput in advance (a reserved capacity or a reservation) that can be applied to all Azure Cosmos databases and containers (for any API or data model) across all Azure regions. Because provisioned throughput price varies per region, it helps to think of reserved capacity as a monetary credit that you've purchased at a discount, that can be drawn from for the provisioned throughput at the respective price in each region. For example, let's say you have an Azure Cosmos account with a single container provisioned with 50-K RU/sec and globally replicated two regions - East US and Japan East. If you choose the pay-as-you-go option, you would pay:

- in East US: for 50-K RU/sec at the rate of$0.008per 100 RU/sec in that region
- in Japan East: for 50-K RU/sec at the rate of$0.009per 100 RU/sec in that region

Your total bill (without reserved capacity) would be (assuming 30 days or 720 hours):

REGION	HOURLY PRICE PER 100 RU/S	UNITS (RU/S)	BILLED AMOUNT (HOURLY)	BILLED AMOUNT (MONTHLY)
East US	$0.008	50 K	$4	$2,880
Japan East	$0.009	50 K	$4.50	$3,240
Total			$8.50	$6,120

Let's consider that you've bought reserved capacity instead. You can buy reserved capacity for 100-K RU/sec at the price of$56,064 for one year (at 20% discount), or$6.40per hour. See reserved capacity pricing on the Pricing page).

- Cost of throughput (pay-as-you-go): 100,000 RU/sec/100 * $0.008/hour * 8760 hours in a year = $70,080
- Cost of throughput (with reserved capacity) $70,080 discounted at 20% = $56,064

What you've effectively purchased is a credit of$8per hour, for 100 K RU/sec using the list price in East US, at the price of$6.40per hour. You can then draw down from this pre-paid throughput reservation on an hourly basis for

the provisioned throughput capacity in any global Azure region at the respective regional list prices set for your subscription. In this example, where you provision 50 K RU/sec each in East US, and Japan East, you will be able to draw$8.00worth of provisioned throughput per hour, and will be billed the overage of$0.50per hour (or$360/month).

REGION	HOURLY PRICE PER 100 RU/S	UNITS (RU/S)	BILLED AMOUNT (HOURLY)	BILLED AMOUNT (MONTHLY)
East US	$0.008	50 K	$4	$2,880
Japan East	$0.009	50 K	$4.50	$3,240
		Pay-as-you-go	$8.50	$6120
Reserved Capacity Purchased	$0.0064 (20% discount)	100 RU/sec or $8 capacity pre-purchased	-$8	-$5,760
Net Bill			$0.50	$360

Next Steps

Next you can proceed to learn about cost optimization in Azure Cosmos DB with the following articles:

- Learn more about How Cosmos DB pricing model is cost-effective for customers
- Learn more about Optimizing for development and testing
- Learn more about Optimizing throughput cost
- Learn more about Optimizing storage cost
- Learn more about Optimizing the cost of reads and writes
- Learn more about Optimizing the cost of queries
- Learn more about Optimizing the cost of multi-region Azure Cosmos accounts

Optimize provisioned throughput cost in Azure Cosmos DB

12/4/2020 • 14 minutes to read • Edit Online

APPLIES TO: SQL API Cassandra API Gremlin API Table API Azure Cosmos DB API for MongoDB

By offering provisioned throughput model, Azure Cosmos DB offers predictable performance at any scale. Reserving or provisioning throughput ahead of time eliminates the "noisy neighbor effect" on your performance. You specify the exact amount of throughput you need and Azure Cosmos DB guarantees the configured throughput, backed by SLA.

You can start with a minimum throughput of 400 RU/sec and scale up to tens of millions of requests per second or even more. Each request you issue against your Azure Cosmos container or database, such as a read request, write request, query request, stored procedures have a corresponding cost that is deducted from your provisioned throughput. If you provision 400 RU/s and issue a query that costs 40 RUs, you will be able to issue 10 such queries per second. Any request beyond that will get rate-limited and you should retry the request. If you are using client drivers, they support the automatic retry logic.

You can provision throughput on databases or containers and each strategy can help you save on costs depending on the scenario.

Optimize by provisioning throughput at different levels

- If you provision throughput on a database, all the containers, for example collections/tables/graphs within that database can share the throughput based on the load. Throughput reserved at the database level is shared unevenly, depending on the workload on a specific set of containers.
- If you provision throughput on a container, the throughput is guaranteed for that container, backed by the SLA. The choice of a logical partition key is crucial for even distribution of load across all the logical partitions of a container. See Partitioning and horizontal scaling articles for more details.

The following are some guidelines to decide on a provisioned throughput strategy:

Consider provisioning throughput on an Azure Cosmos database (containing a set of containers) if:

1. You have a few dozen Azure Cosmos containers and want to share throughput across some or all of them.
2. You are migrating from a single-tenant database designed to run on IaaS-hosted VMs or on-premises, for example, NoSQL or relational databases to Azure Cosmos DB. And if you have many collections/tables/graphs and you do not want to make any changes to your data model. Note, you might have to compromise some of the benefits offered by Azure Cosmos DB if you are not updating your data model when migrating from an on-premises database. It's recommended that you always reaccess your data model to get the most in terms of performance and also to optimize for costs.
3. You want to absorb unplanned spikes in workloads by virtue of pooled throughput at the database level subjected to unexpected spike in workload.
4. Instead of setting specific throughput on individual containers, you care about getting the aggregate throughput across a set of containers within the database.

Consider provisioning throughput on an individual container if:

1. You have a few Azure Cosmos containers. Because Azure Cosmos DB is schema-agnostic, a container can contain items that have heterogeneous schemas and does not require customers to create multiple container types, one for each entity. It is always an option to consider if grouping separate say 10-20 containers into a single container makes sense. With a 400 RUs minimum for containers, pooling all 10-20 containers into one could be more cost effective.

2. You want to control the throughput on a specific container and get the guaranteed throughput on a given container backed by SLA.

Consider a hybrid of the above two strategies:

1. As mentioned earlier, Azure Cosmos DB allows you to mix and match the above two strategies, so you can now have some containers within Azure Cosmos database, which may share the throughput provisioned on the database as well as, some containers within the same database, which may have dedicated amounts of provisioned throughput.

2. You can apply the above strategies to come up with a hybrid configuration, where you have both database level provisioned throughput with some containers having dedicated throughput.

As shown in the following table, depending on the choice of API, you can provision throughput at different granularities.

API	FOR SHARED THROUGHPUT, CONFIGURE	FOR DEDICATED THROUGHPUT, CONFIGURE
SQL API	Database	Container
Azure Cosmos DB's API for MongoDB	Database	Collection
Cassandra API	Keyspace	Table
Gremlin API	Database account	Graph
Table API	Database account	Table

By provisioning throughput at different levels, you can optimize your costs based on the characteristics of your workload. As mentioned earlier, you can programmatically and at any time increase or decrease your provisioned throughput for either individual container(s) or collectively across a set of containers. By elastically scaling throughput as your workload changes, you only pay for the throughput that you have configured. If your container or a set of containers is distributed across multiple regions, then the throughput you configure on the container or a set of containers is guaranteed to be made available across all regions.

Optimize with rate-limiting your requests

For workloads that aren't sensitive to latency, you can provision less throughput and let the application handle rate-limiting when the actual throughput exceeds the provisioned throughput. The server will preemptively end the request with `RequestRateTooLarge` (HTTP status code 429) and return the `x-ms-retry-after-ms` header indicating the amount of time, in milliseconds, that the user must wait before retrying the request.

```
HTTP Status 429,
 Status Line: RequestRateTooLarge
 x-ms-retry-after-ms :100
```

Retry logic in SDKs

The native SDKs (.NET/.NET Core, Java, Node.js and Python) implicitly catch this response, respect the server-

specified retry-after header, and retry the request. Unless your account is accessed concurrently by multiple clients, the next retry will succeed.

If you have more than one client cumulatively operating consistently above the request rate, the default retry count, which is currently set to 9, may not be sufficient. In such cases, the client throws a `RequestRateTooLargeException` with status code 429 to the application. The default retry count can be changed by setting the `RetryOptions` on the ConnectionPolicy instance. By default, the `RequestRateTooLargeException` with status code 429 is returned after a cumulative wait time of 30 seconds if the request continues to operate above the request rate. This occurs even when the current retry count is less than the max retry count, be it the default of 9 or a user-defined value.

MaxRetryAttemptsOnThrottledRequests is set to 3, so in this case, if a request operation is rate limited by exceeding the reserved throughput for the container, the request operation retries three times before throwing the exception to the application. MaxRetryWaitTimeInSeconds is set to 60, so in this case if the cumulative retry wait time in seconds since the first request exceeds 60 seconds, the exception is thrown.

```
ConnectionPolicy connectionPolicy = new ConnectionPolicy();
connectionPolicy.RetryOptions.MaxRetryAttemptsOnThrottledRequests = 3;
connectionPolicy.RetryOptions.MaxRetryWaitTimeInSeconds = 60;
```

Partitioning strategy and provisioned throughput costs

Good partitioning strategy is important to optimize costs in Azure Cosmos DB. Ensure that there is no skew of partitions, which are exposed through storage metrics. Ensure that there is no skew of throughput for a partition, which is exposed with throughput metrics. Ensure that there is no skew towards particular partition keys. Dominant keys in storage are exposed through metrics but the key will be dependent on your application access pattern. It's best to think about the right logical partition key. A good partition key is expected to have the following characteristics:

- Choose a partition key that spreads workload evenly across all partitions and evenly over time. In other words, you shouldn't have some keys to with majority of the data and some keys with less or no data.
- Choose a partition key that enables access patterns to be evenly spread across logical partitions. The workload is reasonably even across all the keys. In other words, the majority of the workload shouldn't be focused on a few specific keys.
- Choose a partition key that has a wide range of values.

The basic idea is to spread the data and the activity in your container across the set of logical partitions, so that resources for data storage and throughput can be distributed across the logical partitions. Candidates for partition keys may include the properties that appear frequently as a filter in your queries. Queries can be efficiently routed by including the partition key in the filter predicate. With such a partitioning strategy, optimizing provisioned throughput will be a lot easier.

Design smaller items for higher throughput

The request charge or the request processing cost of a given operation is directly correlated to the size of the item. Operations on large items will cost more than operations on smaller items.

Data access patterns

It is always a good practice to logically separate your data into logical categories based on how frequently you access the data. By categorizing it as hot, medium, or cold data you can fine-tune the storage consumed and the throughput required. Depending on the frequency of access, you can place the data into separate containers (for example, tables, graphs, and collections) and fine-tune the provisioned throughput on them to accommodate to the needs of that segment of data.

Furthermore, if you're using Azure Cosmos DB, and you know you are not going to search by certain data values or will rarely access them, you should store the compressed values of these attributes. With this method you save on storage space, index space, and provisioned throughput and result in lower costs.

Optimize by changing indexing policy

By default, Azure Cosmos DB automatically indexes every property of every record. This is intended to ease development and ensure excellent performance across many different types of ad hoc queries. If you have large records with thousands of properties, paying the throughput cost for indexing every property may not be useful, especially if you only query against 10 or 20 of those properties. As you get closer to getting a handle on your specific workload, our guidance is to tune your index policy. Full details on Azure Cosmos DB indexing policy can be found here.

Monitoring provisioned and consumed throughput

You can monitor the total number of RUs provisioned, number of rate-limited requests as well as the number of RUs you've consumed in the Azure portal. The following image shows an example usage metric:

You can also set alerts to check if the number of rate-limited requests exceeds a specific threshold. See How to monitor Azure Cosmos DB article for more details. These alerts can send an email to the account administrators or call a custom HTTP Webhook or an Azure Function to automatically increase provisioned throughput.

Scale your throughput elastically and on-demand

Since you are billed for the throughput provisioned, matching the provisioned throughput to your needs can help you avoid the charges for the unused throughput. You can scale your provisioned throughput up or down any time, as needed. If your throughput needs are very predictable you can use Azure Functions and use a Timer Trigger to increase or decrease throughput on a schedule.

- Monitoring the consumption of your RUs and the ratio of rate-limited requests may reveal that you do not need to keep provisioned throughout constant throughout the day or the week. You may receive less traffic at night or during the weekend. By using either Azure portal or Azure Cosmos DB native SDKs or REST API, you can scale your provisioned throughput at any time. Azure Cosmos DB's REST API provides endpoints to programmatically update the performance level of your containers making it straightforward to adjust the throughput from your code depending on the time of the day or the day of the week. The operation is performed without any downtime, and typically takes effect in less than a minute.

- One of the areas you should scale throughput is when you ingest data into Azure Cosmos DB, for example, during data migration. Once you have completed the migration, you can scale provisioned throughput down to handle the solution's steady state.
- Remember, the billing is at the granularity of one hour, so you will not save any money if you change your provisioned throughput more often than one hour at a time.

Determine the throughput needed for a new workload

To determine the provisioned throughput for a new workload, you can use the following steps:

1. Perform an initial, rough evaluation using the capacity planner and adjust your estimates with the help of the Azure Cosmos DB Explorer in the Azure portal.
2. It's recommended to create the containers with higher throughput than expected and then scaling down as needed.
3. It's recommended to use one of the native Azure Cosmos DB SDKs to benefit from automatic retries when requests get rate-limited. If you're working on a platform that is not supported and use Cosmos DB's REST API, implement your own retry policy using the `x-ms-retry-after-ms` header.
4. Make sure that your application code gracefully supports the case when all retries fail.
5. You can configure alerts from the Azure portalto get notifications for rate-limiting. You can start with conservative limits like 10 rate-limited requests over the last 15 minutes and switch to more eager rules once you figure out your actual consumption. Occasional rate-limits are fine, they show that you're playing with the limits you've set and that's exactly what you want to do.
6. Use monitoring to understand your traffic pattern, so you can consider the need to dynamically adjust your throughput provisioning over the day or a week.
7. Monitor your provisioned vs. consumed throughput ratio regularly to make sure you have not provisioned more than required number of containers and databases. Having a little over provisioned throughput is a good safety check.

Best practices to optimize provisioned throughput

The following steps help you to make your solutions highly scalable and cost-effective when using Azure Cosmos DB.

1. If you have significantly over provisioned throughput across containers and databases, you should review RUs provisioned Vs consumed RUs and fine-tune the workloads.
2. One method for estimating the amount of reserved throughput required by your application is to record the request unit RU charge associated with running typical operations against a representative Azure Cosmos container or database used by your application and then estimate the number of operations you anticipate to perform each second. Be sure to measure and include typical queries and their usage as well. To learn how to estimate RU costs of queries programmatically or using portal see Optimizing the cost of queries.
3. Another way to get operations and their costs in RUs is by enabling Azure Monitor logs, which will give you the breakdown of operation/duration and the request charge. Azure Cosmos DB provides request charge for every operation, so every operation charge can be stored back from the response and then used for analysis.
4. You can elastically scale up and down provisioned throughput as you need to accommodate your workload needs.
5. You can add and remove regions associated with your Azure Cosmos account as you need and control

costs.

6. Make sure you have even distribution of data and workloads across logical partitions of your containers. If you have uneven partition distribution, this may cause to provision higher amount of throughput than the value that is needed. If you identify that you have a skewed distribution, we recommend redistributing the workload evenly across the partitions or repartition the data.

7. If you have many containers and these containers do not require SLAs, you can use the database-based offer for the cases where the per container throughput SLAs do not apply. You should identify which of the Azure Cosmos containers you want to migrate to the database level throughput offer and then migrate them by using a change feed-based solution.

8. Consider using the "Cosmos DB Free Tier" (free for one year), Try Cosmos DB (up to three regions) or downloadable Cosmos DB emulator for dev/test scenarios. By using these options for test-dev, you can substantially lower your costs.

9. You can further perform workload-specific cost optimizations – for example, increasing batch-size, load-balancing reads across multiple regions, and de-duplicating data, if applicable.

10. With Azure Cosmos DB reserved capacity, you can get significant discounts for up to 65% for three years. Azure Cosmos DB reserved capacity model is an upfront commitment on requests units needed over time. The discounts are tiered such that the more request units you use over a longer period, the more your discount will be. These discounts are applied immediately. Any RUs used above your provisioned values are charged based on the non-reserved capacity cost. See Cosmos DB reserved capacity) for more details. Consider purchasing reserved capacity to further lower your provisioned throughput costs.

Next steps

Next you can proceed to learn more about cost optimization in Azure Cosmos DB with the following articles:

- Learn more about Optimizing for development and testing
- Learn more about Understanding your Azure Cosmos DB bill
- Learn more about Optimizing storage cost
- Learn more about Optimizing the cost of reads and writes
- Learn more about Optimizing the cost of queries
- Learn more about Optimizing the cost of multi-region Azure Cosmos accounts

Optimize request cost in Azure Cosmos DB

11/2/2020 • 7 minutes to read • Edit Online

APPLIES TO: SQL API Cassandra API Gremlin API Table API Azure Cosmos DB API for MongoDB

This article describes how read and write requests translate into Request Units and how to optimize the cost of these requests. Read operations include point reads and queries. Write operations include insert, replace, delete, and upsert of items.

Azure Cosmos DB offers a rich set of database operations that operate on the items within a container. The cost associated with each of these operations varies based on the CPU, IO, and memory required to complete the operation. Instead of thinking about and managing hardware resources, you can think of a Request Unit (RU) as a single measure for the resources required to perform various database operations to serve a request.

Measuring the RU charge of a request

It is important to measure the RU charge of your requests to understand their actual cost and also evaluate the effectiveness of your optimizations. You can fetch this cost by using the Azure portal or inspecting the response sent back from Azure Cosmos DB through one of the SDKs. See Find the request unit charge in Azure Cosmos DB for detailed instructions on how to achieve that.

Reading data: point reads and queries

Read operations in Azure Cosmos DB are typically ordered from fastest/most efficient to slower/less efficient in terms of RU consumption as follows:

- Point reads (key/value lookup on a single item ID and partition key).
- Query with a filter clause within a single partition key.
- Query without an equality or range filter clause on any property.
- Query without filters.

Role of the consistency level

When using either the **strong** or **bounded staleness** consistency levels, the RU cost of any read operation (point read or query) is doubled.

Point reads

The only factor affecting the RU charge of a point read (besides the consistency level used) is the size of the item retrieved. The following table shows the RU cost of point reads for items that are 1 KB and 100 KB in size.

ITEM SIZE	COST OF ONE POINT READ
1 KB	1 RU
100 KB	10 RUs

Because point reads (key/value lookups on the item ID) are the most efficient kind of read, you should make sure your item ID has a meaningful value so you can fetch your items with a point read (instead of a query) when possible.

Queries

Request units for queries are dependent on a number of factors. For example, the number of Azure Cosmos items loaded/returned, the number of lookups against the index, the query compilation time etc. details. Azure Cosmos DB guarantees that the same query when executed on the same data will always consume the same number of request units even with repeat executions. The query profile using query execution metrics gives you a good idea of how the request units are spent.

In some cases you may see a sequence of 200 and 429 responses, and variable request units in a paged execution of queries, that is because queries will run as fast as possible based on the available RUs. You may see a query execution break into multiple pages/round trips between server and client. For example, 10,000 items may be returned as multiple pages, each charged based on the computation performed for that page. When you sum across these pages, you should get the same number of RUs as you would get for the entire query.

Metrics for troubleshooting queries

The performance and the throughput consumed by queries (including user-defined functions) mostly depend on the function body. The easiest way to find out how much time the query execution is spent in the UDF and the number of RUs consumed, is by enabling the query metrics. If you use the .NET SDK, here are sample query metrics returned by the SDK:

```
Retrieved Document Count : 1
Retrieved Document Size : 9,963 bytes
Output Document Count : 1
Output Document Size : 10,012 bytes
Index Utilization : 100.00 %
Total Query Execution Time : 0.48 milliseconds
 Query Preparation Times
 Query Compilation Time : 0.07 milliseconds
 Logical Plan Build Time : 0.03 milliseconds
 Physical Plan Build Time : 0.05 milliseconds
 Query Optimization Time : 0.00 milliseconds
 Index Lookup Time : 0.06 milliseconds
 Document Load Time : 0.03 milliseconds
 Runtime Execution Times
 Query Engine Execution Time : 0.03 milliseconds
 System Function Execution Time : 0.00 milliseconds
 User-defined Function Execution Time : 0.00 milliseconds
 Document Write Time : 0.00 milliseconds
 Client Side Metrics
 Retry Count : 1
Request Charge : 3.19 RUs
```

Best practices to cost optimize queries

Consider the following best practices when optimizing queries for cost:

- Colocate multiple entity types

  Try to colocate multiple entity types within a single or smaller number of containers. This method yields benefits not only from a pricing perspective, but also for query execution and transactions. Queries are scoped to a single container; and atomic transactions over multiple records via stored procedures/triggers are scoped to a partition key within a single container. Colocating entities within the same container can reduce the number of network round trips to resolve relationships across records. So it increases the end-to-end performance, enables atomic transactions over multiple records for a larger dataset, and as a result lowers costs. If colocating multiple entity types within a single or smaller number of containers is difficult for your scenario, usually because you are migrating an existing application and you do not want to make any code changes - you should then consider provisioning throughput at the database level.

- Measure and tune for lower request units/second usage

The complexity of a query impacts how many request units (RUs) are consumed for an operation. The number of predicates, nature of the predicates, number of UDFs, and the size of the source data set. All these factors influence the cost of query operations.

Azure Cosmos DB provides predictable performance in terms of throughput and latency by using a provisioned throughput model. The throughput provisioned is represented in terms of Request Units per second, or RU/s. A Request Unit (RU) is a logical abstraction over compute resources such as CPU, memory, IO, etc. that are required to perform a request. The provisioned throughput (RUs) is set aside and dedicated to your container or database to provide predictable throughput and latency. Provisioned throughput enables Azure Cosmos DB to provide predictable and consistent performance, guaranteed low latency, and high availability at any scale. Request units represent the normalized currency that simplifies the reasoning about how many resources an application needs.

Request charge returned in the request header indicates the cost of a given query. For example, if a query returns 1000 1-KB items, the cost of the operation is 1000. As such, within one second, the server honors only two such requests before rate limiting subsequent requests. For more information, see request units article and the request unit calculator.

Writing data

The RU cost of writing an item depends on:

- The item size.
- The number of properties covered by the indexing policy and needed to be indexed.

Inserting a 1 KB item with less than 5 properties to index costs around 5 RUs. Replacing an item costs two times the charge required to insert the same item.

Optimizing writes

The best way to optimize the RU cost of write operations is to rightsize your items and the number of properties that get indexed.

- Storing very large items in Azure Cosmos DB results in high RU charges and can be considered as an anti-pattern. In particular, don't store binary content or large chunks of text that you don't need to query on. A best practice is to put this kind of data in Azure Blob Storage and store a reference (or link) to the blob in the item you write to Azure Cosmos DB.
- Optimizing your indexing policy to only index the properties that your queries filter on can make a huge difference in the RUs consumed by your write operations. When creating a new container, the default indexing policy indexes each and every property found in your items. While this is a good default for development activities, it is highly recommended to re-evaluate and customize your indexing policy when going to production or when your workload begins to receive significant traffic.

When performing bulk ingestion of data, it is also recommended to use the Azure Cosmos DB bulk executor library as it is designed to optimize the RU consumption of such operations. Optionally, you can also use Azure Data Factory which is built on that same library.

Next steps

Next you can proceed to learn more about cost optimization in Azure Cosmos DB with the following articles:

- Learn more about Optimizing for development and testing
- Learn more about Understanding your Azure Cosmos DB bill
- Learn more about Optimizing throughput cost
- Learn more about Optimizing storage cost
- Learn more about Optimizing the cost of multi-region Azure Cosmos accounts

- Learn more about Azure Cosmos DB reserved capacity

Optimize storage cost in Azure Cosmos DB

11/2/2020 • 4 minutes to read • Edit Online

APPLIES TO: SQL API Cassandra API Gremlin API Table API Azure Cosmos DB API for MongoDB

Azure Cosmos DB offers unlimited storage and throughput. Unlike throughput, which you have to provision/configure on your Azure Cosmos containers or databases, the storage is billed based on a consumption basis. You are billed only for the logical storage you consume and you don't have to reserve any storage in advance. Storage automatically scales up and down based on the data that you add or remove to an Azure Cosmos container.

Storage cost

Storage is billed with the unit of GBs. Local SSD-backed storage is used by your data and indexing. The total storage used is equal to the storage required by the data and indexes used across all the regions where you are using Azure Cosmos DB. If you globally replicate an Azure Cosmos account across three regions, you will pay for the total storage cost in each of those three regions. To estimate your storage requirement, see capacity planner tool. The cost for storage in Azure Cosmos DB is $0.25 GB/month, see Pricing page for latest updates. You can set up alerts to determine storage used by your Azure Cosmos container, to monitor your storage, see Monitor Azure Cosmos DB) article.

Optimize cost with item size

Azure Cosmos DB expects the item size to be 2 MB or less for optimal performance and cost benefits. If you need any item to store larger than 2 MB of data, consider redesigning the item schema. In the rare event that you cannot redesign the schema, you can split the item into subitems and link them logically with a common identifier(ID). All the Azure Cosmos DB features work consistently by anchoring to that logical identifier.

Optimize cost with indexing

By default, the data is automatically indexed, which can increase the total storage consumed. However, you can apply custom index policies to reduce this overhead. Automatic indexing that has not been tuned through policy is about 10-20% of the item size. By removing or customizing index policies, you don't pay extra cost for writes and don't require additional throughput capacity. See Indexing in Azure Cosmos DB to configure custom indexing policies. If you have worked with relational databases before, you may think that "index everything" means doubling of storage or higher. However, in Azure Cosmos DB, in the median case, it's much lower. In Azure Cosmos DB, the storage overhead of index is typically low (10-20%) even with automatic indexing, because it is designed for a low storage footprint. By managing the indexing policy, you can control the tradeoff of index footprint and query performance in a more fine-grained manner.

Optimize cost with time to live and change feed

Once you no longer need the data you can gracefully delete it from your Azure Cosmos account by using time to live, change feed or you can migrate the old data to another data store such as Azure blob storage or Azure data warehouse. With time to live or TTL, Azure Cosmos DB provides the ability to delete items automatically from a container after a certain time period. By default, you can set time to live at the container level and override the value on a per-item basis. After you set the TTL at a container or at an item level, Azure Cosmos DB will automatically remove these items after the time period since the time they were last modified. By using change feed, you can migrate data to either another container in Azure Cosmos DB or to an external data store. The

migration takes zero down time and when you are done migrating, you can either delete or configure time to live to delete the source Azure Cosmos container.

Optimize cost with rich media data types

If you want to store rich media types, for example, videos, images, etc., you have a number of options in Azure Cosmos DB. One option is to store these rich media types as Azure Cosmos items. There is a 2-MB limit per item, and you can avoid this limit by chaining the data item into multiple subitems. Or you can store them in Azure Blob storage and use the metadata to reference them from your Azure Cosmos items. There are a number of pros and cons with this approach. The first approach gets you the best performance in terms of latency, throughput SLAs plus turnkey global distribution capabilities for the rich media data types in addition to your regular Azure Cosmos items. However the support is available at a higher price. By storing media in Azure Blob storage, you could lower your overall costs. If latency is critical, you could use premium storage for rich media files that are referenced from Azure Cosmos items. This integrates natively with CDN to serve images from the edge server at lower cost to circumvent geo-restriction. The down side with this scenario is that you have to deal with two services - Azure Cosmos DB and Azure Blob storage, which may increase operational costs.

Check storage consumed

To check the storage consumption of an Azure Cosmos container, you can run a HEAD or GET request on the container, and inspect the `x-ms-request-quota` and the `x-ms-request-usage` headers. Alternatively, when working with the .NET SDK, you can use the DocumentSizeQuota, and DocumentSizeUsage properties to get the storage consumed.

Using SDK

```
// Measure the item size usage (which includes the index size)
ResourceResponse<DocumentCollection> collectionInfo = await
client.ReadDocumentCollectionAsync(UriFactory.CreateDocumentCollectionUri("db", "coll"));

Console.WriteLine("Item size quota: {0}, usage: {1}", collectionInfo.DocumentQuota,
collectionInfo.DocumentUsage);
```

Next steps

Next you can proceed to learn more about cost optimization in Azure Cosmos DB with the following articles:

- Learn more about Optimizing for development and testing
- Learn more about Understanding your Azure Cosmos DB bill
- Learn more about Optimizing throughput cost
- Learn more about Optimizing the cost of reads and writes
- Learn more about Optimizing the cost of queries
- Learn more about Optimizing the cost of multi-region Azure Cosmos accounts

Optimize multi-region cost in Azure Cosmos DB

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: SQL API Cassandra API Gremlin API Table API Azure Cosmos DB API for MongoDB

You can add and remove regions to your Azure Cosmos account at any time. The throughput that you configure for various Azure Cosmos databases and containers is reserved in each region associated with your account. If the throughput provisioned per hour, that is the sum of RU/s configured across all the databases and containers for your Azure Cosmos account is `T` and the number of Azure regions associated with your database account is `N`, then the total provisioned throughput for your Cosmos account, for a given hour is equal to `T x N RU/s`.

Provisioned throughput with single write region costs $0.008/hour per 100 RU/s and provisioned throughput with multiple writable regions costs $0.016/per hour per 100 RU/s. To learn more, see Azure Cosmos DB Pricing page.

Costs for multiple write regions

In a multi-region writes system, the net available RUs for write operations increases `N` times where `N` is the number of write regions. Unlike single region writes, every region is now writable and supports conflict resolution. From the cost planning point of view, to perform `M` RU/s worth of writes worldwide, you will need to provision M `RUs` at a container or database level. You can then add as many regions as you would like and use them for writes to perform `M` RU worth of worldwide writes.

Example

Consider that you have a container in West US configured for single-region writes, provisioned with throughput of 10K RU/s, storing 0.5 TB of data this month. Let's assume you add a region, East US, with the same storage and throughput and you want the ability to write to the containers in both the regions from your app. Your new total monthly bill (assuming 730 hours in a month) will be as follows:

ITEM	USAGE (MONTHLY)	RATE	MONTHLY COST
Throughput bill for container in West US (single write region)	10K RU/s * 730 hours	$0.008per 100 RU/s per hour	$584
Throughput bill for container in 2 regions - West US & East US (multiple write regions)	2 * 10K RU/s * 730 hours	$0.016per 100 RU/s per hour	$2,336
Storage bill for container in West US	0.5 TB (or 512 GB)	$0.25/GB	$128
Storage bill for container in 2 regions - West US & East US	2 * 0.5 TB (or 1,024 GB)	$0.25/GB	$256

Improve throughput utilization on a per region-basis

If you have inefficient utilization, for example, one or more under-utilized read regions you can take steps to

make the maximum use of the RUs in read regions by using change feed from the read-region or move it to another secondary if over-utilized. You will need to ensure you optimize provisioned throughput (RUs) in the write region first. Writes cost more than reads unless very large queries so maintaining even utilization can be challenging. Overall, monitor the consumed throughput in your regions and add or remove regions on demand to scale your read and write throughput, making to sure understand the impact to latency for any apps that are deployed in the same region.

Next steps

Next you can proceed to learn more about cost optimization in Azure Cosmos DB with the following articles:

- Learn more about Optimizing for development and testing
- Learn more about Understanding your Azure Cosmos DB bill
- Learn more about Optimizing throughput cost
- Learn more about Optimizing storage cost
- Learn more about Optimizing the cost of reads and writes
- Learn more about Optimizing the cost of queries

Optimize development and testing cost in Azure Cosmos DB

11/2/2020 • 5 minutes to read • Edit Online

APPLIES TO: SQL API Cassandra API Gremlin API Table API Azure Cosmos DB API for MongoDB

This article describes the different options to use Azure Cosmos DB for development and testing for free of cost, as well as techniques to optimize cost in development or test accounts.

Azure Cosmos DB emulator (locally downloadable version)

Azure Cosmos DB emulator is a local downloadable version that mimics the Azure Cosmos DB cloud service. You can write and test code that uses the Azure Cosmos DB APIs even if you have no network connection and without incurring any costs. Azure Cosmos DB emulator provides a local environment for development purposes with high fidelity to the cloud service. You can develop and test your application locally, without creating an Azure subscription. When you're ready to deploy your application to the cloud, update the connection string to connect to the Azure Cosmos DB endpoint in the cloud, no other modifications are needed. You can also set up a CI/CD pipeline with the Azure Cosmos DB emulator build task in Azure DevOps to run tests. You can get started by visiting the Azure Cosmos DB emulator article.

Azure Cosmos DB free tier

Azure Cosmos DB free tier makes it easy to get started, develop and test your applications, or even run small production workloads for free. When free tier is enabled on an account, you'll get the first 400 RU/s and 5 GB of storage in the account free. You can also create a shared throughput database with 25 containers that share 400 RU/s at the database level, all covered by free tier (limit 5 shared throughput databases in a free tier account). When using the free tier, if you provision a shared database with a minimum throughput of 400 RU/s, all the containers within that database can share the throughput. Any new databases with shared throughput or containers with dedicated throughput are billed at the regular pricing.

> **NOTE**
>
> Free tier is only available in provisioned throughput mode.

Free tier lasts indefinitely for the lifetime of the account and comes with all the benefits and features of a regular Azure Cosmos DB account, including unlimited storage and throughput (RU/s), SLAs, high availability, turnkey global distribution in all Azure regions, and more. You can have up to one free tier account per Azure subscription and must opt-in when creating the account. To get started, create a new account in Azure portal with free tier enabled or use an ARM Template. See the pricing page for more details.

Try Azure Cosmos DB for free

Try Azure Cosmos DB for free is a free of charge experience that allows you to experiment with Azure Cosmos DB in the cloud without signing up for an Azure account or using your credit card. The Try Azure Cosmos DB accounts are available for a limited time, currently 30 days. You can renew them at any time. Try Azure Cosmos DB accounts makes it easy to evaluate Azure Cosmos DB, build and test an application or use the Quickstarts or tutorials. You can also create a demo, perform unit testing, or even create a multi-region account and run an app

on it without incurring any costs. In a Try Azure Cosmos DB account, you can have one shared throughput database with a maximum of 25 containers and 20,000 RU/s of throughput, or one container with up to 5000 RU/s. To get started, see Try Azure Cosmos DB for free page.

Azure free account

Azure Cosmos DB is included in the Azure free account, which offers Azure credits and resources for free for a certain time period. Specifically for Azure Cosmos DB, this free account offers 5-GB storage and 400 RUs of provisioned throughput for the entire year. This experience enables any developer to easily test the features of Azure Cosmos DB or integrate it with other Azure services at zero cost. With Azure free account, you get a $200 credit to spend in the first 30 days. You won't be charged, even if you start using the services until you choose to upgrade. To get started, visit Azure free account page.

Azure Cosmos DB serverless

Azure Cosmos DB serverless lets you use your Azure Cosmos account in a consumption-based fashion where you are only charged for the Request Units consumed by your database operations and the storage consumed by your data. There is no minimum charge involved when using Azure Cosmos DB in serverless mode. Because it eliminates the concept of provisioned capacity, it is best suited for development or testing activities specifically when your database is idle most of the time.

Use shared throughput databases

In a shared throughput database, all containers inside the database share the provisioned throughput (RU/s) of the database. For example, if you provision a database with 400 RU/s and have four containers, all four containers will share the 400 RU/s. In a development or testing environment, where each container may be be accessed less frequently and thus require lower than the minimum of 400 RU/s, putting containers in a shared throughput database can help optimize cost.

For example, suppose your development or test account has four containers. If you create four containers with dedicated throughput (minimum of 400 RU/s), your total RU/s will be 1600 RU/s. In contrast, if you create a shared throughput database (minimum 400 RU/s) and put your containers there, your total RU/s will be just 400 RU/s. In general, shared throughput databases are great for scenarios where you don't need guaranteed throughput on any individual container. Learn more about shared throughput databases.

Next steps

You can get started with using the emulator or the free Azure Cosmos DB accounts with the following articles:

- Learn more about Understanding your Azure Cosmos DB bill
- Learn more about Azure Cosmos DB serverless
- Learn more about Optimizing throughput cost
- Learn more about Optimizing storage cost
- Learn more about Optimizing the cost of reads and writes
- Learn more about Optimizing the cost of queries
- Learn more about Optimizing the cost of multi-region Azure Cosmos accounts

Optimize cost with reserved capacity in Azure Cosmos DB

11/2/2020 • 6 minutes to read • Edit Online

APPLIES TO: SQL API Cassandra API Gremlin API Table API Azure Cosmos DB API for MongoDB

Azure Cosmos DB reserved capacity helps you save money by committing to a reservation for Azure Cosmos DB resources for either one year or three years. With Azure Cosmos DB reserved capacity, you can get a discount on the throughput provisioned for Cosmos DB resources. Examples of resources are databases and containers (tables, collections, and graphs).

Azure Cosmos DB reserved capacity can significantly reduce your Cosmos DB costs—up to 65 percent on regular prices with a one-year or three-year upfront commitment. Reserved capacity provides a billing discount and doesn't affect the runtime state of your Azure Cosmos DB resources.

Azure Cosmos DB reserved capacity covers throughput provisioned for your resources. It doesn't cover the storage and networking charges. As soon as you buy a reservation, the throughput charges that match the reservation attributes are no longer charged at the pay-as-you go rates. For more information on reservations, see the Azure reservations article.

You can buy Azure Cosmos DB reserved capacity from the Azure portal. Pay for the reservation up front or with monthly payments. To buy reserved capacity:

- You must be in the Owner role for at least one Enterprise or individual subscription with pay-as-you-go rates.
- For Enterprise subscriptions, **Add Reserved Instances** must be enabled in the EA portal. Or, if that setting is disabled, you must be an EA Admin on the subscription.
- For the Cloud Solution Provider (CSP) program, only admin agents or sales agents can buy Azure Cosmos DB reserved capacity.

Determine the required throughput before purchase

The size of the reserved capacity purchase should be based on the total amount of throughput that the existing or soon-to-be-deployed Azure Cosmos DB resources will use on an hourly basis. For example: Purchase 30,000 RU/s reserved capacity if that's your consistent hourly usage pattern. In this example, any provisioned throughput above 30,000 RU/s will be billed using your Pay-as-you-go rate. If provisioned throughput is below 30,000 RU/s in an hour, then the extra reserved capacity for that hour will be wasted.

We calculate purchase recommendations based on your hourly usage pattern. Usage over last 7, 30 and 60 days is analyzed, and reserved capacity purchase that maximizes your savings is recommended. You can view recommended reservation sizes in the Azure portal using the following steps:

1. Sign in to the Azure portal.
2. Select **All services** > **Reservations** > **Add**.
3. From the **Purchase reservations** pane, choose **Azure Cosmos DB**.
4. Select the **Recommended** tab to view recommended reservations:

You can filter recommendations by the following attributes:

- **Term** (1 year or 3 years)

- **Billing frequency** (Monthly or Upfront)
- **Throughput Type** (RU/s vs multi-region write RU/s)

Additionally, you can scope recommendations to be within a single resource group, single subscription, or your entire Azure enrollment.

Here's an example recommendation:

This recommendation to purchase a 30,000 RU/s reservation indicates that, among 3 year reservations, a 30,000 RU/s reservation size will maximize savings. In this case, the recommendation is calculated based on the past 30 days of Azure Cosmos DB usage. If this customer expects that the past 30 days of Azure Cosmos DB usage is representative of future use, they would maximize savings by purchasing a 30,000 RU/s reservation.

Buy Azure Cosmos DB reserved capacity

1. Sign in to the Azure portal.
2. Select **All services** > **Reservations** > **Add**.
3. From the **Purchase reservations** pane, choose **Azure Cosmos DB** to buy a new reservation.
4. Fill in the required fields as described in the following table:

FIELD	DESCRIPTION
Scope	Option that controls how many subscriptions can use the billing benefit associated with the reservation. It also controls how the reservation is applied to specific subscriptions. If you select **Shared**, the reservation discount is applied to Azure Cosmos DB instances that run in any subscription within your billing context. The billing context is based on how you signed up for Azure. For enterprise customers, the shared scope is the enrollment and includes all subscriptions within the enrollment. For pay-as-you-go customers, the shared scope is all individual subscriptions with pay-as-you-go rates created by the account administrator. If you select **Single subscription**, the reservation discount is applied to Azure Cosmos DB instances in the selected subscription. If you select **Single resource group**, the reservation discount is applied to Azure Cosmos DB instances in the selected subscription and the selected resource group within that subscription. You can change the reservation scope after you buy the reserved capacity.

FIELD	DESCRIPTION
Subscription	Subscription that's used to pay for the Azure Cosmos DB reserved capacity. The payment method on the selected subscription is used in charging the costs. The subscription must be one of the following types: Enterprise Agreement (offer numbers: MS-AZR-0017P or MS-AZR-0148P): For an Enterprise subscription, the charges are deducted from the enrollment's monetary commitment balance or charged as overage. Individual subscription with pay-as-you-go rates (offer numbers: MS-AZR-0003P or MS-AZR-0023P): For an individual subscription with pay-as-you-go rates, the charges are billed to the credit card or invoice payment method on the subscription.
Resource Group	Resource group to which the reserved capacity discount is applied.
Term	One year or three years.
Throughput Type	Throughput is provisioned as request units. You can buy a reservation for the provisioned throughput for both setups - single region writes as well as multiple region writes. The throughput type has two values to choose from: 100 RU/s per hour and 100 multi-region writes RU/s per hour.
Reserved Capacity Units	The amount of throughput that you want to reserve. You can calculate this value by determining the throughput needed for all your Cosmos DB resources (for example, databases or containers) per region. You then multiply it by the number of regions that you'll associate with your Cosmos database. For example: If you have five regions with 1 million RU/sec in every region, select 5 million RU/sec for the reservation capacity purchase.

5. After you fill the form, the price required to purchase the reserved capacity is calculated. The output also shows the percentage of discount you get with the chosen options. Next click **Select**

6. In the **Purchase reservations** pane, review the discount and the price of the reservation. This reservation price applies to Azure Cosmos DB resources with throughput provisioned across all regions.

7. Select **Review + buy** and then **buy now**. You see the following page when the purchase is successful:

After you buy a reservation, it's applied immediately to any existing Azure Cosmos DB resources that match the terms of the reservation. If you don't have any existing Azure Cosmos DB resources, the reservation will apply when you deploy a new Cosmos DB instance that matches the terms of the reservation. In both cases, the period

of the reservation starts immediately after a successful purchase.

When your reservation expires, your Azure Cosmos DB instances continue to run and are billed at the regular pay-as-you-go rates.

Cancel, exchange, or refund reservations

You can cancel, exchange, or refund reservations with certain limitations. For more information, see Self-service exchanges and refunds for Azure Reservations.

Next steps

The reservation discount is applied automatically to the Azure Cosmos DB resources that match the reservation scope and attributes. You can update the scope of the reservation through the Azure portal, PowerShell, Azure CLI, or the API.

- To learn how reserved capacity discounts are applied to Azure Cosmos DB, see Understand the Azure reservation discount.
- To learn more about Azure reservations, see the following articles:
  - What are Azure reservations?
  - Manage Azure reservations
  - Understand reservation usage for your Enterprise enrollment
  - Understand reservation usage for your Pay-As-You-Go subscription
  - Azure reservations in the Partner Center CSP program

Need help? Contact us.

If you have questions or need help, create a support request.

Change feed in Azure Cosmos DB

12/4/2020 • 5 minutes to read • Edit Online

APPLIES TO: SQL API Cassandra API Gremlin API Table API Azure Cosmos DB API for MongoDB

Change feed in Azure Cosmos DB is a persistent record of changes to a container in the order they occur. Change feed support in Azure Cosmos DB works by listening to an Azure Cosmos container for any changes. It then outputs the sorted list of documents that were changed in the order in which they were modified. The persisted changes can be processed asynchronously and incrementally, and the output can be distributed across one or more consumers for parallel processing.

Learn more about change feed design patterns.

Supported APIs and client SDKs

This feature is currently supported by the following Azure Cosmos DB APIs and client SDKs.

CLIENT DRIVERS	SQL API	AZURE COSMOS DB'S API FOR CASSANDRA	AZURE COSMOS DB'S API FOR MONGODB	GREMLIN API	TABLE API
.NET	Yes	Yes	Yes	Yes	No
Java	Yes	Yes	Yes	Yes	No
Python	Yes	Yes	Yes	Yes	No
Node/JS	Yes	Yes	Yes	Yes	No

Change feed and different operations

Today, you see all inserts and updates in the change feed. You can't filter the change feed for a specific type of operation. One possible alternative, is to add a "soft marker" on the item for updates and filter based on that when processing items in the change feed.

Currently change feed doesn't log deletes. Similar to the previous example, you can add a soft marker on the items that are being deleted. For example, you can add an attribute in the item called "deleted" and set it to "true" and set a TTL on the item, so that it can be automatically deleted. You can read the change feed for historic items (the most recent change corresponding to the item, it doesn't include the intermediate changes), for example, items that were added five years ago. You can read the change feed as far back as the origin of your container but if an item is deleted, it will be removed from the change feed.

Sort order of items in change feed

Change feed items come in the order of their modification time. This sort order is guaranteed per logical partition key.

Consistency level

While consuming the change feed in an Eventual consistency level, there could be duplicate events in-between subsequent change feed read operations (the last event of one read operation appears as the first of the next).

Change feed in multi-region Azure Cosmos accounts

In a multi-region Azure Cosmos account, if a write-region fails over, change feed will work across the manual failover operation and it will be contiguous.

Change feed and Time to Live (TTL)

If a TTL (Time to Live) property is set on an item to -1, change feed will persist forever. If the data is not deleted, it will remain in the change feed.

Change feed and _etag, _lsn or _ts

The _etag format is internal and you should not take dependency on it, because it can change anytime. _ts is a modification or a creation timestamp. You can use _ts for chronological comparison. _lsn is a batch ID that is added for change feed only; it represents the transaction ID. Many items may have same _lsn. ETag on FeedResponse is different from the _etag you see on the item. _etag is an internal identifier and it is used for concurrency control. The _etag property tells about the version of the item, whereas the ETag property is used for sequencing the feed.

Working with change feed

You can work with change feed using the following options:

- Using change feed with Azure Functions
- Using change feed with change feed processor

Change feed is available for each logical partition key within the container, and it can be distributed across one or more consumers for parallel processing as shown in the image below.

Features of change feed

- Change feed is enabled by default for all Azure Cosmos accounts.
- You can use your provisioned throughput to read from the change feed, just like any other Azure Cosmos DB operation, in any of the regions associated with your Azure Cosmos database.
- The change feed includes inserts and update operations made to items within the container. You can capture deletes by setting a "soft-delete" flag within your items (for example, documents) in place of deletes. Alternatively, you can set a finite expiration period for your items with the TTL capability. For example, 24 hours and use the value of that property to capture deletes. With this solution, you have to process the changes within a shorter time interval than the TTL expiration period.

- Each change to an item appears exactly once in the change feed, and the clients must manage the checkpointing logic. If you want to avoid the complexity of managing checkpoints, the change feed processor provides automatic checkpointing and "at least once" semantics. See using change feed with change feed processor.
- Only the most recent change for a given item is included in the change log. Intermediate changes may not be available.
- The change feed is sorted by the order of modification within each logical partition key value. There is no guaranteed order across the partition key values.
- Changes can be synchronized from any point-in-time, that is there is no fixed data retention period for which changes are available.
- Changes are available in parallel for all logical partition keys of an Azure Cosmos container. This capability allows changes from large containers to be processed in parallel by multiple consumers.
- Applications can request multiple change feeds on the same container simultaneously. ChangeFeedOptions.StartTime can be used to provide an initial starting point. For example, to find the continuation token corresponding to a given clock time. The ContinuationToken, if specified, takes precedence over the StartTime and StartFromBeginning values. The precision of ChangeFeedOptions.StartTime is ~5 secs.

Change feed in APIs for Cassandra and MongoDB

Change feed functionality is surfaced as change stream in MongoDB API and Query with predicate in Cassandra API. To learn more about the implementation details for MongoDB API, see the Change streams in the Azure Cosmos DB API for MongoDB.

Native Apache Cassandra provides change data capture (CDC), a mechanism to flag specific tables for archival as well as rejecting writes to those tables once a configurable size-on-disk for the CDC log is reached. The change feed feature in Azure Cosmos DB API for Cassandra enhances the ability to query the changes with predicate via CQL. To learn more about the implementation details, see Change feed in the Azure Cosmos DB API for Cassandra.

Next steps

You can now proceed to learn more about change feed in the following articles:

- Options to read change feed
- Using change feed with Azure Functions
- Using change feed processor

Built-in Jupyter Notebooks support in Azure Cosmos DB (preview)

11/2/2020 • 3 minutes to read • Edit Online

APPLIES TO: SQL API Cassandra API Gremlin API Table API Azure Cosmos DB API for MongoDB

Jupyter Notebook is an open-source web application that allows you to create and share documents that contain live code, equations, visualizations, and narrative text.

Azure Cosmos DB built-in Jupyter Notebooks are directly integrated into the Azure portal and your Azure Cosmos DB accounts, making them convenient and easy to use. Developers, data scientists, engineers, and analysts can use the familiar Jupyter Notebooks experience to do data exploration, data cleaning, data transformations, numerical simulations, statistical modeling, data visualization, and machine learning.

Azure Cosmos DB supports both C# and Python notebooks for all APIs, including Core (SQL), Cassandra, Gremlin, Table, and API for MongoDB. Inside the notebook, you can take advantage of built-in commands and features that make it easy to create Azure Cosmos DB resources, upload data, and query and visualize your data in Azure Cosmos DB.

Benefits of Jupyter Notebooks

Jupyter Notebooks were originally developed for data science applications written in Python and R. However, they can be used in various ways for different kinds of projects, including:

Data visualization: Jupyter Notebooks allow you to visualize data in the form of a shared notebook that renders a data set as a graphic. You can create visualizations, make interactive changes to the shared code and data set, and share the results.

Code sharing: Services like GitHub provide ways to share code, but they're largely non-interactive. With a Jupyter Notebook, you can view code, execute it, and display the results directly in the Azure portal.

Live interactions with code: Code in a Jupyter Notebook is dynamic; you can edit it and run the updates incrementally in real time. You can also embed user controls (for example, sliders or text input fields) that are used as input sources for code, demos, or Proof of Concepts (POCs).

Documentation of code samples and outcomes of data exploration: If you have a piece of code and you want to explain line-by-line how it works, you can embed it in a Jupyter Notebook. You can add interactivity along with the documentation at the same time.

Built-in commands for Azure Cosmos DB: Azure Cosmos DB's built-in magic commands make it easy to interact with your account. You can use commands like %%upload and %%sql to upload data into a container and query it using SQL API syntax. You don't need to write additional custom code.

All in one place environment: Jupyter Notebooks combine code, rich text, images, videos, animations, mathematical equations, plots, maps, interactive figures, widgets, and graphical user interfaces into a single document.

Components of a Jupyter Notebook

Jupyter Notebooks can include several types of components, each organized into discrete blocks or cells:

Text and HTML: Plain text, or text annotated in the markdown syntax to generate HTML, can be inserted into the document at any point. CSS styling can also be included inline or added to the template used to generate the notebook.

Code and output: Jupyter Notebooks support Python and C# code. The results of the executed code appear

immediately after the code blocks, and the code blocks can be executed multiple times in any order you like.

Visualizations: You can generate graphics and charts from the code by using modules like Matplotlib, Plotly, Bokeh, and others. Similar to the output, these visualizations appear inline next to the code that generates them. Similar to the output, these visualizations appear inline next to the code that generates them.

Multimedia: Because Jupyter Notebooks are built on web technology, they can display all the types of multimedia supported by a web page. You can include them in a notebook as HTML elements, or you can generate them programmatically by using the `IPython.display` module.

Data: You can import the data from Azure Cosmos containers or the results of queries into a Jupyter Notebook programmatically. Use built-in magic commands to upload or query data in Azure Cosmos DB.

Next steps

To get started with built-in Jupyter Notebooks in Azure Cosmos DB, see the following articles:

- Enable notebooks in an Azure Cosmos account
- Use Python notebook features and commands
- Use C# notebook features and commands

Azure Cosmos DB service quotas

12/4/2020 • 11 minutes to read • Edit Online

APPLIES TO: SQL API Cassandra API Gremlin API Table API Azure Cosmos DB API for MongoDB

This article provides an overview of the default quotas offered to different resources in the Azure Cosmos DB.

Storage and database operations

After you create an Azure Cosmos account under your subscription, you can manage data in your account by creating databases, containers, and items.

Provisioned throughput

You can provision throughput at a container-level or a database-level in terms of request units (RU/s or RUs). The following table lists the limits for storage and throughput per container/database.

RESOURCE	DEFAULT LIMIT
Maximum RUs per container (dedicated throughput provisioned mode)	1,000,000 by default. You can increase it by filing an Azure support ticket
Maximum RUs per database (shared throughput provisioned mode)	1,000,000 by default. You can increase it by filing an Azure support ticket
Maximum RUs per (logical) partition	10,000
Maximum storage across all items per (logical) partition	20 GB
Maximum number of distinct (logical) partition keys	Unlimited
Maximum storage per container	Unlimited
Maximum storage per database	Unlimited
Maximum attachment size per Account (Attachment feature is being deprecated)	2 GB
Minimum RU/s required per 1 GB	10 RU/s **Note:** if your container or database contains more than 1 TB of data, your account may be eligible to our "high storage / low throughput" program

> **NOTE**
>
> To learn about best practices for managing workloads that have partition keys requiring higher limits for storage or throughput, see Create a synthetic partition key.

Minimum throughput limits

A Cosmos container (or shared throughput database) must have a minimum throughput of 400 RU/s. As the

container grows, Cosmos DB requires a minimum throughput to ensure the database or container has sufficient resource for its operations.

The current and minimum throughput of a container or a database can be retrieved from the Azure portal or the SDKs. For more information, see Provision throughput on containers and databases.

The actual minimum RU/s may vary depending on your account configuration. You can use Azure Monitor metrics to view the history of provisioned throughput (RU/s) and storage on a resource.

Minimum throughput on container

To estimate the minimum throughput required of a container with manual throughput, find the maximum of:

- 400 RU/s
- Current storage in GB * 10 RU/s
- Highest RU/s provisioned on the container / 100

Example: Suppose you have a container provisioned with 400 RU/s and 0 GB storage. You increase the throughput to 50,000 RU/s and import 20 GB of data. The minimum RU/s is now `MAX(400, 20 * 10 RU/s per GB, 50,000 RU/s / 100)` = 500 RU/s. Over time, the storage grows to 200 GB. The minimum RU/s is now `MAX(400, 200 * 10 RU/s per GB, 50,000 / 100)` = 2000 RU/s.

Note: if your container or database contains more than 1 TB of data, your account may be eligible to our "high storage / low throughput" program.

Minimum throughput on shared throughput database

To estimate the minimum throughput required of a shared throughput database with manual throughput, find the maximum of:

- 400 RU/s
- Current storage in GB * 10 RU/s
- Highest RU/s provisioned on the database / 100
- 400 + MAX(Container count - 25, 0) * 100 RU/s

Example: Suppose you have a database provisioned with 400 RU/s, 15 GB of storage, and 10 containers. The minimum RU/s is `MAX(400, 15 * 10 RU/s per GB, 400 / 100, 400 + 0 )` = 400 RU/s. If there were 30 containers in the database, the minimum RU/s would be `400 + MAX(30 - 5, 0) * 100 RU/s` = 900 RU/s.

Note: if your container or database contains more than 1 TB of data, your account may be eligible to our "high storage / low throughput" program.

In summary, here are the minimum provisioned RU limits.

RESOURCE	DEFAULT LIMIT
Minimum RUs per container (dedicated throughput provisioned mode)	400
Minimum RUs per database (shared throughput provisioned mode)	400 RU/s for first 25 containers. Additional 100 RU/s for each container afterward.

Cosmos DB supports programmatic scaling of throughput (RU/s) per container or database via the SDKs or portal.

Depending on the current RU/s provisioned and resource settings, each resource can scale synchronously and immediately between the minimum RU/s to up to 100x the minimum RU/s. If the requested throughput value is outside the range, scaling is performed asynchronously. Asynchronous scaling may take minutes to hours to complete depending on the requested throughput and data storage size in the container.

Serverless

Serverless lets you use your Azure Cosmos DB resources in a consumption-based fashion. The following table lists the limits for storage and throughput burstability per container/database.

RESOURCE	LIMIT
Maximum RU/s per (logical) partition	5,000
Maximum storage across all items per (logical) partition	20 GB
Maximum number of distinct (logical) partition keys	Unlimited
Maximum storage per container	50 GB

Control plane operations

You can provision and manage your Azure Cosmos account using the Azure portal, Azure PowerShell, Azure CLI, and Azure Resource Manager templates. The following table lists the limits per subscription, account, and number of operations.

RESOURCE	DEFAULT LIMIT
Maximum database accounts per subscription	50 by default. You can increase it by filing an Azure support ticket
Maximum number of regional failovers	1/hour by default. You can increase it by filing an Azure support ticket

> **NOTE**
> Regional failovers only apply to single region writes accounts. Multi-region write accounts do not require or have any limits on changing the write region.

Cosmos DB automatically takes backups of your data at regular intervals. For details on backup retention intervals and windows, see Online backup and on-demand data restore in Azure Cosmos DB.

Per-account limits

Provisioned throughput

RESOURCE	DEFAULT LIMIT
Maximum number of databases	Unlimited
Maximum number of containers per database with shared throughput	25
Maximum number of containers per database or account with dedicated throughput	unlimited
Maximum number of regions	No limit (All Azure regions)

Serverless

RESOURCE	LIMIT
Maximum number of databases	Unlimited
Maximum number of containers per account	100
Maximum number of regions	1 (Any Azure region)

Per-container limits

Depending on which API you use, an Azure Cosmos container can represent either a collection, a table, or graph. Containers support configurations for unique key constraints, stored procedures, triggers, and UDFs, and indexing policy. The following table lists the limits specific to configurations within a container.

RESOURCE	DEFAULT LIMIT
Maximum length of database or container name	255
Maximum stored procedures per container	100 *
Maximum UDFs per container	25 *
Maximum number of paths in indexing policy	100 *
Maximum number of unique keys per container	10 *
Maximum number of paths per unique key constraint	16 *
Maximum TTL value	2147483647

* You can increase any of these per-container limits by creating an Azure Support request.

Per-item limits

Depending on which API you use, an Azure Cosmos item can represent either a document in a collection, a row in a table, or a node or edge in a graph. The following table shows the limits per item in Cosmos DB.

RESOURCE	DEFAULT LIMIT
Maximum size of an item	2 MB (UTF-8 length of JSON representation)
Maximum length of partition key value	2048 bytes
Maximum length of ID value	1023 bytes
Maximum number of properties per item	No practical limit
Maximum length of property name	No practical limit
Maximum length of property value	No practical limit
Maximum length of string property value	No practical limit

RESOURCE	DEFAULT LIMIT
Maximum length of numeric property value	IEEE754 double-precision 64-bit
Maximum level of nesting for embedded objects / arrays	128
Maximum TTL value	2147483647

There are no restrictions on the item payloads like number of properties and nesting depth, except for the length restrictions on partition key and ID values, and the overall size restriction of 2 MB. You may have to configure indexing policy for containers with large or complex item structures to reduce RU consumption. See Modeling items in Cosmos DB for a real-world example, and patterns to manage large items.

Per-request limits

Azure Cosmos DB supports CRUD and query operations against resources like containers, items, and databases. It also supports transactional batch requests against multiple items with the same partition key in a container.

RESOURCE	DEFAULT LIMIT
Maximum execution time for a single operation (like a stored procedure execution or a single query page retrieval)	5 sec
Maximum request size (for example, stored procedure, CRUD)	2 MB
Maximum response size (for example, paginated query)	4 MB
Maximum number of operations in a transactional batch	100

Once an operation like query reaches the execution timeout or response size limit, it returns a page of results and a continuation token to the client to resume execution. There is no practical limit on the duration a single query can run across pages/continuations.

Cosmos DB uses HMAC for authorization. You can use either a primary key, or a resource tokens for fine-grained access control to resources like containers, partition keys, or items. The following table lists limits for authorization tokens in Cosmos DB.

RESOURCE	DEFAULT LIMIT
Maximum primary token expiry time	15 min
Minimum resource token expiry time	10 min
Maximum resource token expiry time	24 h by default. You can increase it by filing an Azure support ticket
Maximum clock skew for token authorization	15 min

Cosmos DB supports execution of triggers during writes. The service supports a maximum of one pre-trigger and one post-trigger per write operation.

Metadata request limits

Azure Cosmos DB maintains system metadata for each account. This metadata allows you to enumerate

collections, databases, other Azure Cosmos DB resources, and their configurations for free of charge.

RESOURCE	DEFAULT LIMIT
Maximum collection create rate per minute	100
Maximum Database create rate per minute	100
Maximum provisioned throughput update rate per minute	5

Limits for autoscale provisioned throughput

See the Autoscale article and FAQ for more detailed explanation of the throughput and storage limits with autoscale.

RESOURCE	DEFAULT LIMIT
Maximum RU/s the system can scale to	`Tmax`, the autoscale max RU/s set by user
Minimum RU/s the system can scale to	`0.1 * Tmax`
Current RU/s the system is scaled to	`0.1*Tmax <= T <= Tmax`, based on usage
Minimum billable RU/s per hour	`0.1 * Tmax` Billing is done on a per-hour basis, where you are billed for the highest RU/s the system scaled to in the hour, or `0.1*Tmax`, whichever is higher.
Minimum autoscale max RU/s for a container	`MAX(4000, highest max RU/s ever provisioned / 10, current storage in GB * 100)` rounded to nearest 1000 RU/s
Minimum autoscale max RU/s for a database	`MAX(4000, highest max RU/s ever provisioned / 10, current storage in GB * 100, 4000 + (MAX(Container count - 25, 0) * 1000))` , rounded to nearest 1000 RU/s. Note if your database has more than 25 containers, the system increments the minimum autoscale max RU/s by 1000 RU/s per additional container. For example, if you have 30 containers, the lowest autoscale maximum RU/s you can set is 9000 RU/s (scales between 900 - 9000 RU/s).

SQL query limits

Cosmos DB supports querying items using SQL. The following table describes restrictions in query statements, for example in terms of number of clauses or query length.

RESOURCE	DEFAULT LIMIT
Maximum length of SQL query	256 KB
Maximum JOINs per query	5 *
Maximum UDFs per query	10 *

RESOURCE	DEFAULT LIMIT
Maximum points per polygon	4096
Maximum included paths per container	500
Maximum excluded paths per container	500
Maximum properties in a composite index	8

* You can increase any of these SQL query limits by creating an Azure Support request.

MongoDB API-specific limits

Cosmos DB supports the MongoDB wire protocol for applications written against MongoDB. You can find the supported commands and protocol versions at Supported MongoDB features and syntax.

The following table lists the limits specific to MongoDB feature support. Other service limits mentioned for the SQL (core) API also apply to the MongoDB API.

RESOURCE	DEFAULT LIMIT
Maximum MongoDB query memory size (This limitation is only for 3.2 server version)	40 MB
Maximum execution time for MongoDB operations	30s
Idle connection timeout for server side connection closure*	30 minutes

* We recommend that client applications set the idle connection timeout in the driver settings to 2-3 minutes because the default timeout for Azure LoadBalancer is 4 minutes. This timeout will ensure that idle connections are not closed by an intermediate load balancer between the client machine and Azure Cosmos DB.

Try Cosmos DB Free limits

The following table lists the limits for the Try Azure Cosmos DB for Free trial.

RESOURCE	DEFAULT LIMIT
Duration of the trial	30 days (a new trial can be requested after expiration) After expiration, the information stored is deleted.
Maximum containers per subscription (SQL, Gremlin, Table API)	1
Maximum containers per subscription (MongoDB API)	3
Maximum throughput per container	5000
Maximum throughput per shared-throughput database	20000
Maximum total storage per account	10 GB

Try Cosmos DB supports global distribution in only the Central US, North Europe, and Southeast Asia regions.

Azure support tickets can't be created for Try Azure Cosmos DB accounts. However, support is provided for subscribers with existing support plans.

Free tier account limits

The following table lists the limits for Azure Cosmos DB free tier accounts.

RESOURCE	DEFAULT LIMIT
Number of free tier accounts per Azure subscription	1
Duration of free-tier discount	Lifetime of the account. Must opt-in during account creation.
Maximum RU/s for free	400 RU/s
Maximum storage for free	5 GB
Maximum number of shared throughput databases	5
Maximum number of containers in a shared throughput database	25 In free tier accounts, the minimum RU/s for a shared throughput database with up to 25 containers is 400 RU/s.

In addition to the above, the Per-account limits also apply to free tier accounts.

Next steps

Read more about Cosmos DB's core concepts global distribution and partitioning and provisioned throughput.

Get started with Azure Cosmos DB with one of our quickstarts:

- Get started with Azure Cosmos DB SQL API
- Get started with Azure Cosmos DB's API for MongoDB
- Get started with Azure Cosmos DB Cassandra API
- Get started with Azure Cosmos DB Gremlin API
- Get started with Azure Cosmos DB Table API

Try Azure Cosmos DB for free

Compliance in Azure Cosmos DB

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: SQL API Cassandra API Gremlin API Table API Azure Cosmos DB API for MongoDB

Azure Cosmos DB is available in all Azure regions. Microsoft makes five distinct Azure cloud environments available to customers:

- **Azure public** cloud, which is available globally.
- **Azure China 21Vianet** is available through a unique partnership between Microsoft and 21Vianet, one of the country/region's largest internet providers.
- **Azure Germany** provides services under a data trustee model, which ensures that customer data remains in Germany under the control of T-Systems International GmbH, a subsidiary of Deutsche Telecom, acting as the German data trustee.
- **Azure Government** is available in four regions in the United States to US government agencies and their partners.
- **Azure Government for Department of Defense(DoD)** is available in two regions in the United States to the US Department of Defense.

To help customers meet their own compliance obligations across regulated industries and markets worldwide, Azure maintains the largest compliance portfolio in the industry in terms of both breadth (total number of offerings) and depth (number of customer-facing services in assessment scope). Azure compliance offerings are grouped into four segments - globally applicable, US Government, industry specific, and region or country/region specific. Compliance offerings are based on various types of assurances, including formal certifications, attestations, validations, authorizations, and assessments produced by independent third-party auditing firms, as well as contractual amendments, self-assessments, and customer guidance documents produced by Microsoft.

Azure Cosmos DB certifications

Azure Cosmos DB is continually expanding its certification coverage. Currently, Azure Cosmos DB is certified with the following certificates:

GLOBALLY APPLICABLE	US GOVERNMENT	INDUSTRY SPECIFIC	REGION OR COUNTRY SPECIFIC
CSA STAR Certification	DoD SRG Level 2	HIPAA BAA	Australia IRAP
CSA STAR Attestation	FedRAMP Moderate	HITRUST	Germany C5
ISO 20000-1:2011	GxP (FDA 21 CFR Part 11)	PCI DSS	Singapore MTCS Level 3
ISO 22301:2012			Spain ENS High
ISO 27001:2013			
ISO 27017:2015			

GLOBALLY APPLICABLE	US GOVERNMENT	INDUSTRY SPECIFIC	REGION OR COUNTRY SPECIFIC
ISO 27018:2014			
ISO 9001:2015			
ISO 27701:2019			
SOC 1, 2, 3			

To learn more about each of these compliance offerings and how they benefit you, see Microsoft Azure Compliance offerings, Appendix A & B.

The following table lists the certifications supported by Azure Cosmos DB in Azure Government:

GLOBALLY APPLICABLE	US GOVERNMENT	INDUSTRY SPECIFIC
CSA STAR Certification	CJIS	HIPAA BAA
CSA STAR Attestation	DoD SRG Level 2	HITRUST
ISO 20000-1:2011	DoD SRG Level 4	PCI DSS
ISO 9001:2012	DoD SRG Level 5	
ISO 27001:2013	FedRAMP High	
ISO 9001:2015	IRS 1075	
ISO 27017:2014	NIST CSF	
ISO 27018:2015	NIST SP 800-171	
SOC 1, 2, 3		

Next steps

To learn more about Azure compliance certifications, see the following articles:

- To find out the latest compliance certifications for Azure Cosmos DB, see Microsoft Azure Compliance offerings, Appendix A & B.
- For an overview of Azure Cosmos DB security and the latest improvements, see Azure Cosmos database security.
- For recommendations to improve the security posture of your Azure Cosmos DB deployment, see the Azure Cosmos DB Security Baseline.
- For more information about Microsoft certifications, see the Azure Trust Center.
- For FedRAMP compliance information, see Azure services by FedRAMP and DoD CC SRG audit scope.
- For DoD compliance information, see DoD Compliance Offerings.

Automated recommendations for Azure Cosmos DB

11/2/2020 • 4 minutes to read • Edit Online

APPLIES TO: SQL API Cassandra API Gremlin API Table API Azure Cosmos DB API for MongoDB

All the cloud services including Azure Cosmos DB get frequent updates with new features, capabilities, and improvements. It's important for your application to keep up with the latest performance and security updates. The Azure portal offers customized recommendations that enable you to maximize the performance of your application. The Azure Cosmos DB's advisory engine continuously analyzes the usage history of your Azure Cosmos DB resources and provides recommendations based on your workload patterns. These recommendations correspond to areas like partitioning, indexing, network, security etc. These customized recommendations help you to improve the performance of your application.

View recommendations

You can view recommendations for Azure Cosmos DB in the following ways:

- One way to view the recommendations is within the notifications tab. If there are new recommendations, you will see a message bar. Sign into your Azure portal and navigate to your Azure Cosmos account. Within your Azure Cosmos account, open the **Notifications** pane and then select the **Recommendations** tab. You can select the message and view recommendations.

- You can also find the recommendations through Azure Advisor in categorized by different buckets such as cost, security, reliability, performance, and operational excellence. You can select specific subscriptions and filter by the resource type, which is **Azure Cosmos DB accounts**. When you select a specific recommendation, it displays the actions you can take to benefit your workloads.

Not all recommendations shown in the Azure Cosmos DB pane are available in the Azure Advisor and vice versa. That's because based on the type of recommendation they fit in either the Azure Advisor pane, Azure Cosmos DB pane or both.

Currently Azure Cosmos DB supports recommendations on the following areas. Each of these recommendations includes a link to the relevant section of the documentation, so it's easy for you to take the next steps.

SDK usage recommendations

In this category, the advisor detects the usage of an old version of SDKs and recommends that you upgrade to a newer version to leverage the latest bug fixes and performance improvements. Currently the following SDK-specific recommendations are available:

NAME	DESCRIPTION
Old Spark connector	Detects the usage of old versions of the Spark connector and recommends upgrading.
Old .NET SDK	Detects the usage of old versions of the .NET SDK and recommends upgrading.
Old Java SDK	Detects the usage of old versions of the Java connector and recommends upgrading.

Indexing recommendations

In this category, the advisor detects the indexing mode, indexing policy, indexed paths and recommends changing if the current configuration impacts the query performance. Currently the following indexing-specific recommendations are available:

NAME	DESCRIPTION
Lazy indexing	Detects usage of lazy indexing mode and recommends using consistent indexing mode instead. The purpose of Azure Cosmos DB's lazy indexing mode is limited and can impact the freshness of query results in some situations so consistent indexing mode is recommended.
Composite indexing	Detects the accounts where queries could benefit from composite indexes and recommend using them. Composite indexes can dramatically improve the performance and throughput consumption of some queries.
Default indexing policy with many indexed paths	Detects containers running on default indexing with many indexed paths and recommends customizing the indexing policy.
ORDER BY queries with high RU/s charge	Detects containers issuing ORDER BY queries with high RU/s charge and recommends exploring composite indexes.
MongoDB 3.6 accounts with no index and high RU/s consumption	Detects Azure Cosmos DB's API for MongoDB with 3.6 version of containers issuing queries with high RU/s charge and recommends adding indexes.

Cost optimization recommendations

In this category, the advisor detects the RU/s usage and determines that you can optimize the price by making some changes to your resources or by leveraging a different pricing model. Currently the following cost optimization-specific recommendations are available:

NAME	DESCRIPTION
Reserved capacity	Detects your RU/s utilization and recommends reserved instances to users who can benefit from it.
Inactive containers	Detects the containers that haven't been used for more than 30 days and recommends reducing the throughput for such containers or deleting them.
New subscriptions with high throughput	Detects new subscriptions with accounts spending unusually high RU/s per day and provides them a notification. This notification is specifically to bring awareness to new customers that Azure Cosmos DB operates on provisioned throughput-based model and not consumption-based model.

Migration recommendations

In this category, the advisor detects that you are using legacy features recommends migrating so that you can leverage Azure Cosmos DB's massive scalability and other benefits. Currently the following migration-specific recommendations are available:

NAME	DESCRIPTION
Non-partitioned containers	Detects fixed-size containers that are approaching their max storage limit and recommends migrating them to partitioned containers.

Query usage recommendations

In this category, the advisor detects the query execution and identifies that the query performance can be tuned with some changes. Currently the following query usage recommendations are available:

NAME	DESCRIPTION
Queries with fixed page size	Detects queries issued with a fixed page size and recommends using -1 (no limit on the page size) instead of defining a specific value. This option reduces the number of network round trips required to retrieve all results.

Next steps

- Tuning query performance in Azure Cosmos DB
- Troubleshoot query issues when using Azure Cosmos DB

Quickstart: Create an Azure Cosmos account, database, container, and items from the Azure portal

12/4/2020 • 7 minutes to read • Edit Online

APPLIES TO: SQL API

Azure Cosmos DB is Microsoft's globally distributed multi-model database service. You can use Azure Cosmos DB to quickly create and query key/value databases, document databases, and graph databases, all of which benefit from the global distribution and horizontal scale capabilities at the core of Azure Cosmos DB.

This quickstart demonstrates how to use the Azure portal to create an Azure Cosmos DB SQL API account, create a document database, and container, and add data to the container.

Prerequisites

An Azure subscription or free Azure Cosmos DB trial account

- If you don't have an Azure subscription, create a free account before you begin.
- You can Try Azure Cosmos DB for free without an Azure subscription, free of charge and commitments, or create an Azure Cosmos DB free tier account, with the first 400 RU/s and 5 GB of storage for free. You can also use the Azure Cosmos DB Emulator with a URI of `https://localhost:8081`. For the key to use with the emulator, see Authenticating requests.

Create an Azure Cosmos DB account

Go to the Azure portal to create an Azure Cosmos DB account. Search for and select **Azure Cosmos DB**.

1. Select **Add**.
2. On the **Create Azure Cosmos DB Account** page, enter the basic settings for the new Azure Cosmos account.

SETTING	VALUE	DESCRIPTION
Subscription	Subscription name	Select the Azure subscription that you want to use for this Azure Cosmos account.
Resource Group	Resource group name	Select a resource group, or select **Create new**, then enter a unique name for the new resource group.
Account Name	A unique name	Enter a name to identify your Azure Cosmos account. Because *documents.azure.com* is appended to the name that you provide to create your URI, use a unique name. The name can only contain lowercase letters, numbers, and the hyphen (-) character. It must be between 3-31 characters in length.
API	The type of account to create	Select **Core (SQL)** to create a document database and query by using SQL syntax. The API determines the type of account to create. Azure Cosmos DB provides five APIs: Core (SQL) and MongoDB for document data, Gremlin for graph data, Azure Table, and Cassandra. Currently, you must create a separate account for each API. Learn more about the SQL API.
Capacity mode	Provisioned throughput or Serverless	Select **Provisioned throughput** to create an account in provisioned throughput mode. Select **Serverless** to create an account in serverless mode.
Apply Free Tier Discount	Apply or Do not apply	With Azure Cosmos DB free tier, you will get the first 400 RU/s and 5 GB of storage for free in an account. Learn more about free tier.
Location	The region closest to your users	Select a geographic location to host your Azure Cosmos DB account. Use the location that is closest to your users to give them the fastest access to the data.

SETTING	VALUE	DESCRIPTION
Account Type	Production or Non-Production	Select **Production** if the account will be used for a production workload. Select **Non-Production** if the account will be used for non-production, e.g. development, testing, QA, or staging. This is an Azure resource tag setting that tunes the Portal experience but does not affect the underlying Azure Cosmos DB account. You can change this value anytime.
Geo-Redundancy	Enable or Disable	Enable or disable global distribution on your account by pairing your region with a pair region. You can add more regions to your account later.
Multi-region Writes	Enable or Disable	Multi-region writes capability allows you to take advantage of the provisioned throughput for your databases and containers across the globe.
Availability Zones	Enable or Disable	Availability Zones help you further improve availability and resiliency of your application.

NOTE

You can have up to one free tier Azure Cosmos DB account per Azure subscription and must opt-in when creating the account. If you do not see the option to apply the free tier discount, this means another account in the subscription has already been enabled with free tier.

NOTE

The following options are not available if you select **Serverless** as the **Capacity mode**:

- Apply Free Tier Discount
- Geo-redundancy
- Multi-region Writes

Microsoft Azure (Preview) | Search resources, services, and docs (G+/)

Home > New >

Create Azure Cosmos DB Account

Basics | Networking | Backup Policy | Encryption | Tags | Review + create

Azure Cosmos DB is a globally distributed, multi-model, fully managed database service. Try it for free, for 30 days with unlimited renewals. Go to production starting at $24/month per database, multiple containers included. Learn more

Project Details

Select the subscription to manage deployed resources and costs. Use resource groups like folders to organize and manage all your resources.

Subscription * : SubscriptionName

Resource Group * : (New) my-resource-group

Create new

Instance Details

Account Name * : Enter account name

API * : Core (SQL)

Notebooks (Preview) : On / Off

Location * : (US) West US

Capacity mode : Provisioned throughput / Serverless (Preview)

Learn more about capacity mode

With Azure Cosmos DB free tier, you will get 400 RU/s and 5 GB of storage for free in an account. You can enable free tier on up to one account per subscription. Estimated $[illegible]/month discount per account.

Apply Free Tier Discount : Apply / Do Not Apply

Account Type : Production / Non-Production

Geo-Redundancy : Enable / Disable

Multi-region Writes : Enable / Disable

Review + create | Previous | Next: Networking

1. Select **Review + create**. You can skip the **Network** and **Tags** sections.
2. Review the account settings, and then select **Create**. It takes a few minutes to create the account. Wait for the portal page to display **Your deployment is complete**.

Dashboard > Microsoft.Azure.CosmosDB-20190321000000 - Overview

Microsoft.Azure.CosmosDB-20190321000000 · Overview

Deployment

Search (Ctrl+/)

Overview
Inputs
Outputs
Template

Delete | Cancel | Redeploy | Refresh

Your deployment is complete

Go to resource

Deployment name: Microsoft.Azure.CosmosDB-20190321000000
Subscription: Contoso Subscription
Resource group: myResourceGroup

DEPLOYMENT DETAILS (Download)

Start time: 3/21/2019, 5:00:03 PM
Duration: 5 minutes 38 seconds
Correlation ID: 8e0be948-0c60-4da0-0000-000000000000

RESOURCE	TYPE	STATUS	OPERATION DETAILS
mysqlapicosmosdb	Microsoft.DocumentDb/databaseAcc...	OK	Operation details

3. Select **Go to resource** to go to the Azure Cosmos DB account page.

Add a database and a container

You can use the Data Explorer in the Azure portal to create a database and container.

1. Select **Data Explorer** from the left navigation on your Azure Cosmos DB account page, and then select **New Container**.

   You may need to scroll right to see the **Add Container** window.

2. In the **Add container** pane, enter the settings for the new container.

SETTING	SUGGESTED VALUE	DESCRIPTION
Database ID	ToDoList	Enter *ToDoList* as the name for the new database. Database names must contain from 1 through 255 characters, and they cannot contain `/, \\, #, ?`, or a trailing space. Check the **Provision database throughput** option, it allows you to share the throughput provisioned to the database across all the containers within the database. This option also helps with cost savings.
Throughput	400	Leave the throughput at 400 request units per second (RU/s). If you want to reduce latency, you can scale up the throughput later. **Note**: This setting is not available when creating a new container in a serverless account.
Container ID	Items	Enter *Items* as the name for your new container. Container IDs have the same character requirements as database names.
Partition key	/category	The sample described in this article uses */category* as the partition key.

Don't add **Unique keys** for this example. Unique keys let you add a layer of data integrity to the database by ensuring the uniqueness of one or more values per partition key. For more information, see Unique keys in Azure Cosmos DB.

3. Select **OK**. The Data Explorer displays the new database and the container that you created.

Add data to your database

Add data to your new database using Data Explorer.

1. In **Data Explorer**, expand the **ToDoList** database, and expand the **Items** container. Next, select **Items**, and then select **New Item**.

2. Add the following structure to the document on the right side of the **Documents** pane:

```
{
    "id": "1",
    "category": "personal",
    "name": "groceries",
    "description": "Pick up apples and strawberries.",
    "isComplete": false
}
```

3. Select **Save**.

4. Select **New Document** again, and create and save another document with a unique `id`, and any other properties and values you want. Your documents can have any structure, because Azure Cosmos DB doesn't impose any schema on your data.

Query your data

You can use queries in Data Explorer to retrieve and filter your data.

1. At the top of the **Items** tab in Data Explorer, review the default query `SELECT * FROM c`. This query retrieves and displays all documents from the container ordered by ID.

2. To change the query, select **Edit Filter**, replace the default query with `ORDER BY c._ts DESC`, and then select **Apply Filter**.

The modified query displays the documents in descending order based on their time stamp, so now your second document is listed first.

If you're familiar with SQL syntax, you can enter any supported SQL queries in the query predicate box. You can also use Data Explorer to create stored procedures, UDFs, and triggers for server-side business logic.

Data Explorer provides easy Azure portal access to all of the built-in programmatic data access features available in the APIs. You also use the portal to scale throughput, get keys and connection strings, and review metrics and SLAs for your Azure Cosmos DB account.

Clean up resources

When you're done with your app and Azure Cosmos DB account, you can delete the Azure resources you created

so you don't incur more charges. To delete the resources:

1. In the Azure portal Search bar, search for and select **Resource groups**.

2. From the list, select the resource group you created for this quickstart.

3. On the resource group **Overview** page, select **Delete resource group**.

4. In the next window, enter the name of the resource group to delete, and then select **Delete**.

If you wish to delete just the database and use the Azure Cosmos account in future, you can delete the database with the following steps:

- Got to your Azure Cosmos account.
- Open **Data Explorer**, right click on the database that you want to delete and select **Delete Database**.
- Enter the Database ID/database name to confirm the delete operation.

Next steps

In this quickstart, you learned how to create an Azure Cosmos DB account, create a database and container using the Data Explorer. You can now import additional data to your Azure Cosmos DB account.

Import data into Azure Cosmos DB

Quickstart: Create an Azure Cosmos DB and a container by using an ARM template

12/4/2020 • 6 minutes to read • Edit Online

APPLIES TO: SQL API

Azure Cosmos DB is Microsoft's fast NoSQL database with open APIs for any scale. You can use Azure Cosmos DB to quickly create and query key/value databases, document databases, and graph databases. This quickstart focuses on the process of deploying an Azure Resource Manager template (ARM template) to create an Azure Cosmos database and a container within that database. You can later store data in this container.

An ARM template is a JavaScript Object Notation (JSON) file that defines the infrastructure and configuration for your project. The template uses declarative syntax. In declarative syntax, you describe your intended deployment without writing the sequence of programming commands to create the deployment.

If your environment meets the prerequisites and you're familiar with using ARM templates, select the **Deploy to Azure** button. The template will open in the Azure portal.

Deploy to Azure

Prerequisites

An Azure subscription or free Azure Cosmos DB trial account

- If you don't have an Azure subscription, create a free account before you begin.
- You can Try Azure Cosmos DB for free without an Azure subscription, free of charge and commitments, or create an Azure Cosmos DB free tier account, with the first 400 RU/s and 5 GB of storage for free. You can also use the Azure Cosmos DB Emulator with a URI of `https://localhost:8081`. For the key to use with the emulator, see Authenticating requests.

Review the template

The template used in this quickstart is from Azure Quickstart Templates.

```
{
  "$schema": "https://schema.management.azure.com/schemas/2019-04-01/deploymentTemplate.json#",
  "contentVersion": "1.0.0.0",
  "parameters": {
    "accountName": {
      "type": "string",
      "defaultValue": "[concat('sql-', uniqueString(resourceGroup().id))]",
      "metadata": {
        "description": "Cosmos DB account name, max length 44 characters"
      }
    },
    "location": {
      "type": "string",
      "defaultValue": "[resourceGroup().location]",
      "metadata": {
        "description": "Location for the Cosmos DB account."
      }
    },
    "primaryRegion": {
      "type": "string",
```

```
      "metadata": {
        "description": "The primary replica region for the Cosmos DB account."
      }
    },
    "secondaryRegion": {
      "type": "string",
      "metadata": {
        "description": "The secondary replica region for the Cosmos DB account."
      }
    },
    "defaultConsistencyLevel": {
      "type": "string",
      "defaultValue": "Session",
      "allowedValues": [ "Eventual", "ConsistentPrefix", "Session", "BoundedStaleness", "Strong" ],
      "metadata": {
        "description": "The default consistency level of the Cosmos DB account."
      }
    },
    "maxStalenessPrefix": {
      "type": "int",
      "defaultValue": 100000,
      "minValue": 10,
      "maxValue": 2147483647,
      "metadata": {
        "description": "Max stale requests. Required for BoundedStaleness. Valid ranges, Single Region: 10 to
1000000. Multi Region: 100000 to 1000000."
      }
    },
    "maxIntervalInSeconds": {
      "type": "int",
      "defaultValue": 300,
      "minValue": 5,
      "maxValue": 86400,
      "metadata": {
        "description": "Max lag time (minutes). Required for BoundedStaleness. Valid ranges, Single Region: 5
to 84600. Multi Region: 300 to 86400."
      }
    },
    "automaticFailover": {
      "type": "bool",
      "defaultValue": true,
      "allowedValues": [ true, false ],
      "metadata": {
        "description": "Enable automatic failover for regions"
      }
    },
    "databaseName": {
      "type": "string",
      "defaultValue": "myDatabase",
      "metadata": {
        "description": "The name for the database"
      }
    },
    "containerName": {
      "type": "string",
      "defaultValue": "myContainer",
      "metadata": {
        "description": "The name for the container"
      }
    },
    "throughput": {
      "type": "int",
      "defaultValue": 400,
      "minValue": 400,
      "maxValue": 1000000,
      "metadata": {
        "description": "The throughput for the container"
      }
    }
```

```
    }
  },
  "variables": {
    "accountName": "[toLower(parameters('accountName'))]",
    "consistencyPolicy": {
      "Eventual": {
        "defaultConsistencyLevel": "Eventual"
      },
      "ConsistentPrefix": {
        "defaultConsistencyLevel": "ConsistentPrefix"
      },
      "Session": {
        "defaultConsistencyLevel": "Session"
      },
      "BoundedStaleness": {
        "defaultConsistencyLevel": "BoundedStaleness",
        "maxStalenessPrefix": "[parameters('maxStalenessPrefix')]",
        "maxIntervalInSeconds": "[parameters('maxIntervalInSeconds')]"
      },
      "Strong": {
        "defaultConsistencyLevel": "Strong"
      }
    },
    "locations": [
      {
        "locationName": "[parameters('primaryRegion')]",
        "failoverPriority": 0,
        "isZoneRedundant": false
      },
      {
        "locationName": "[parameters('secondaryRegion')]",
        "failoverPriority": 1,
        "isZoneRedundant": false
      }
    ]
  },
  "resources": [
    {
      "type": "Microsoft.DocumentDB/databaseAccounts",
      "apiVersion": "2020-06-01-preview",
      "name": "[variables('accountName')]",
      "location": "[parameters('location')]",
      "kind": "GlobalDocumentDB",
      "properties": {
        "consistencyPolicy": "[variables('consistencyPolicy')[parameters('defaultConsistencyLevel')]]",
        "locations": "[variables('locations')]",
        "databaseAccountOfferType": "Standard",
        "enableAutomaticFailover": "[parameters('automaticFailover')]"
      }
    },
    {
      "type": "Microsoft.DocumentDB/databaseAccounts/sqlDatabases",
      "apiVersion": "2020-06-01-preview",
      "name": "[concat(variables('accountName'), '/', parameters('databaseName'))]",
      "dependsOn": [ "[resourceId('Microsoft.DocumentDB/databaseAccounts', variables('accountName'))]" ],
      "properties": {
        "resource": {
          "id": "[parameters('databaseName')]"
        }
      }
    },
    {
      "type": "Microsoft.DocumentDB/databaseAccounts/sqlDatabases/containers",
      "apiVersion": "2020-06-01-preview",
      "name": "[concat(variables('accountName'), '/', parameters('databaseName'), '/',
parameters('containerName'))]",
      "dependsOn": [ "[resourceId('Microsoft.DocumentDB/databaseAccounts/sqlDatabases',
variables('accountName'), parameters('databaseName'))]" ],
      "properties": {
        "resource": {
```

```
          "resource": {
            "id": "[parameters('containerName')]",
            "partitionKey": {
              "paths": [
                "/myPartitionKey"
              ],
              "kind": "Hash"
            },
            "indexingPolicy": {
              "indexingMode": "consistent",
              "includedPaths": [
                {
                  "path": "/*"
                }
              ],
              "excludedPaths": [
                {
                  "path": "/myPathToNotIndex/*"
                }
              ],
              "compositeIndexes": [
                [
                  {
                    "path": "/name",
                    "order": "ascending"
                  },
                  {
                    "path": "/age",
                    "order": "descending"
                  }
                ]
              ],
              "spatialIndexes": [
                {
                  "path": "/path/to/geojson/property/?",
                  "types": [
                    "Point",
                    "Polygon",
                    "MultiPolygon",
                    "LineString"
                  ]
                }
              ]
            },
            "defaultTtl": 86400,
            "uniqueKeyPolicy": {
              "uniqueKeys": [
                {
                  "paths": [
                    "/phoneNumber"
                  ]
                }
              ]
            }
          },
          "options": { "throughput": "[parameters('throughput')]" }
        }
      }
    ]
  }
```

Three Azure resources are defined in the template:

- Microsoft.DocumentDB/databaseAccounts: Create an Azure Cosmos account.
- Microsoft.DocumentDB/databaseAccounts/sqlDatabases: Create an Azure Cosmos database.
- Microsoft.DocumentDB/databaseAccounts/sqlDatabases/containers: Create an Azure Cosmos container.

More Azure Cosmos DB template samples can be found in the quickstart template gallery.

Deploy the template

1. Select the following image to sign in to Azure and open a template. The template creates an Azure Cosmos account, a database, and a container.

   Deploy to Azure

2. Select or enter the following values.

Unless it is specified, use the default values to create the Azure Cosmos resources.

- **Subscription**: select an Azure subscription.
- **Resource group**: select **Create new**, enter a unique name for the resource group, and then click **OK**.
- **Location**: select a location. For example, **Central US**.
- **Account Name**: enter a name for the Azure Cosmos account. It must be globally unique.

- **Location**: enter a location where you want to create your Azure Cosmos account. The Azure Cosmos account can be in the same location as the resource group.
- **Primary Region**: The primary replica region for the Azure Cosmos account.
- **Secondary region**: The secondary replica region for the Azure Cosmos account.
- **Default Consistency Level**: The default consistency level for the Azure Cosmos account.
- **Max Staleness Prefix**: Max stale requests. Required for BoundedStaleness.
- **Max Interval in Seconds**: Max lag time. Required for BoundedStaleness.
- **Database Name**: The name of the Azure Cosmos database.
- **Container Name**: The name of the Azure Cosmos container.
- **Throughput**: The throughput for the container, minimum throughput value is 400 RU/s.
- **I agree to the terms and conditions state above**: Select.

3. Select **Purchase**. After the Azure Cosmos account has been deployed successfully, you get a notification:

The Azure portal is used to deploy the template. In addition to the Azure portal, you can also use the Azure PowerShell, Azure CLI, and REST API. To learn other deployment methods, see Deploy templates.

Validate the deployment

You can either use the Azure portal to check the Azure Cosmos account, the database, and the container or use the following Azure CLI or Azure PowerShell script to list the secret created.

- CLI
- PowerShell

```
echo "Enter your Azure Cosmos account name:" &&
read cosmosAccountName &&
echo "Enter the resource group where the Azure Cosmos account exists:" &&
read resourcegroupName &&
az cosmosdb show -g $resourcegroupName -n $cosmosAccountName
```

Clean up resources

If you plan to continue working with subsequent quickstarts and tutorials, you might want to leave these resources in place. When no longer needed, delete the resource group, which deletes the Azure Cosmos account and the related resources. To delete the resource group by using Azure CLI or Azure PowerShell:

- CLI
- PowerShell

```
echo "Enter the Resource Group name:" &&
read resourceGroupName &&
az group delete --name $resourceGroupName &&
echo "Press [ENTER] to continue ..."
```

Next steps

In this quickstart, you created an Azure Cosmos account, a database and a container by using an ARM template and validated the deployment. To learn more about Azure Cosmos DB and Azure Resource Manager, continue on to the articles below.

- Read an Overview of Azure Cosmos DB
- Learn more about Azure Resource Manager
- Get other Azure Cosmos DB Resource Manager templates

Quickstart: Build a console app using the .NET V4 SDK to manage Azure Cosmos DB SQL API account resources.

11/2/2020 • 13 minutes to read • Edit Online

APPLIES TO: SQL API

Get started with the Azure Cosmos DB SQL API client library for .NET. Follow the steps in this doc to install the .NET V4 (Azure.Cosmos) package, build an app, and try out the example code for basic CRUD operations on the data stored in Azure Cosmos DB.

Azure Cosmos DB is Microsoft's fast NoSQL database with open APIs for any scale. You can use Azure Cosmos DB to quickly create and query key/value, document, and graph databases. Use the Azure Cosmos DB SQL API client library for .NET to:

- Create an Azure Cosmos database and a container
- Add sample data to the container
- Query the data
- Delete the database

Library source code | Package (NuGet)

Prerequisites

- Azure subscription - create one for free or you can Try Azure Cosmos DB for free without an Azure subscription, free of charge and commitments.
- NET Core 3 SDK. You can verify which version is available in your environment by running `dotnet --version`.

Setting up

This section walks you through creating an Azure Cosmos account and setting up a project that uses Azure Cosmos DB SQL API client library for .NET to manage resources. The example code described in this article creates a `FamilyDatabase` database and family members (each family member is an item) within that database. Each family member has properties such as `Id, FamilyName, FirstName, LastName, Parents, Children, Address,`. The `LastName` property is used as the partition key for the container.

Create an Azure Cosmos account

If you use the Try Azure Cosmos DB for free option to create an Azure Cosmos account, you must create an Azure Cosmos DB account of type **SQL API**. An Azure Cosmos DB test account is already created for you. You don't have to create the account explicitly, so you can skip this section and move to the next section.

If you have your own Azure subscription or created a subscription for free, you should create an Azure Cosmos account explicitly. The following code will create an Azure Cosmos account with session consistency. The account is replicated in `South Central US` and `North Central US`.

You can use Azure Cloud Shell to create the Azure Cosmos account. Azure Cloud Shell is an interactive, authenticated, browser-accessible shell for managing Azure resources. It provides the flexibility of choosing the shell experience that best suits the way you work, either Bash or PowerShell. For this quickstart, choose **Bash** mode. Azure Cloud Shell also requires a storage account, you can create one when prompted.

Select the **Try It** button next to the following code, choose **Bash** mode select **create a storage account** and login to Cloud Shell. Next copy and paste the following code to Azure Cloud Shell and run it. The Azure Cosmos account name must be globally unique, make sure to update the `mysqlapicosmosdb` value before you run the command.

```
# Set variables for the new SQL API account, database, and container
resourceGroupName='myResourceGroup'
location='southcentralus'

# The Azure Cosmos account name must be globally unique, make sure to update the `mysqlapicosmosdb` value
before you run the command
accountName='mysqlapicosmosdb'

# Create a resource group
az group create \
    --name $resourceGroupName \
    --location $location

# Create a SQL API Cosmos DB account with session consistency and multi-region writes enabled
az cosmosdb create \
    --resource-group $resourceGroupName \
    --name $accountName \
    --kind GlobalDocumentDB \
    --locations regionName="South Central US" failoverPriority=0 --locations regionName="North Central US"
failoverPriority=1 \
    --default-consistency-level "Session" \
    --enable-multiple-write-locations true
```

The creation of the Azure Cosmos account takes a while, once the operation is successful, you can see the confirmation output. After the command completes successfully, sign into the Azure portal and verify that the Azure Cosmos account with the specified name exists. You can close the Azure Cloud Shell window after the resource is created.

Create a new .NET app

Create a new .NET application in your preferred editor or IDE. Open the Windows command prompt or a Terminal window from your local computer. You will run all the commands in the next sections from the command prompt or terminal. Run the following dotnet new command to create a new app with the name `todo`. The --langVersion parameter sets the LangVersion property in the created project file.

```
dotnet new console –langVersion:8 -n todo
```

Change your directory to the newly created app folder. You can build the application with:

```
cd todo
dotnet build
```

The expected output from the build should look something like this:

```
  Restore completed in 100.37 ms for C:\Users\user1\Downloads\CosmosDB_Samples\todo\todo.csproj.
  todo -> C:\Users\user1\Downloads\CosmosDB_Samples\todo\bin\Debug\netcoreapp3.0\todo.dll

Build succeeded.
    0 Warning(s)
    0 Error(s)

Time Elapsed 00:00:34.17
```

Install the Azure Cosmos DB package

While still in the application directory, install the Azure Cosmos DB client library for .NET Core by using the dotnet add package command.

```
dotnet add package Azure.Cosmos --version 4.0.0-preview3
```

Copy your Azure Cosmos account credentials from the Azure portal

The sample application needs to authenticate to your Azure Cosmos account. To authenticate, you should pass the Azure Cosmos account credentials to the application. Get your Azure Cosmos account credentials by following these steps:

1. Sign in to the Azure portal.
2. Navigate to your Azure Cosmos account.
3. Open the **Keys** pane and copy the **URI** and **PRIMARY KEY** of your account. You will add the URI and keys values to an environment variable in the next step.

Object model

Before you start building the application, let's look into the hierarchy of resources in Azure Cosmos DB and the object model used to create and access these resources. The Azure Cosmos DB creates resources in the following order:

- Azure Cosmos account
- Databases
- Containers
- Items

To learn in more about the hierarchy of different entities, see the working with databases, containers, and items in Azure Cosmos DB article. You will use the following .NET classes to interact with these resources:

- CosmosClient - This class provides a client-side logical representation for the Azure Cosmos DB service. The client object is used to configure and execute requests against the service.
- CreateDatabaseIfNotExistsAsync - This method creates (if doesn't exist) or gets (if already exists) a database resource as an asynchronous operation.
- CreateContainerIfNotExistsAsync - This method creates (if it doesn't exist) or gets (if it already exists) a container as an asynchronous operation. You can check the status code from the response to determine whether the container was newly created (201) or an existing container was returned (200).
- CreateItemAsync - This method creates an item within the container.
- UpsertItemAsync - This method creates an item within the container if it doesn't already exist or replaces the item if it already exists.
- GetItemQueryIterator - This method creates a query for items under a container in an Azure Cosmos database using a SQL statement with parameterized values.

- DeleteAsync - Deletes the specified database from your Azure Cosmos account. `DeleteAsync` method only deletes the database.

Code examples

The sample code described in this article creates a family database in Azure Cosmos DB. The family database contains family details such as name, address, location, the associated parents, children, and pets. Before populating the data to your Azure Cosmos account, define the properties of a family item. Create a new class named `Family.cs` at the root level of your sample application and add the following code to it:

```
using System.Text.Json;
using System.Text.Json.Serialization;

namespace todo
{
    public class Family
    {
        [JsonPropertyName("id")]
        public string Id { get; set; }
        public string LastName { get; set; }
        public Parent[] Parents { get; set; }
        public Child[] Children { get; set; }
        public Address Address { get; set; }
        public bool IsRegistered { get; set; }
        public override string ToString()
        {
            return JsonSerializer.Serialize(this);
        }
    }

    public class Parent
    {
        public string FamilyName { get; set; }
        public string FirstName { get; set; }
    }

    public class Child
    {
        public string FamilyName { get; set; }
        public string FirstName { get; set; }
        public string Gender { get; set; }
        public int Grade { get; set; }
        public Pet[] Pets { get; set; }
    }

    public class Pet
    {
        public string GivenName { get; set; }
    }

    public class Address
    {
        public string State { get; set; }
        public string County { get; set; }
        public string City { get; set; }
    }
}
```

Add the using directives & define the client object

From the project directory, open the `Program.cs` file in your editor and add the following using directives at the top of your application:

```
using System;
using System.Collections.Generic;
using System.Net;
using System.Threading.Tasks;
using Azure.Cosmos;
```

Add the following global variables in your `Program` class. These will include the endpoint and authorization keys, the name of the database, and container that you will create. Make sure to replace the endpoint and authorization keys values according to your environment.

```
private const string EndpointUrl = "https://<your-account>.documents.azure.com:443/";
private const string AuthorizationKey = "<your-account-key>";
private const string DatabaseId = "FamilyDatabase";
private const string ContainerId = "FamilyContainer";
```

Finally, replace the `Main` method:

```
static async Task Main(string[] args)
{

    CosmosClient cosmosClient = new CosmosClient(EndpointUrl, AuthorizationKey);
    await Program.CreateDatabaseAsync(cosmosClient);
    await Program.CreateContainerAsync(cosmosClient);
    await Program.AddItemsToContainerAsync(cosmosClient);
    await Program.QueryItemsAsync(cosmosClient);
    await Program.ReplaceFamilyItemAsync(cosmosClient);
    await Program.DeleteFamilyItemAsync(cosmosClient);
    await Program.DeleteDatabaseAndCleanupAsync(cosmosClient);
}
```

Create a database

Define the `CreateDatabaseAsync` method within the `program.cs` class. This method creates the `FamilyDatabase` if it doesn't already exist.

```
/// <summary>
/// Create the database if it does not exist
/// </summary>
private static async Task CreateDatabaseAsync(CosmosClient cosmosClient)
{
    // Create a new database
    CosmosDatabase database = await cosmosClient.CreateDatabaseIfNotExistsAsync(Program.DatabaseId);
    Console.WriteLine("Created Database: {0}\n", database.Id);
}
```

Create a container

Define the `CreateContainerAsync` method within the `Program` class. This method creates the `FamilyContainer` if it doesn't already exist.

```
/// <summary>
/// Create the container if it does not exist.
/// Specify "/LastName" as the partition key since we're storing family information, to ensure good
distribution of requests and storage.
/// </summary>
/// <returns></returns>
private static async Task CreateContainerAsync(CosmosClient cosmosClient)
{
    // Create a new container
    CosmosContainer container = await
cosmosClient.GetDatabase(Program.DatabaseId).CreateContainerIfNotExistsAsync(Program.ContainerId,
"/LastName");
    Console.WriteLine("Created Container: {0}\n", container.Id);
}
```

Create an item

Create a family item by adding the `AddItemsToContainerAsync` method with the following code. You can use the `CreateItemAsync` or `UpsertItemAsync` methods to create an item:

```
/// <summary>
/// Add Family items to the container
/// </summary>
private static async Task AddItemsToContainerAsync(CosmosClient cosmosClient)
{
    // Create a family object for the Andersen family
    Family andersenFamily = new Family
    {
        Id = "Andersen.1",
        LastName = "Andersen",
        Parents = new Parent[]
        {
            new Parent { FirstName = "Thomas" },
            new Parent { FirstName = "Mary Kay" }
        },
        Children = new Child[]
        {
            new Child
            {
                FirstName = "Henriette Thaulow",
                Gender = "female",
                Grade = 5,
                Pets = new Pet[]
                {
                    new Pet { GivenName = "Fluffy" }
                }
            }
        },
        Address = new Address { State = "WA", County = "King", City = "Seattle" },
        IsRegistered = false
    };

    CosmosContainer container = cosmosClient.GetContainer(Program.DatabaseId, Program.ContainerId);
    try
    {
        // Read the item to see if it exists.
        ItemResponse<Family> andersenFamilyResponse = await container.ReadItemAsync<Family>(andersenFamily.Id,
new PartitionKey(andersenFamily.LastName));
        Console.WriteLine("Item in database with id: {0} already exists\n", andersenFamilyResponse.Value.Id);
    }
    catch(CosmosException ex) when (ex.Status == (int)HttpStatusCode.NotFound)
    {
        // Create an item in the container representing the Andersen family. Note we provide the value of the
partition key for this item, which is "Andersen"
        ItemResponse<Family> andersenFamilyResponse = await container.CreateItemAsync<Family>(andersenFamily,
new PartitionKey(andersenFamily.LastName));
```

```
        // Note that after creating the item, we can access the body of the item with the Resource property
off the ItemResponse.
        Console.WriteLine("Created item in database with id: {0}\n", andersenFamilyResponse.Value.Id);
    }

    // Create a family object for the Wakefield family
    Family wakefieldFamily = new Family
    {
        Id = "Wakefield.7",
        LastName = "Wakefield",
        Parents = new Parent[]
        {
            new Parent { FamilyName = "Wakefield", FirstName = "Robin" },
            new Parent { FamilyName = "Miller", FirstName = "Ben" }
        },
        Children = new Child[]
        {
            new Child
            {
                FamilyName = "Merriam",
                FirstName = "Jesse",
                Gender = "female",
                Grade = 8,
                Pets = new Pet[]
                {
                    new Pet { GivenName = "Goofy" },
                    new Pet { GivenName = "Shadow" }
                }
            },
            new Child
            {
                FamilyName = "Miller",
                FirstName = "Lisa",
                Gender = "female",
                Grade = 1
            }
        },
        Address = new Address { State = "NY", County = "Manhattan", City = "NY" },
        IsRegistered = true
    };

    // Create an item in the container representing the Wakefield family. Note we provide the value of the
partition key for this item, which is "Wakefield"
    ItemResponse<Family> wakefieldFamilyResponse = await container.UpsertItemAsync<Family>(wakefieldFamily,
new PartitionKey(wakefieldFamily.LastName));

    // Note that after creating the item, we can access the body of the item with the Resource property off
the ItemResponse. We can also access the RequestCharge property to see the amount of RUs consumed on this
request.
    Console.WriteLine("Created item in database with id: {0}\n", wakefieldFamilyResponse.Value.Id);
}
```

Query the items

After inserting an item, you can run a query to get the details of "Andersen" family. The following code shows how to execute the query using the SQL query directly. The SQL query to get the "Anderson" family details is: `SELECT * FROM c WHERE c.LastName = 'Andersen'`. Define the `QueryItemsAsync` method within the `Program` class and add the following code to it:

```
/// <summary>
/// Run a query (using Azure Cosmos DB SQL syntax) against the container
/// </summary>
private static async Task QueryItemsAsync(CosmosClient cosmosClient)
{
    var sqlQueryText = "SELECT * FROM c WHERE c.LastName = 'Andersen'";

    Console.WriteLine("Running query: {0}\n", sqlQueryText);

    CosmosContainer container = cosmosClient.GetContainer(Program.DatabaseId, Program.ContainerId);

    QueryDefinition queryDefinition = new QueryDefinition(sqlQueryText);

    List<Family> families = new List<Family>();

    await foreach (Family family in container.GetItemQueryIterator<Family>(queryDefinition))
    {
        families.Add(family);
        Console.WriteLine("\tRead {0}\n", family);
    }
}
```

Replace an item

Read a family item and then update it by adding the `ReplaceFamilyItemAsync` method with the following code.

```
/// <summary>
/// Replace an item in the container
/// </summary>
private static async Task ReplaceFamilyItemAsync(CosmosClient cosmosClient)
{
    CosmosContainer container = cosmosClient.GetContainer(Program.DatabaseId, Program.ContainerId);

    ItemResponse<Family> wakefieldFamilyResponse = await container.ReadItemAsync<Family>("Wakefield.7", new
PartitionKey("Wakefield"));
    Family itemBody = wakefieldFamilyResponse;

    // update registration status from false to true
    itemBody.IsRegistered = true;
    // update grade of child
    itemBody.Children[0].Grade = 6;

    // replace the item with the updated content
    wakefieldFamilyResponse = await container.ReplaceItemAsync<Family>(itemBody, itemBody.Id, new
PartitionKey(itemBody.LastName));
    Console.WriteLine("Updated Family [{0},{1}].\n \tBody is now: {2}\n", itemBody.LastName, itemBody.Id,
wakefieldFamilyResponse.Value);
}
```

Delete an item

Delete a family item by adding the `DeleteFamilyItemAsync` method with the following code.

```
/// <summary>
/// Delete an item in the container
/// </summary>
private static async Task DeleteFamilyItemAsync(CosmosClient cosmosClient)
{
    CosmosContainer container = cosmosClient.GetContainer(Program.DatabaseId, Program.ContainerId);

    string partitionKeyValue = "Wakefield";
    string familyId = "Wakefield.7";

    // Delete an item. Note we must provide the partition key value and id of the item to delete
    ItemResponse<Family> wakefieldFamilyResponse = await container.DeleteItemAsync<Family>(familyId,new
PartitionKey(partitionKeyValue));
    Console.WriteLine("Deleted Family [{0},{1}]\n", partitionKeyValue, familyId);
}
```

Delete the database

Finally you can delete the database adding the `DeleteDatabaseAndCleanupAsync` method with the following code:

```
/// <summary>
/// Delete the database and dispose of the Cosmos Client instance
/// </summary>
private static async Task DeleteDatabaseAndCleanupAsync(CosmosClient cosmosClient)
{
    CosmosDatabase database = cosmosClient.GetDatabase(Program.DatabaseId);
    DatabaseResponse databaseResourceResponse = await database.DeleteAsync();

    Console.WriteLine("Deleted Database: {0}\n", Program.DatabaseId);
}
```

After you add all the required methods, save the `Program` file.

Run the code

Next build and run the application to create the Azure Cosmos DB resources.

```
dotnet run
```

The following output is generated when you run the application. You can also sign into the Azure portal and validate that the resources are created:

```
Created Database: FamilyDatabase

Created Container: FamilyContainer

Created item in database with id: Andersen.1

Running query: SELECT * FROM c WHERE c.LastName = 'Andersen'

        Read {"id":"Andersen.1","LastName":"Andersen","Parents":[{"FamilyName":null,"FirstName":"Thomas"},
{"FamilyName":null   "FirstName":"Mary Kay"}],"Children":[{"FamilyName":null,"FirstName":"Henriette
Thaulow","Gender":"female","Grade":5,"Pets": [{"GivenName":"Fluffy"}]}],"Address":
{"State":"WA","County":"King","City":"Seattle"},"IsRegistered":false}

Updated Family [Wakefield,Wakefield.7].
        Body is now: {"id":"Wakefield.7","LastName":"Wakefield","Parents":
[{"FamilyName":"Wakefield","FirstName":"Robin"}   {"FamilyName":"Miller","FirstName":"Ben"}],"Children":
[{"FamilyName":"Merriam","FirstName":"Jesse","Gender":"female","Grade":6   "Pets":[{"GivenName":"Goofy"},
{"GivenName":"Shadow"}]},{"FamilyName":"Miller","FirstName":"Lisa","Gender":"female","Grade":1
"Pets":null}],"Address":{"State":"NY","County":"Manhattan","City":"NY"},"IsRegistered":true}

Deleted Family [Wakefield,Wakefield.7]

Deleted Database: FamilyDatabase

End of demo, press any key to exit.
```

You can validate that the data is created by signing into the Azure portal and see the required items in your Azure Cosmos account.

Clean up resources

When no longer needed, you can use the Azure CLI or Azure PowerShell to remove the Azure Cosmos account and the corresponding resource group. The following command shows how to delete the resource group by using the Azure CLI:

```
az group delete -g "myResourceGroup"
```

Next steps

In this quickstart, you learned how to create an Azure Cosmos account, create a database and a container using a .NET Core app. You can now import additional data to your Azure Cosmos account with the instructions int the following article.

Import data into Azure Cosmos DB

Quickstart: Build a .NET console app to manage Azure Cosmos DB SQL API resources

11/2/2020 • 12 minutes to read • Edit Online

APPLIES TO: SQL API

Get started with the Azure Cosmos DB SQL API client library for .NET. Follow the steps in this doc to install the .NET package, build an app, and try out the example code for basic CRUD operations on the data stored in Azure Cosmos DB.

Azure Cosmos DB is Microsoft's fast NoSQL database with open APIs for any scale. You can use Azure Cosmos DB to quickly create and query key/value, document, and graph databases. Use the Azure Cosmos DB SQL API client library for .NET to:

- Create an Azure Cosmos database and a container
- Add sample data to the container
- Query the data
- Delete the database

API reference documentation | Library source code | Package (NuGet)

Prerequisites

- Azure subscription - create one for free or you can Try Azure Cosmos DB for free without an Azure subscription, free of charge and commitments.
- The .NET Core 2.1 SDK or later.

Setting up

This section walks you through creating an Azure Cosmos account and setting up a project that uses Azure Cosmos DB SQL API client library for .NET to manage resources. The example code described in this article creates a `FamilyDatabase` database and family members (each family member is an item) within that database. Each family member has properties such as `Id, FamilyName, FirstName, LastName, Parents, Children, Address,`. The `LastName` property is used as the partition key for the container.

Create an Azure Cosmos account

If you use the Try Azure Cosmos DB for free option to create an Azure Cosmos account, you must create an Azure Cosmos DB account of type **SQL API**. An Azure Cosmos DB test account is already created for you. You don't have to create the account explicitly, so you can skip this section and move to the next section.

If you have your own Azure subscription or created a subscription for free, you should create an Azure Cosmos account explicitly. The following code will create an Azure Cosmos account with session consistency. The account is replicated in `South Central US` and `North Central US`.

You can use Azure Cloud Shell to create the Azure Cosmos account. Azure Cloud Shell is an interactive, authenticated, browser-accessible shell for managing Azure resources. It provides the flexibility of choosing the shell experience that best suits the way you work, either Bash or PowerShell. For this quickstart, choose **Bash** mode. Azure Cloud Shell also requires a storage account, you can create one when prompted.

Select the **Try It** button next to the following code, choose **Bash** mode select **create a storage account** and login to Cloud Shell. Next copy and paste the following code to Azure Cloud Shell and run it. The Azure Cosmos

account name must be globally unique, make sure to update the `mysqlapicosmosdb` value before you run the command.

```
# Set variables for the new SQL API account, database, and container
resourceGroupName='myResourceGroup'
location='southcentralus'

# The Azure Cosmos account name must be globally unique, make sure to update the `mysqlapicosmosdb` value
before you run the command
accountName='mysqlapicosmosdb'

# Create a resource group
az group create \
    --name $resourceGroupName \
    --location $location

# Create a SQL API Cosmos DB account with session consistency and multi-region writes enabled
az cosmosdb create \
    --resource-group $resourceGroupName \
    --name $accountName \
    --kind GlobalDocumentDB \
    --locations regionName="South Central US" failoverPriority=0 --locations regionName="North Central US"
failoverPriority=1 \
    --default-consistency-level "Session" \
    --enable-multiple-write-locations true
```

The creation of the Azure Cosmos account takes a while, once the operation is successful, you can see the confirmation output. After the command completes successfully, sign into the Azure portal and verify that the Azure Cosmos account with the specified name exists. You can close the Azure Cloud Shell window after the resource is created.

Create a new .NET app

Create a new .NET application in your preferred editor or IDE. Open the Windows command prompt or a Terminal window from your local computer. You will run all the commands in the next sections from the command prompt or terminal. Run the following dotnet new command to create a new app with the name `todo`. The --langVersion parameter sets the LangVersion property in the created project file.

```
dotnet new console --langVersion 7.1 -n todo
```

Change your directory to the newly created app folder. You can build the application with:

```
cd todo
dotnet build
```

The expected output from the build should look something like this:

```
  Restore completed in 100.37 ms for C:\Users\user1\Downloads\CosmosDB_Samples\todo\todo.csproj.
  todo -> C:\Users\user1\Downloads\CosmosDB_Samples\todo\bin\Debug\netcoreapp2.2\todo.dll
  todo -> C:\Users\user1\Downloads\CosmosDB_Samples\todo\bin\Debug\netcoreapp2.2\todo.Views.dll

Build succeeded.
    0 Warning(s)
    0 Error(s)

Time Elapsed 00:00:34.17
```

Install the Azure Cosmos DB package

While still in the application directory, install the Azure Cosmos DB client library for .NET Core by using the dotnet add package command.

```
dotnet add package Microsoft.Azure.Cosmos
```

Copy your Azure Cosmos account credentials from the Azure portal

The sample application needs to authenticate to your Azure Cosmos account. To authenticate, you should pass the Azure Cosmos account credentials to the application. Get your Azure Cosmos account credentials by following these steps:

1. Sign in to the Azure portal.
2. Navigate to your Azure Cosmos account.
3. Open the **Keys** pane and copy the **URI** and **PRIMARY KEY** of your account. You will add the URI and keys values to an environment variable in the next step.

Set the environment variables

After you have copied the **URI** and **PRIMARY KEY** of your account, save them to a new environment variable on the local machine running the application. To set the environment variable, open a console window, and run the following command. Make sure to replace `<Your_Azure_Cosmos_account_URI>` and `<Your_Azure_Cosmos_account_PRIMARY_KEY>` values.

Windows

```
setx EndpointUrl "<Your_Azure_Cosmos_account_URI>"
setx PrimaryKey "<Your_Azure_Cosmos_account_PRIMARY_KEY>"
```

Linux

```
export EndpointUrl = "<Your_Azure_Cosmos_account_URI>"
export PrimaryKey = "<Your_Azure_Cosmos_account_PRIMARY_KEY>"
```

macOS

```
export EndpointUrl = "<Your_Azure_Cosmos_account_URI>"
export PrimaryKey = "<Your_Azure_Cosmos_account_PRIMARY_KEY>"
```

Object model

Before you start building the application, let's look into the hierarchy of resources in Azure Cosmos DB and the object model used to create and access these resources. The Azure Cosmos DB creates resources in the following order:

- Azure Cosmos account
- Databases
- Containers
- Items

To learn in more about the hierarchy of different entities, see the working with databases, containers, and items in Azure Cosmos DB article. You will use the following .NET classes to interact with these resources:

- CosmosClient - This class provides a client-side logical representation for the Azure Cosmos DB service. The client object is used to configure and execute requests against the service.
- CreateDatabaseIfNotExistsAsync - This method creates (if doesn't exist) or gets (if already exists) a database resource as an asynchronous operation.
- CreateContainerIfNotExistsAsync- - This method creates (if it doesn't exist) or gets (if it already exists) a container as an asynchronous operation. You can check the status code from the response to determine whether the container was newly created (201) or an existing container was returned (200).
- CreateItemAsync - This method creates an item within the container.
- UpsertItemAsync - This method creates an item within the container if it doesn't already exist or replaces the item if it already exists.
- GetItemQueryIterator - This method creates a query for items under a container in an Azure Cosmos database using a SQL statement with parameterized values.
- DeleteAsync - Deletes the specified database from your Azure Cosmos account. `DeleteAsync` method only deletes the database. Disposing of the `Cosmosclient` instance should happen separately (which it does in the DeleteDatabaseAndCleanupAsync method.

Code examples

The sample code described in this article creates a family database in Azure Cosmos DB. The family database contains family details such as name, address, location, the associated parents, children, and pets. Before populating the data to your Azure Cosmos account, define the properties of a family item. Create a new class named `Family.cs` at the root level of your sample application and add the following code to it:

```
using Newtonsoft.Json;

namespace todo
{
    public class Family
    {
        [JsonProperty(PropertyName = "id")]
        public string Id { get; set; }
        public string LastName { get; set; }
        public Parent[] Parents { get; set; }
        public Child[] Children { get; set; }
        public Address Address { get; set; }
        public bool IsRegistered { get; set; }
        // The ToString() method is used to format the output, it's used for demo purpose only. It's not
required by Azure Cosmos DB
        public override string ToString()
        {
            return JsonConvert.SerializeObject(this);
        }
    }

    public class Parent
    {
        public string FamilyName { get; set; }
        public string FirstName { get; set; }
    }

    public class Child
    {
        public string FamilyName { get; set; }
        public string FirstName { get; set; }
        public string Gender { get; set; }
        public int Grade { get; set; }
        public Pet[] Pets { get; set; }
    }

    public class Pet
    {
        public string GivenName { get; set; }
    }

    public class Address
    {
        public string State { get; set; }
        public string County { get; set; }
        public string City { get; set; }
    }
}
```

Add the using directives & define the client object

From the project directory, open the `Program.cs` file in your editor and add the following using directives at the top of your application:

```
using System;
using System.Threading.Tasks;
using System.Configuration;
using System.Collections.Generic;
using System.Net;
using Microsoft.Azure.Cosmos;
```

To the **Program.cs** file, add code to read the environment variables that you have set in the previous step. Define the `CosmosClient`, `Database`, and the `Container` objects. Next add code to the main method that calls the

`GetStartedDemoAsync` method where you manage Azure Cosmos account resources.

```
namespace todo
{
public class Program
{

    /// The Azure Cosmos DB endpoint for running this GetStarted sample.
    private string EndpointUrl = Environment.GetEnvironmentVariable("EndpointUrl");

    /// The primary key for the Azure DocumentDB account.
    private string PrimaryKey = Environment.GetEnvironmentVariable("PrimaryKey");

    // The Cosmos client instance
    private CosmosClient cosmosClient;

    // The database we will create
    private Database database;

    // The container we will create.
    private Container container;

    // The name of the database and container we will create
    private string databaseId = "FamilyDatabase";
    private string containerId = "FamilyContainer";

    public static async Task Main(string[] args)
    {
       try
       {
           Console.WriteLine("Beginning operations...\n");
           Program p = new Program();
           await p.GetStartedDemoAsync();

        }
        catch (CosmosException de)
        {
           Exception baseException = de.GetBaseException();
           Console.WriteLine("{0} error occurred: {1}", de.StatusCode, de);
        }
        catch (Exception e)
        {
            Console.WriteLine("Error: {0}", e);
        }
        finally
        {
            Console.WriteLine("End of demo, press any key to exit.");
            Console.ReadKey();
        }
    }
}
}
```

Create a database

Define the `CreateDatabaseAsync` method within the `program.cs` class. This method creates the `FamilyDatabase` if it doesn't already exist.

```
private async Task CreateDatabaseAsync()
{
    // Create a new database
    this.database = await this.cosmosClient.CreateDatabaseIfNotExistsAsync(databaseId);
    Console.WriteLine("Created Database: {0}\n", this.database.Id);
}
```

Create a container

Define the `CreateContainerAsync` method within the `program.cs` class. This method creates the `FamilyContainer` if it doesn't already exist.

```
/// Create the container if it does not exist.
/// Specifiy "/LastName" as the partition key since we're storing family information, to ensure good
distribution of requests and storage.
private async Task CreateContainerAsync()
{
    // Create a new container
    this.container = await this.database.CreateContainerIfNotExistsAsync(containerId, "/LastName");
    Console.WriteLine("Created Container: {0}\n", this.container.Id);
}
```

Create an item

Create a family item by adding the `AddItemsToContainerAsync` method with the following code. You can use the `CreateItemAsync` or `UpsertItemAsync` methods to create an item:

```
private async Task AddItemsToContainerAsync()
{
    // Create a family object for the Andersen family
    Family andersenFamily = new Family
    {
        Id = "Andersen.1",
        LastName = "Andersen",
        Parents = new Parent[]
        {
           new Parent { FirstName = "Thomas" },
           new Parent { FirstName = "Mary Kay" }
        },
        Children = new Child[]
        {
           new Child
            {
                FirstName = "Henriette Thaulow",
                Gender = "female",
                Grade = 5,
                Pets = new Pet[]
                {
                    new Pet { GivenName = "Fluffy" }
                }
            }
        },
        Address = new Address { State = "WA", County = "King", City = "Seattle" },
        IsRegistered = false
    };

    try
    {
        // Create an item in the container representing the Andersen family. Note we provide the value of
the partition key for this item, which is "Andersen".
        ItemResponse<Family> andersenFamilyResponse = await this.container.CreateItemAsync<Family>
(andersenFamily, new PartitionKey(andersenFamily.LastName));
        // Note that after creating the item, we can access the body of the item with the Resource property
of the ItemResponse. We can also access the RequestCharge property to see the amount of RUs consumed on this
request.
        Console.WriteLine("Created item in database with id: {0} Operation consumed {1} RUs.\n",
andersenFamilyResponse.Resource.Id, andersenFamilyResponse.RequestCharge);
    }
    catch (CosmosException ex) when (ex.StatusCode == HttpStatusCode.Conflict)
    {
        Console.WriteLine("Item in database with id: {0} already exists\n", andersenFamily.Id);
    }
}
```

Query the items

After inserting an item, you can run a query to get the details of "Andersen" family. The following code shows how to execute the query using the SQL query directly. The SQL query to get the "Anderson" family details is: `SELECT * FROM c WHERE c.LastName = 'Andersen'`. Define the `QueryItemsAsync` method within the `program.cs` class and add the following code to it:

```
private async Task QueryItemsAsync()
{
    var sqlQueryText = "SELECT * FROM c WHERE c.LastName = 'Andersen'";

    Console.WriteLine("Running query: {0}\n", sqlQueryText);

    QueryDefinition queryDefinition = new QueryDefinition(sqlQueryText);
    FeedIterator<Family> queryResultSetIterator = this.container.GetItemQueryIterator<Family>
(queryDefinition);

    List<Family> families = new List<Family>();

    while (queryResultSetIterator.HasMoreResults)
    {
        FeedResponse<Family> currentResultSet = await queryResultSetIterator.ReadNextAsync();
        foreach (Family family in currentResultSet)
        {
            families.Add(family);
            Console.WriteLine("\tRead {0}\n", family);
        }
    }
}
```

Delete the database

Finally you can delete the database adding the `DeleteDatabaseAndCleanupAsync` method with the following code:

```
private async Task DeleteDatabaseAndCleanupAsync()
{
   DatabaseResponse databaseResourceResponse = await this.database.DeleteAsync();
   // Also valid: await this.cosmosClient.Databases["FamilyDatabase"].DeleteAsync();

   Console.WriteLine("Deleted Database: {0}\n", this.databaseId);

   //Dispose of CosmosClient
    this.cosmosClient.Dispose();
}
```

Execute the CRUD operations

After you have defined all the required methods, execute them with in the `GetStartedDemoAsync` method. The `DeleteDatabaseAndCleanupAsync` method commented out in this code because you will not see any resources if that method is executed. You can uncomment it after validating that your Azure Cosmos DB resources were created in the Azure portal.

```
public async Task GetStartedDemoAsync()
{
    // Create a new instance of the Cosmos Client
    this.cosmosClient = new CosmosClient(EndpointUrl, PrimaryKey);
    await this.CreateDatabaseAsync();
    await this.CreateContainerAsync();
    await this.AddItemsToContainerAsync();
    await this.QueryItemsAsync();
}
```

After you add all the required methods, save the `Program.cs` file.

Run the code

Next build and run the application to create the Azure Cosmos DB resources. Make sure to open a new

command prompt window, don't use the same instance that you have used to set the environment variables. Because the environment variables are not set in the current open window. You will need to open a new command prompt to see the updates.

```
dotnet build
```

```
dotnet run
```

The following output is generated when you run the application. You can also sign into the Azure portal and validate that the resources are created:

```
Created Database: FamilyDatabase

Created Container: FamilyContainer

Created item in database with id: Andersen.1 Operation consumed 11.62 RUs.

Running query: SELECT * FROM c WHERE c.LastName = 'Andersen'

        Read {"id":"Andersen.1","LastName":"Andersen","Parents":[{"FamilyName":null,"FirstName":"Thomas"},
{"FamilyName":null,"FirstName":"Mary Kay"}],"Children":[{"FamilyName":null,"FirstName":"Henriette
Thaulow","Gender":"female","Grade":5,"Pets":[{"GivenName":"Fluffy"}]}],"Address":
{"State":"WA","County":"King","City":"Seattle"},"IsRegistered":false}

End of demo, press any key to exit.
```

You can validate that the data is created by signing into the Azure portal and see the required items in your Azure Cosmos account.

Clean up resources

When no longer needed, you can use the Azure CLI or Azure PowerShell to remove the Azure Cosmos account and the corresponding resource group. The following command shows how to delete the resource group by using the Azure CLI:

```
az group delete -g "myResourceGroup"
```

Next steps

In this quickstart, you learned how to create an Azure Cosmos account, create a database and a container using a .NET Core app. You can now import additional data to your Azure Cosmos account with the instructions int the following article.

Import data into Azure Cosmos DB

Quickstart: Build a Java app to manage Azure Cosmos DB SQL API data

11/2/2020 • 15 minutes to read • Edit Online

APPLIES TO: SQL API

In this quickstart, you create and manage an Azure Cosmos DB SQL API account from the Azure portal, and by using a Java app cloned from GitHub. First, you create an Azure Cosmos DB SQL API account using the Azure portal, then create a Java app using the SQL Java SDK, and then add resources to your Cosmos DB account by using the Java application. Azure Cosmos DB is a multi-model database service that lets you quickly create and query document, table, key-value, and graph databases with global distribution and horizontal scale capabilities.

> **IMPORTANT**
>
> This quickstart is for Azure Cosmos DB Java SDK v4 only. Please view the Azure Cosmos DB Java SDK v4 Release notes, Maven repository, Azure Cosmos DB Java SDK v4 performance tips, and Azure Cosmos DB Java SDK v4 troubleshooting guide for more information. If you are currently using an older version than v4, see the Migrate to Azure Cosmos DB Java SDK v4 guide for help upgrading to v4.

Prerequisites

- An Azure account with an active subscription. Create one for free. Or try Azure Cosmos DB for free without an Azure subscription. You can also use the Azure Cosmos DB Emulator with a URI of `https://localhost:8081` and the key `C2y6yDjf5/R+ob0N8A7Cgv30VRDJIWEHLM+4QDU5DE2nQ9nDuVTqobD4b8mGGyPMbIZnqyMsEcaGQy67XIw/Jw==`.
- Java Development Kit (JDK) 8. Point your `JAVA_HOME` environment variable to the folder where the JDK is installed.
- A Maven binary archive. On Ubuntu, run `apt-get install maven` to install Maven.
- Git. On Ubuntu, run `sudo apt-get install git` to install Git.

Introductory notes

The structure of a Cosmos DB account. Irrespective of API or programming language, a Cosmos DB *account* contains zero or more *databases*, a *database* (DB) contains zero or more *containers*, and a *container* contains zero or more items, as shown in the diagram below:

You may read more about databases, containers and items here. A few important properties are defined at the level of the container, among them *provisioned throughput* and *partition key*.

The provisioned throughput is measured in Request Units (*RUs*) which have a monetary price and are a substantial determining factor in the operating cost of the account. Provisioned throughput can be selected at per-container granularity or per-database granularity, however container-level throughput specification is typically preferred. You may read more about throughput provisioning here.

As items are inserted into a Cosmos DB container, the database grows horizontally by adding more storage and compute to handle requests. Storage and compute capacity are added in discrete units known as *partitions*, and you must choose one field in your documents to be the partition key which maps each document to a partition. The way partitions are managed is that each partition is assigned a roughly equal slice out of the range of partition key values; therefore you are advised to choose a partition key which is relatively random or evenly-distributed. Otherwise, some partitions will see substantially more requests (*hot partition*) while other partitions see substantially fewer requests (*cold partition*), and this is to be avoided. You may learn more about partitioning here.

Create a database account

Before you can create a document database, you need to create a SQL API account with Azure Cosmos DB.

1. From the Azure portal menu or the **Home page**, select **Create a resource**.
2. On the **New** page, search for and select **Azure Cosmos DB**.
3. On the **Azure Cosmos DB** page, select **Create**.
4. On the **Create Azure Cosmos DB Account** page, enter the basic settings for the new Azure Cosmos account.

SETTING	VALUE	DESCRIPTION
Subscription	Subscription name	Select the Azure subscription that you want to use for this Azure Cosmos account.
Resource Group	Resource group name	Select a resource group, or select **Create new**, then enter a unique name for the new resource group.

SETTING	VALUE	DESCRIPTION
Account Name	A unique name	Enter a name to identify your Azure Cosmos account. Because *documents.azure.com* is appended to the name that you provide to create your URI, use a unique name. The name can only contain lowercase letters, numbers, and the hyphen (-) character. It must be between 3-44 characters in length.
API	The type of account to create	Select **Core (SQL)** to create a document database and query by using SQL syntax. The API determines the type of account to create. Azure Cosmos DB provides five APIs: Core (SQL) and MongoDB for document data, Gremlin for graph data, Azure Table, and Cassandra. Currently, you must create a separate account for each API.
Capacity mode	Provisioned throughput or Serverless	Select **Provisioned throughput** to create an account in provisioned throughput mode. Select **Serverless** to create an account in serverless mode.
Apply Free Tier Discount	Apply or Do not apply	With Azure Cosmos DB free tier, you will get the first 400 RU/s and 5 GB of storage for free in an account. Learn more about free tier.
Location	The region closest to your users	Select a geographic location to host your Azure Cosmos DB account. Use the location that is closest to your users to give them the fastest access to the data.
Account Type	Production or Non-Production	Select **Production** if the account will be used for a production workload. Select **Non-Production** if the account will be used for non-production, e.g. development, testing, QA, or staging. This is an Azure resource tag setting that tunes the Portal experience but does not affect the underlying Azure Cosmos DB account. You can change this value anytime.

> **NOTE**
>
> You can have up to one free tier Azure Cosmos DB account per Azure subscription and must opt-in when creating the account. If you do not see the option to apply the free tier discount, this means another account in the subscription has already been enabled with free tier.

> **NOTE**
>
> The following options are not available if you select **Serverless** as the **Capacity mode**:
>
> - Apply Free Tier Discount
> - Geo-redundancy
> - Multi-region Writes

5. Select **Review + create**. You can skip the **Network** and **Tags** sections.

6. Review the account settings, and then select **Create**. It takes a few minutes to create the account. Wait for the portal page to display **Your deployment is complete**.

7. Select **Go to resource** to go to the Azure Cosmos DB account page.

Add a container

You can now use the Data Explorer tool in the Azure portal to create a database and container.

1. Select **Data Explorer** > **New Container**.

   The **Add Container** area is displayed on the far right, you may need to scroll right to see it.

2. In the **Add container** page, enter the settings for the new container.

SETTING	SUGGESTED VALUE	DESCRIPTION
Database ID	Tasks	Enter *Tasks* as the name for the new database. Database names must contain from 1 through 255 characters, and they cannot contain `/, \\, #, ?`, or a trailing space. Check the **Provision database throughput** option, it allows you to share the throughput provisioned to the database across all the containers within the database. This option also helps with cost savings.
Throughput	400	Leave the throughput at 400 request units per second (RU/s). If you want to reduce latency, you can scale up the throughput later.
Container ID	Items	Enter *Items* as the name for your new container. Container IDs have the same character requirements as database names.
Partition key	/category	The sample described in this article uses */category* as the partition key.

In addition to the preceding settings, you can optionally add **Unique keys** for the container. Let's leave the field empty in this example. Unique keys provide developers with the ability to add a layer of data integrity to the database. By creating a unique key policy while creating a container, you ensure the uniqueness of one or

more values per partition key. To learn more, refer to the Unique keys in Azure Cosmos DB article.

Select **OK**. The Data Explorer displays the new database and container.

Add sample data

You can now add data to your new container using Data Explorer.

1. From the **Data Explorer**, expand the **Tasks** database, expand the **Items** container. Select **Items**, and then select **New Item**.

2. Now add a document to the container with the following structure.

```
{
    "id": "1",
    "category": "personal",
    "name": "groceries",
    "description": "Pick up apples and strawberries.",
    "isComplete": false
}
```

3. Once you've added the json to the **Documents** tab, select **Save**.

4. Create and save one more document where you insert a unique value for the `id` property, and change the other properties as you see fit. Your new documents can have any structure you want as Azure Cosmos DB doesn't impose any schema on your data.

Query your data

You can use queries in Data Explorer to retrieve and filter your data.

1. At the top of the **Items** tab in Data Explorer, review the default query `SELECT * FROM c`. This query retrieves and displays all documents from the container ordered by ID.

2. To change the query, select **Edit Filter**, replace the default query with `ORDER BY c._ts DESC`, and then select **Apply Filter**.

The modified query displays the documents in descending order based on their time stamp, so now your second document is listed first.

If you're familiar with SQL syntax, you can enter any supported SQL queries in the query predicate box. You can also use Data Explorer to create stored procedures, UDFs, and triggers for server-side business logic.

Data Explorer provides easy Azure portal access to all of the built-in programmatic data access features available in the APIs. You also use the portal to scale throughput, get keys and connection strings, and review metrics and SLAs for your Azure Cosmos DB account.

Clone the sample application

Now let's switch to working with code. Let's clone a SQL API app from GitHub, set the connection string, and run it. You'll see how easy it is to work with data programmatically.

Run the following command to clone the sample repository. This command creates a copy of the sample app on your computer.

```
git clone https://github.com/Azure-Samples/azure-cosmos-java-getting-started.git
```

Review the code

This step is optional. If you're interested in learning how the database resources are created in the code, you can review the following snippets. Otherwise, you can skip ahead to Run the app .

- Sync API
- Async API

Managing database resources using the synchronous (sync) API

- `CosmosClient` initialization. The `CosmosClient` provides client-side logical representation for the Azure Cosmos database service. This client is used to configure and execute requests against the service.

```
client = new CosmosClientBuilder()
    .endpoint(AccountSettings.HOST)
    .key(AccountSettings.MASTER_KEY)
    //  Setting the preferred location to Cosmos DB Account region
    //  West US is just an example. User should set preferred location to the Cosmos DB region closest
to the application
    .preferredRegions(Collections.singletonList("West US"))
    .consistencyLevel(ConsistencyLevel.EVENTUAL)
    .buildClient();
```

- `CosmosDatabase` creation.

```
CosmosDatabaseResponse cosmosDatabaseResponse = client.createDatabaseIfNotExists(databaseName);
database = client.getDatabase(cosmosDatabaseResponse.getProperties().getId());
```

- `CosmosContainer` creation.

```
CosmosContainerProperties containerProperties =
    new CosmosContainerProperties(containerName, "/lastName");

//  Create container with 400 RU/s
CosmosContainerResponse cosmosContainerResponse =
    database.createContainerIfNotExists(containerProperties,
ThroughputProperties.createManualThroughput(400));
container = database.getContainer(cosmosContainerResponse.getProperties().getId());
```

- Item creation by using the `createItem` method.

```
//  Create item using container that we created using sync client

//  Use lastName as partitionKey for cosmos item
//  Using appropriate partition key improves the performance of database operations
CosmosItemRequestOptions cosmosItemRequestOptions = new CosmosItemRequestOptions();
CosmosItemResponse<Family> item = container.createItem(family, new PartitionKey(family.getLastName()),
cosmosItemRequestOptions);
```

- Point reads are performed using `readItem` method.

```
try {
    CosmosItemResponse<Family> item = container.readItem(family.getId(), new
PartitionKey(family.getLastName()), Family.class);
    double requestCharge = item.getRequestCharge();
    Duration requestLatency = item.getDuration();
    logger.info("Item successfully read with id {} with a charge of {} and within duration {}",
        item.getItem().getId(), requestCharge, requestLatency);
} catch (CosmosException e) {
    logger.error("Read Item failed with", e);
}
```

- SQL queries over JSON are performed using the `queryItems` method.

```
// Set some common query options
CosmosQueryRequestOptions queryOptions = new CosmosQueryRequestOptions();
//queryOptions.setEnableCrossPartitionQuery(true); //No longer necessary in SDK v4
//  Set query metrics enabled to get metrics around query executions
queryOptions.setQueryMetricsEnabled(true);

CosmosPagedIterable<Family> familiesPagedIterable = container.queryItems(
    "SELECT * FROM Family WHERE Family.lastName IN ('Andersen', 'Wakefield', 'Johnson')", queryOptions,
Family.class);

familiesPagedIterable.iterableByPage(10).forEach(cosmosItemPropertiesFeedResponse -> {
    logger.info("Got a page of query result with {} items(s) and request charge of {}",
            cosmosItemPropertiesFeedResponse.getResults().size(),
cosmosItemPropertiesFeedResponse.getRequestCharge());

    logger.info("Item Ids {}", cosmosItemPropertiesFeedResponse
        .getResults()
        .stream()
        .map(Family::getId)
        .collect(Collectors.toList()));
});
```

Run the app

Now go back to the Azure portal to get your connection string information and launch the app with your endpoint information. This enables your app to communicate with your hosted database.

1. In the git terminal window, `cd` to the sample code folder.

   ```
   cd azure-cosmos-java-getting-started
   ```

2. In the git terminal window, use the following command to install the required Java packages.

   ```
   mvn package
   ```

3. In the git terminal window, use the following command to start the Java application (replace SYNCASYNCMODE with `sync` or `async` depending on which sample code you would like to run, replace YOUR_COSMOS_DB_HOSTNAME with the quoted URI value from the portal, and replace YOUR_COSMOS_DB_MASTER_KEY with the quoted primary key from portal)

   ```
   mvn exec:java@SYNCASYNCMODE -DACCOUNT_HOST=YOUR_COSMOS_DB_HOSTNAME -
   DACCOUNT_KEY=YOUR_COSMOS_DB_MASTER_KEY
   ```

   The terminal window displays a notification that the FamilyDB database was created.

4. The app creates database with name `AzureSampleFamilyDB`
5. The app creates container with name `FamilyContainer`
6. The app will perform point reads using object IDs and partition key value (which is lastName in our sample).
7. The app will query items to retrieve all families with last name in ('Andersen', 'Wakefield', 'Johnson')
8. The app doesn't delete the created resources. Switch back to the portal to clean up the resources. from your account so that you don't incur charges.

Review SLAs in the Azure portal

The Azure portal monitors your Cosmos DB account throughput, storage, availability, latency, and consistency. Charts for metrics associated with an Azure Cosmos DB Service Level Agreement (SLA) show the SLA value compared to actual performance. This suite of metrics makes monitoring your SLAs transparent.

To review metrics and SLAs:

1. Select **Metrics** in your Cosmos DB account's navigation menu.
2. Select a tab such as **Latency**, and select a timeframe on the right. Compare the **Actual** and **SLA** lines on the charts.

3. Review the metrics on the other tabs.

Clean up resources

When you're done with your app and Azure Cosmos DB account, you can delete the Azure resources you created so you don't incur more charges. To delete the resources:

1. In the Azure portal Search bar, search for and select **Resource groups**.
2. From the list, select the resource group you created for this quickstart.

3. On the resource group **Overview** page, select **Delete resource group**.

4. In the next window, enter the name of the resource group to delete, and then select **Delete**.

Next steps

In this quickstart, you've learned how to create an Azure Cosmos DB SQL API account, create a document database and container using the Data Explorer, and run a Java app to do the same thing programmatically. You can now import additional data into your Azure Cosmos DB account.

Import data into Azure Cosmos DB

Quickstart: Build a Spring Data Azure Cosmos DB v3 app to manage Azure Cosmos DB SQL API data

11/2/2020 • 11 minutes to read • Edit Online

APPLIES TO: SQL API

In this quickstart, you create and manage an Azure Cosmos DB SQL API account from the Azure portal, and by using a Spring Data Azure Cosmos DB v3 app cloned from GitHub. First, you create an Azure Cosmos DB SQL API account using the Azure portal, then create a Spring Boot app using the Spring Data Azure Cosmos DB v3 connector, and then add resources to your Cosmos DB account by using the Spring Boot application. Azure Cosmos DB is a multi-model database service that lets you quickly create and query document, table, key-value, and graph databases with global distribution and horizontal scale capabilities.

> **IMPORTANT**
>
> These release notes are for version 3 of Spring Data Azure Cosmos DB. You can find release notes for version 2 here.
>
> Spring Data Azure Cosmos DB supports only the SQL API.
>
> See these articles for information about Spring Data on other Azure Cosmos DB APIs:
>
> - Spring Data for Apache Cassandra with Azure Cosmos DB
> - Spring Data MongoDB with Azure Cosmos DB
> - Spring Data Gremlin with Azure Cosmos DB

Prerequisites

- An Azure account with an active subscription. Create one for free. Or try Azure Cosmos DB for free without an Azure subscription. You can also use the Azure Cosmos DB Emulator with a URI of `https://localhost:8081` and the key `C2y6yDjf5/R+ob0N8A7Cgv30VRDJIWEHLM+4QDU5DE2nQ9nDuVTqobD4b8mGGyPMbIZnqyMsEcaGQy67XIw/Jw==`.
- Java Development Kit (JDK) 8. Point your `JAVA_HOME` environment variable to the folder where the JDK is installed.
- A Maven binary archive. On Ubuntu, run `apt-get install maven` to install Maven.
- Git. On Ubuntu, run `sudo apt-get install git` to install Git.

Introductory notes

The structure of a Cosmos DB account. Irrespective of API or programming language, a Cosmos DB *account* contains zero or more *databases*, a *database* (DB) contains zero or more *containers*, and a *container* contains zero or more items, as shown in the diagram below:

You may read more about databases, containers and items here. A few important properties are defined at the level of the container, among them *provisioned throughput* and *partition key*.

The provisioned throughput is measured in Request Units (*RUs*) which have a monetary price and are a substantial determining factor in the operating cost of the account. Provisioned throughput can be selected at per-container granularity or per-database granularity, however container-level throughput specification is typically preferred. You may read more about throughput provisioning here.

As items are inserted into a Cosmos DB container, the database grows horizontally by adding more storage and compute to handle requests. Storage and compute capacity are added in discrete units known as *partitions*, and you must choose one field in your documents to be the partition key which maps each document to a partition. The way partitions are managed is that each partition is assigned a roughly equal slice out of the range of partition key values; therefore you are advised to choose a partition key which is relatively random or evenly-distributed. Otherwise, some partitions will see substantially more requests (*hot partition*) while other partitions see substantially fewer requests (*cold partition*), and this is to be avoided. You may learn more about partitioning here.

Create a database account

Before you can create a document database, you need to create a SQL API account with Azure Cosmos DB.

1. From the Azure portal menu or the **Home page**, select **Create a resource**.
2. On the **New** page, search for and select **Azure Cosmos DB**.
3. On the **Azure Cosmos DB** page, select **Create**.
4. On the **Create Azure Cosmos DB Account** page, enter the basic settings for the new Azure Cosmos account.

SETTING	VALUE	DESCRIPTION
Subscription	Subscription name	Select the Azure subscription that you want to use for this Azure Cosmos account.
Resource Group	Resource group name	Select a resource group, or select **Create new**, then enter a unique name for the new resource group.

SETTING	VALUE	DESCRIPTION
Account Name	A unique name	Enter a name to identify your Azure Cosmos account. Because *documents.azure.com* is appended to the name that you provide to create your URI, use a unique name. The name can only contain lowercase letters, numbers, and the hyphen (-) character. It must be between 3-44 characters in length.
API	The type of account to create	Select **Core (SQL)** to create a document database and query by using SQL syntax. The API determines the type of account to create. Azure Cosmos DB provides five APIs: Core (SQL) and MongoDB for document data, Gremlin for graph data, Azure Table, and Cassandra. Currently, you must create a separate account for each API.
Capacity mode	Provisioned throughput or Serverless	Select **Provisioned throughput** to create an account in provisioned throughput mode. Select **Serverless** to create an account in serverless mode.
Apply Free Tier Discount	Apply or Do not apply	With Azure Cosmos DB free tier, you will get the first 400 RU/s and 5 GB of storage for free in an account. Learn more about free tier.
Location	The region closest to your users	Select a geographic location to host your Azure Cosmos DB account. Use the location that is closest to your users to give them the fastest access to the data.
Account Type	Production or Non-Production	Select **Production** if the account will be used for a production workload. Select **Non-Production** if the account will be used for non-production, e.g. development, testing, QA, or staging. This is an Azure resource tag setting that tunes the Portal experience but does not affect the underlying Azure Cosmos DB account. You can change this value anytime.

NOTE

You can have up to one free tier Azure Cosmos DB account per Azure subscription and must opt-in when creating the account. If you do not see the option to apply the free tier discount, this means another account in the subscription has already been enabled with free tier.

NOTE

The following options are not available if you select **Serverless** as the **Capacity mode**:

- Apply Free Tier Discount
- Geo-redundancy
- Multi-region Writes

5. Select **Review + create**. You can skip the **Network** and **Tags** sections.

6. Review the account settings, and then select **Create**. It takes a few minutes to create the account. Wait for the portal page to display **Your deployment is complete**.

7. Select **Go to resource** to go to the Azure Cosmos DB account page.

Add a container

You can now use the Data Explorer tool in the Azure portal to create a database and container.

1. Select **Data Explorer** > **New Container**.

   The **Add Container** area is displayed on the far right, you may need to scroll right to see it.

2. In the **Add container** page, enter the settings for the new container.

SETTING	SUGGESTED VALUE	DESCRIPTION
Database ID	Tasks	Enter *Tasks* as the name for the new database. Database names must contain from 1 through 255 characters, and they cannot contain `/, \\, #, ?`, or a trailing space. Check the **Provision database throughput** option, it allows you to share the throughput provisioned to the database across all the containers within the database. This option also helps with cost savings.
Throughput	400	Leave the throughput at 400 request units per second (RU/s). If you want to reduce latency, you can scale up the throughput later.
Container ID	Items	Enter *Items* as the name for your new container. Container IDs have the same character requirements as database names.
Partition key	/category	The sample described in this article uses */category* as the partition key.

In addition to the preceding settings, you can optionally add **Unique keys** for the container. Let's leave the field empty in this example. Unique keys provide developers with the ability to add a layer of data integrity to the database. By creating a unique key policy while creating a container, you ensure the uniqueness of

one or more values per partition key. To learn more, refer to the Unique keys in Azure Cosmos DB article.

Select **OK**. The Data Explorer displays the new database and container.

Add sample data

You can now add data to your new container using Data Explorer.

1. From the **Data Explorer**, expand the **Tasks** database, expand the **Items** container. Select **Items**, and then select **New Item**.

2. Now add a document to the container with the following structure.

```
{
    "id": "1",
    "category": "personal",
    "name": "groceries",
    "description": "Pick up apples and strawberries.",
    "isComplete": false
}
```

3. Once you've added the json to the **Documents** tab, select **Save**.

4. Create and save one more document where you insert a unique value for the `id` property, and change the other properties as you see fit. Your new documents can have any structure you want as Azure Cosmos DB doesn't impose any schema on your data.

Query your data

You can use queries in Data Explorer to retrieve and filter your data.

1. At the top of the **Items** tab in Data Explorer, review the default query `SELECT * FROM c`. This query retrieves and displays all documents from the container ordered by ID.

2. To change the query, select **Edit Filter**, replace the default query with `ORDER BY c._ts DESC`, and then select **Apply Filter**.

The modified query displays the documents in descending order based on their time stamp, so now your second document is listed first.

If you're familiar with SQL syntax, you can enter any supported SQL queries in the query predicate box. You can also use Data Explorer to create stored procedures, UDFs, and triggers for server-side business logic.

Data Explorer provides easy Azure portal access to all of the built-in programmatic data access features available in the APIs. You also use the portal to scale throughput, get keys and connection strings, and review metrics and SLAs for your Azure Cosmos DB account.

Clone the sample application

Now let's switch to working with code. Let's clone a SQL API app from GitHub, set the connection string, and run it. You'll see how easy it is to work with data programmatically.

Run the following command to clone the sample repository. This command creates a copy of the sample app on your computer.

```
git clone https://github.com/Azure-Samples/azure-spring-data-cosmos-java-sql-api-getting-started.git
```

Review the code

This step is optional. If you're interested in learning how the database resources are created in the code, you can review the following snippets. Otherwise, you can skip ahead to Run the app .

Application configuration file

Here we showcase how Spring Boot and Spring Data enhance user experience - the process of establishing a Cosmos client and connecting to Cosmos resources is now config rather than code. At application startup Spring Boot handles all of this boilerplate using the settings in **application.properties**:

```
cosmos.uri=${ACCOUNT_HOST}
cosmos.key=${ACCOUNT_KEY}
cosmos.secondaryKey=${SECONDARY_ACCOUNT_KEY}

dynamic.collection.name=spel-property-collection
# Populate query metrics
cosmos.queryMetricsEnabled=true
```

Once you create an Azure Cosmos DB account, database, and container, just fill-in-the-blanks in the config file and Spring Boot/Spring Data will automatically do the following: (1) create an underlying Java SDK `CosmosClient` instance with the URI and key, and (2) connect to the database and container. You're all set - **no more resource management code!**

Java source

The Spring Data value-add also comes from its simple, clean, standardized and platform-independent interface for operating on datastores. Building on the Spring Data GitHub sample linked above, below are CRUD and query samples for manipulating Azure Cosmos DB documents with Spring Data Azure Cosmos DB.

- Item creation and updates by using the `save` method.

```
logger.info("Saving user : {}", testUser1);
userRepository.save(testUser1);
```

- Point-reads using the derived query method defined in the repository. The `findByIdAndLastName` performs point-reads for `UserRepository`. The fields mentioned in the method name cause Spring Data to execute a point-read defined by the `id` and `lastName` fields:

```
// to find by Id, please specify partition key value if collection is partitioned
final User result = userRepository.findByIdAndLastName(testUser1.getId(), testUser1.getLastName());
logger.info("Found user : {}", result);
```

- Item deletes using `deleteAll`:

```
userRepository.deleteAll();
```

- Derived query based on repository method name. Spring Data implements the `UserRepository` `findByFirstName` method as a Java SDK SQL query on the `firstName` field (this query could not be implemented as a point-read):

```
Flux<User> users = reactiveUserRepository.findByFirstName("testFirstName");
users.map(u -> {
    logger.info("user is : {}", u);
    return u;
}).subscribe();
```

Run the app

Now go back to the Azure portal to get your connection string information and launch the app with your endpoint information. This enables your app to communicate with your hosted database.

1. In the git terminal window, `cd` to the sample code folder.

```
cd azure-spring-data-cosmos-java-sql-api-getting-started/azure-spring-data-cosmos-java-getting-started/
```

2. In the git terminal window, use the following command to install the required Spring Data Azure Cosmos DB packages.

```
mvn clean package
```

3. In the git terminal window, use the following command to start the Spring Data Azure Cosmos DB application:

```
mvn spring-boot:run
```

4. The app loads **application.properties** and connects the resources in your Azure Cosmos DB account.
5. The app will perform point CRUD operations described above.
6. The app will perform a derived query.
7. The app doesn't delete your resources. Switch back to the portal to clean up the resources from your account if you want to avoid incurring charges.

Review SLAs in the Azure portal

The Azure portal monitors your Cosmos DB account throughput, storage, availability, latency, and consistency. Charts for metrics associated with an Azure Cosmos DB Service Level Agreement (SLA) show the SLA value compared to actual performance. This suite of metrics makes monitoring your SLAs transparent.

To review metrics and SLAs:

1. Select **Metrics** in your Cosmos DB account's navigation menu.
2. Select a tab such as **Latency**, and select a timeframe on the right. Compare the **Actual** and **SLA** lines on the charts.

3. Review the metrics on the other tabs.

Clean up resources

When you're done with your app and Azure Cosmos DB account, you can delete the Azure resources you created so you don't incur more charges. To delete the resources:

1. In the Azure portal Search bar, search for and select **Resource groups**.

2. From the list, select the resource group you created for this quickstart.

3. On the resource group **Overview** page, select **Delete resource group**.

4. In the next window, enter the name of the resource group to delete, and then select **Delete**.

Next steps

In this quickstart, you've learned how to create an Azure Cosmos DB SQL API account, create a document database and container using the Data Explorer, and run a Spring Data app to do the same thing programmatically. You can now import additional data into your Azure Cosmos DB account.

Import data into Azure Cosmos DB

Quickstart: Use Node.js to connect and query data from Azure Cosmos DB SQL API account

11/2/2020 • 7 minutes to read • Edit Online

APPLIES TO: SQL API

In this quickstart, you create and manage an Azure Cosmos DB SQL API account from the Azure portal, and by using a Node.js app cloned from GitHub. Azure Cosmos DB is a multi-model database service that lets you quickly create and query document, table, key-value, and graph databases with global distribution and horizontal scale capabilities.

Walkthrough video

Watch this video for a complete walkthrough of the content in this article.

Prerequisites

- An Azure account with an active subscription. Create one for free. Or try Azure Cosmos DB for free without an Azure subscription. You can also use the Azure Cosmos DB Emulator with a URI of `https://localhost:8081` and the key `C2y6yDjf5/R+ob0N8A7Cgv30VRDJIWEHLM+4QDU5DE2nQ9nDuVTqobD4b8mGGyPMbIZnqyMsEcaGQy67XIw/Jw==`.
- Node.js 6.0.0+.
- Git.

Create an Azure Cosmos account

For this quickstart purpose, you can use the try Azure Cosmos DB for free option to create an Azure Cosmos account.

1. Navigate to the try Azure Cosmos DB for free page.
2. Choose the **SQL** API account and select **Create**. Sign-in using your Microsoft account.
3. After the sign-in is successful, your Azure Cosmos account should be ready. Select **Open in the Azure portal** to open the newly created account.

The "try Azure Cosmos DB for free" option doesn't require an Azure subscription and it offers you an Azure Cosmos account for a limited period of 30 days. If you want to use the Azure Cosmos account for a longer period, you should instead create the account within your Azure subscription.

Add a container

You can now use the Data Explorer tool in the Azure portal to create a database and container.

1. Select **Data Explorer** > **New Container**.

   The **Add Container** area is displayed on the far right, you may need to scroll right to see it.

2. In the **Add container** page, enter the settings for the new container.

SETTING	SUGGESTED VALUE	DESCRIPTION
Database ID	Tasks	Enter *Tasks* as the name for the new database. Database names must contain from 1 through 255 characters, and they cannot contain `/, \\, #, ?`, or a trailing space. Check the **Provision database throughput** option, it allows you to share the throughput provisioned to the database across all the containers within the database. This option also helps with cost savings.
Throughput	400	Leave the throughput at 400 request units per second (RU/s). If you want to reduce latency, you can scale up the throughput later.
Container ID	Items	Enter *Items* as the name for your new container. Container IDs have the same character requirements as database names.
Partition key	/category	The sample described in this article uses */category* as the partition key.

In addition to the preceding settings, you can optionally add **Unique keys** for the container. Let's leave the field empty in this example. Unique keys provide developers with the ability to add a layer of data integrity to the database. By creating a unique key policy while creating a container, you ensure the uniqueness of

one or more values per partition key. To learn more, refer to the Unique keys in Azure Cosmos DB article.

Select **OK**. The Data Explorer displays the new database and container.

Add sample data

You can now add data to your new container using Data Explorer.

1. From the **Data Explorer**, expand the **Tasks** database, expand the **Items** container. Select **Items**, and then select **New Item**.

2. Now add a document to the container with the following structure.

```
{
    "id": "1",
    "category": "personal",
    "name": "groceries",
    "description": "Pick up apples and strawberries.",
    "isComplete": false
}
```

3. Once you've added the json to the **Documents** tab, select **Save**.

4. Create and save one more document where you insert a unique value for the `id` property, and change the other properties as you see fit. Your new documents can have any structure you want as Azure Cosmos DB doesn't impose any schema on your data.

Query your data

You can use queries in Data Explorer to retrieve and filter your data.

1. At the top of the **Items** tab in Data Explorer, review the default query `SELECT * FROM c`. This query retrieves and displays all documents from the container ordered by ID.

2. To change the query, select **Edit Filter**, replace the default query with `ORDER BY c._ts DESC`, and then select **Apply Filter**.

The modified query displays the documents in descending order based on their time stamp, so now your second document is listed first.

If you're familiar with SQL syntax, you can enter any supported SQL queries in the query predicate box. You can also use Data Explorer to create stored procedures, UDFs, and triggers for server-side business logic.

Data Explorer provides easy Azure portal access to all of the built-in programmatic data access features available in the APIs. You also use the portal to scale throughput, get keys and connection strings, and review metrics and SLAs for your Azure Cosmos DB account.

Clone the sample application

Now let's clone a Node.js app from GitHub, set the connection string, and run it.

1. Run the following command to clone the sample repository. This command creates a copy of the sample app on your computer.

```
git clone https://github.com/Azure-Samples/azure-cosmos-db-sql-api-nodejs-getting-started.git
```

Review the code

This step is optional. If you're interested in learning how the Azure Cosmos database resources are created in the code, you can review the following snippets. Otherwise, you can skip ahead to Update your connection string.

If you're familiar with the previous version of the SQL JavaScript SDK, you may be used to seeing the terms *collection* and *document*. Because Azure Cosmos DB supports multiple API models, version 2.0+ of the JavaScript SDK uses the generic terms *container*, which may be a collection, graph, or table, and *item* to describe the content of the container.

The Cosmos DB JavaScript SDK is called "@azure/cosmos" and can be installed from npm...

```
npm install @azure/cosmos
```

The following snippets are all taken from the *app.js* file.

- The `CosmosClient` is imported from the `@azure/cosmos` npm package.

```
const CosmosClient = require("@azure/cosmos").CosmosClient;
```

- A new `CosmosClient` object is initialized.

```
const client = new CosmosClient({ endpoint, key });
```

- Select the "Tasks" database.

```
const database = client.database(databaseId);
```

- Select the "Items" container/collection.

```
const container = database.container(containerId);
```

- Select all the items in the "Items" container.

```
// query to return all items
const querySpec = {
  query: "SELECT * from c"
};

const { resources: items } = await container.items
  .query(querySpec)
  .fetchAll();
```

- Create a new item

```
const { resource: createdItem } = await container.items.create(newItem);
```

- Update an item

```
const { id, category } = createdItem;

createdItem.isComplete = true;
const { resource: updatedItem } = await container
  .item(id, category)
  .replace(createdItem);
```

- Delete an item

```
const { resource: result } = await container.item(id, category).delete();
```

> **NOTE**
> In both the "update" and "delete" methods, the item has to be selected from the database by calling `container.item()`. The two parameters passed in are the id of the item and the item's partition key. In this case, the parition key is the value of the "category" field.

Update your connection string

Now go back to the Azure portal to get the connection string details of your Azure Cosmos account. Copy the connection string into the app so that it can connect to your database.

1. In your Azure Cosmos DB account in the Azure portal, select **Keys** from the left navigation, and then select

Read-write Keys. Use the copy buttons on the right side of the screen to copy the URI and Primary Key into the *app.js* file in the next step.

2. In Open the *config.js* file.

3. Copy your URI value from the portal (using the copy button) and make it the value of the endpoint key in *config.js*.

```
endpoint: "<Your Azure Cosmos account URI>"
```

4. Then copy your PRIMARY KEY value from the portal and make it the value of the `config.key` in *config.js*. You've now updated your app with all the info it needs to communicate with Azure Cosmos DB.

```
key: "<Your Azure Cosmos account key>"
```

Run the app

1. Run `npm install` in a terminal to install the "@azure/cosmos" npm package

2. Run `node app.js` in a terminal to start your node application.

3. The two items that you created earlier in this quickstart are listed out. A new item is created. The "isComplete" flag on that item is updated to "true" and then finally, the item is deleted.

You can continue to experiment with this sample application or go back to Data Explorer, modify, and work with your data.

Review SLAs in the Azure portal

The Azure portal monitors your Cosmos DB account throughput, storage, availability, latency, and consistency. Charts for metrics associated with an Azure Cosmos DB Service Level Agreement (SLA) show the SLA value compared to actual performance. This suite of metrics makes monitoring your SLAs transparent.

To review metrics and SLAs:

1. Select **Metrics** in your Cosmos DB account's navigation menu.

2. Select a tab such as **Latency**, and select a timeframe on the right. Compare the **Actual** and **SLA** lines on

the charts.

3. Review the metrics on the other tabs.

Next steps

In this quickstart, you've learned how to create an Azure Cosmos DB account, create a container using the Data Explorer, and run a Node.js app. You can now import additional data to your Azure Cosmos DB account.

import data into azure cosmos db

Quickstart: Build a Python application using an Azure Cosmos DB SQL API account

11/2/2020 • 10 minutes to read • Edit Online

APPLIES TO: SQL API

In this quickstart, you create and manage an Azure Cosmos DB SQL API account from the Azure portal, and from Visual Studio Code with a Python app cloned from GitHub. Azure Cosmos DB is a multi-model database service that lets you quickly create and query document, table, key-value, and graph databases with global distribution and horizontal scale capabilities.

Prerequisites

- A Cosmos DB Account. You options are:
  - Within an Azure active subscription:
    - Create an Azure free Account or use your existing subscription
    - Visual Studio Monthly Credits
    - Azure Cosmos DB Free Tier
  - Without an Azure active subscription:
    - Try Azure Cosmos DB for free, a tests environment that lasts for 30 days.
    - Azure Cosmos DB Emulator
- Python 2.7 or 3.5.3+, with the `python` executable in your `PATH`.
- Visual Studio Code.
- The Python extension for Visual Studio Code.
- Git.
- Azure Cosmos DB SQL API SDK for Python

Create a database account

1. From the Azure portal menu or the **Home page**, select **Create a resource**.
2. On the **New** page, search for and select **Azure Cosmos DB**.
3. On the **Azure Cosmos DB** page, select **Create**.
4. On the **Create Azure Cosmos DB Account** page, enter the basic settings for the new Azure Cosmos account.

SETTING	VALUE	DESCRIPTION
Subscription	Subscription name	Select the Azure subscription that you want to use for this Azure Cosmos account.
Resource Group	Resource group name	Select a resource group, or select **Create new**, then enter a unique name for the new resource group.

SETTING	VALUE	DESCRIPTION
Account Name	A unique name	Enter a name to identify your Azure Cosmos account. Because *documents.azure.com* is appended to the name that you provide to create your URI, use a unique name. The name can only contain lowercase letters, numbers, and the hyphen (-) character. It must be between 3-44 characters in length.
API	The type of account to create	Select **Core (SQL)** to create a document database and query by using SQL syntax. The API determines the type of account to create. Azure Cosmos DB provides five APIs: Core (SQL) and MongoDB for document data, Gremlin for graph data, Azure Table, and Cassandra. Currently, you must create a separate account for each API.
Capacity mode	Provisioned throughput or Serverless	Select **Provisioned throughput** to create an account in provisioned throughput mode. Select **Serverless** to create an account in serverless mode.
Apply Free Tier Discount	Apply or Do not apply	With Azure Cosmos DB free tier, you will get the first 400 RU/s and 5 GB of storage for free in an account. Learn more about free tier.
Location	The region closest to your users	Select a geographic location to host your Azure Cosmos DB account. Use the location that is closest to your users to give them the fastest access to the data.
Account Type	Production or Non-Production	Select **Production** if the account will be used for a production workload. Select **Non-Production** if the account will be used for non-production, e.g. development, testing, QA, or staging. This is an Azure resource tag setting that tunes the Portal experience but does not affect the underlying Azure Cosmos DB account. You can change this value anytime.

NOTE

You can have up to one free tier Azure Cosmos DB account per Azure subscription and must opt-in when creating the account. If you do not see the option to apply the free tier discount, this means another account in the subscription has already been enabled with free tier.

NOTE

The following options are not available if you select **Serverless** as the **Capacity mode**:

- Apply Free Tier Discount
- Geo-redundancy
- Multi-region Writes

5. Select **Review + create**. You can skip the **Network** and **Tags** sections.

6. Review the account settings, and then select **Create**. It takes a few minutes to create the account. Wait for the portal page to display **Your deployment is complete**.

7. Select **Go to resource** to go to the Azure Cosmos DB account page.

Add a container

You can now use the Data Explorer tool in the Azure portal to create a database and container.

1. Select **Data Explorer** > **New Container**.

   The **Add Container** area is displayed on the far right, you may need to scroll right to see it.

2. In the **Add container** page, enter the settings for the new container.

SETTING	SUGGESTED VALUE	DESCRIPTION
Database ID	Tasks	Enter *Tasks* as the name for the new database. Database names must contain from 1 through 255 characters, and they cannot contain `/, \\, #, ?`, or a trailing space. Check the **Provision database throughput** option, it allows you to share the throughput provisioned to the database across all the containers within the database. This option also helps with cost savings.
Throughput	400	Leave the throughput at 400 request units per second (RU/s). If you want to reduce latency, you can scale up the throughput later.
Container ID	Items	Enter *Items* as the name for your new container. Container IDs have the same character requirements as database names.
Partition key	/category	The sample described in this article uses */category* as the partition key.

In addition to the preceding settings, you can optionally add **Unique keys** for the container. Let's leave the field empty in this example. Unique keys provide developers with the ability to add a layer of data integrity to the database. By creating a unique key policy while creating a container, you ensure the uniqueness of one or more values per partition key. To learn more, refer to the Unique keys in Azure Cosmos DB article.

Select **OK**. The Data Explorer displays the new database and container.

Add sample data

You can now add data to your new container using Data Explorer.

1. From the **Data Explorer**, expand the **Tasks** database, expand the **Items** container. Select **Items**, and then select **New Item**.

2. Now add a document to the container with the following structure.

```
{
    "id": "1",
    "category": "personal",
    "name": "groceries",
    "description": "Pick up apples and strawberries.",
    "isComplete": false
}
```

3. Once you've added the json to the **Documents** tab, select **Save**.

4. Create and save one more document where you insert a unique value for the `id` property, and change

the other properties as you see fit. Your new documents can have any structure you want as Azure Cosmos DB doesn't impose any schema on your data.

Query your data

You can use queries in Data Explorer to retrieve and filter your data.

1. At the top of the **Items** tab in Data Explorer, review the default query `SELECT * FROM c`. This query retrieves and displays all documents from the container ordered by ID.

2. To change the query, select **Edit Filter**, replace the default query with `ORDER BY c._ts DESC`, and then select **Apply Filter**.

The modified query displays the documents in descending order based on their time stamp, so now your second document is listed first.

If you're familiar with SQL syntax, you can enter any supported SQL queries in the query predicate box. You can

also use Data Explorer to create stored procedures, UDFs, and triggers for server-side business logic.

Data Explorer provides easy Azure portal access to all of the built-in programmatic data access features available in the APIs. You also use the portal to scale throughput, get keys and connection strings, and review metrics and SLAs for your Azure Cosmos DB account.

Clone the sample application

Now let's clone a SQL API app from GitHub, set the connection string, and run it. This quickstart uses version 4 of the Python SDK.

1. Open a command prompt, create a new folder named git-samples, then close the command prompt.

   ```
   md "git-samples"
   ```

   If you are using a bash prompt, you should instead use the following command:

   ```
   mkdir "git-samples"
   ```

2. Open a git terminal window, such as git bash, and use the `cd` command to change to the new folder to install the sample app.

   ```
   cd "git-samples"
   ```

3. Run the following command to clone the sample repository. This command creates a copy of the sample app on your computer.

   ```
   git clone https://github.com/Azure-Samples/azure-cosmos-db-python-getting-started.git
   ```

Update your connection string

Now go back to the Azure portal to get your connection string information and copy it into the app.

1. In your Azure Cosmos DB account in the Azure portal, select **Keys** from the left navigation. Use the copy buttons on the right side of the screen to copy the **URI** and **Primary Key** into the *cosmos_get_started.py* file in the next step.

2. In Visual Studio Code, open the *cosmos_get_started.py* file in *\git-samples\azure-cosmos-db-python-getting-started*.

3. Copy your **URI** value from the portal (using the copy button) and make it the value of the **endpoint** variable in *cosmos_get_started.py*.

```
endpoint = 'https://FILLME.documents.azure.com',
```

4. Then copy your **PRIMARY KEY** value from the portal and make it the value of the **key** in *cosmos_get_started.py*. You've now updated your app with all the info it needs to communicate with Azure Cosmos DB.

```
key = 'FILLME'
```

5. Save the *cosmos_get_started.py* file.

Review the code

This step is optional. Learn about the database resources created in code, or skip ahead to Update your connection string.

The following snippets are all taken from the *cosmos_get_started.py* file.

- The CosmosClient is initialized. Make sure to update the "endpoint" and "key" values as described in the Update your connection string section.

```
client = CosmosClient(endpoint, key)
```

- A new database is created.

```
database_name = 'AzureSampleFamilyDatabase'
database = client.create_database_if_not_exists(id=database_name)
```

- A new container is created, with 400 RU/s of provisioned throughput. We choose `lastName` as the partition key, which allows us to do efficient queries that filter on this property.

```
container_name = 'FamilyContainer'
container = database.create_container_if_not_exists(
    id=container_name,
    partition_key=PartitionKey(path="/lastName"),
    offer_throughput=400
)
```

- Some items are added to the container. Containers are a collection of items (JSON documents) that can have varied schema. The helper methods `get_[name]_family_item` return representations of a family that are stored in Azure Cosmos DB as JSON documents.

```
for family_item in family_items_to_create:
    container.create_item(body=family_item)
```

- Point reads (key value lookups) are performed using the `read_item` method. We print out the RU charge of each operation.

```
for family in family_items_to_create:
    item_response = container.read_item(item=family['id'], partition_key=family['lastName'])
    request_charge = container.client_connection.last_response_headers['x-ms-request-charge']
    print('Read item with id {0}. Operation consumed {1} request units'.format(item_response['id'],
(request_charge)))
```

- A query is performed using SQL query syntax. Because we're using partition key values of `lastName` in the WHERE clause, Azure Cosmos DB will efficiently route this query to the relevant partitions, improving performance.

```
query = "SELECT * FROM c WHERE c.lastName IN ('Wakefield', 'Andersen')"

items = list(container.query_items(
    query=query,
    enable_cross_partition_query=True
))

request_charge = container.client_connection.last_response_headers['x-ms-request-charge']

print('Query returned {0} items. Operation consumed {1} request units'.format(len(items),
request_charge))
```

Run the app

1. In Visual Studio Code, select **View** > **Command Palette**.
2. At the prompt, enter **Python: Select Interpreter** and then select the version of Python to use.

   The Footer in Visual Studio Code is updated to indicate the interpreter selected.
3. Select **View** > **Integrated Terminal** to open the Visual Studio Code integrated terminal.
4. In the integrated terminal window, ensure you are in the *azure-cosmos-db-python-getting-started* folder. If not, run the following command to switch to the sample folder.

```
cd "\git-samples\azure-cosmos-db-python-getting-started"`
```

5. Run the following command to install the azure-cosmos package.

```
pip install --pre azure-cosmos
```

If you get an error about access being denied when attempting to install azure-cosmos, you'll need to run VS Code as an administrator.

6. Run the following command to run the sample and create and store new documents in Azure Cosmos DB.

```
python cosmos_get_started.py
```

7. To confirm the new items were created and saved, in the Azure portal, select **Data Explorer** > **AzureSampleFamilyDatabase** > **Items**. View the items that were created. For example, here is a sample JSON document for the Andersen family:


```
{
    "id": "Andersen-1569479288379",
    "lastName": "Andersen",
    "district": "WA5",
    "parents": [
        {
            "familyName": null,
            "firstName": "Thomas"
        },
        {
            "familyName": null,
            "firstName": "Mary Kay"
        }
    ],
    "children": null,
    "address": {
        "state": "WA",
        "county": "King",
        "city": "Seattle"
    },
    "registered": true,
    "_rid": "8K5qAIYtZXeBhB4AAAAAAA==",
    "_self": "dbs/8K5qAA==/colls/8K5qAIYtZXc=/docs/8K5qAIYtZXeBhB4AAAAAAA==/",
    "_etag": "\"a3004d78-0000-0800-0000-5d8c5a780000\"",
    "_attachments": "attachments/",
    "_ts": 1569479288
}
```


Review SLAs in the Azure portal

The Azure portal monitors your Cosmos DB account throughput, storage, availability, latency, and consistency. Charts for metrics associated with an Azure Cosmos DB Service Level Agreement (SLA) show the SLA value compared to actual performance. This suite of metrics makes monitoring your SLAs transparent.

To review metrics and SLAs:

1. Select **Metrics** in your Cosmos DB account's navigation menu.

2. Select a tab such as **Latency**, and select a timeframe on the right. Compare the **Actual** and **SLA** lines on the charts.

3. Review the metrics on the other tabs.

Clean up resources

When you're done with your app and Azure Cosmos DB account, you can delete the Azure resources you created so you don't incur more charges. To delete the resources:

1. In the Azure portal Search bar, search for and select **Resource groups**.

2. From the list, select the resource group you created for this quickstart.

3. On the resource group **Overview** page, select **Delete resource group**.

4. In the next window, enter the name of the resource group to delete, and then select **Delete**.

Next steps

In this quickstart, you've learned how to create an Azure Cosmos DB account, create a container using the Data Explorer, and run a Python app in Visual Studio Code. You can now import additional data to your Azure Cosmos DB account.

Import data into Azure Cosmos DB for the SQL API

Quickstart: Build a todo app with Xamarin using Azure Cosmos DB SQL API account

11/2/2020 • 13 minutes to read • Edit Online

APPLIES TO: SQL API

Azure Cosmos DB is Microsoft's globally distributed multi-model database service. You can quickly create and query document, key/value, and graph databases, all of which benefit from the global distribution and horizontal scale capabilities at the core of Azure Cosmos DB.

> **NOTE**
> Sample code for an entire canonical sample Xamarin app showcasing multiple Azure offerings, including CosmosDB, can be found on GitHub here. This app demonstrates viewing geographically dispersed contacts, and allowing those contacts to update their location.

This quickstart demonstrates how to create an Azure Cosmos DB SQL API account, document database, and container using the Azure portal. You'll then build and deploy a todo list mobile app built on the SQL .NET API and Xamarin utilizing Xamarin.Forms and the MVVM architectural pattern.

Prerequisites

If you are developing on Windows and don't already have Visual Studio 2019 installed, you can download and use the **free** Visual Studio 2019 Community Edition. Make sure that you enable **Azure development** and **Mobile Development with .NET** workloads during the Visual Studio setup.

If you are using a Mac, you can download the **free** Visual Studio for Mac.

If you don't have an Azure subscription, create a free account before you begin.

You can Try Azure Cosmos DB for free without an Azure subscription, free of charge and commitments, or create an Azure Cosmos DB free tier account, with the first 400 RU/s and 5 GB of storage for free. You can also use the Azure Cosmos DB Emulator with a URI of `https://localhost:8081`. For the key to use with the emulator, see Authenticating requests.

Create a database account

1. From the Azure portal menu or the **Home page**, select **Create a resource**.
2. On the **New** page, search for and select **Azure Cosmos DB**.
3. On the **Azure Cosmos DB** page, select **Create**.
4. On the **Create Azure Cosmos DB Account** page, enter the basic settings for the new Azure Cosmos account.

SETTING	VALUE	DESCRIPTION
Subscription	Subscription name	Select the Azure subscription that you want to use for this Azure Cosmos account.
Resource Group	Resource group name	Select a resource group, or select **Create new**, then enter a unique name for the new resource group.
Account Name	A unique name	Enter a name to identify your Azure Cosmos account. Because *documents.azure.com* is appended to the name that you provide to create your URI, use a unique name. The name can only contain lowercase letters, numbers, and the hyphen (-) character. It must be between 3-44 characters in length.
API	The type of account to create	Select **Core (SQL)** to create a document database and query by using SQL syntax. The API determines the type of account to create. Azure Cosmos DB provides five APIs: Core (SQL) and MongoDB for document data, Gremlin for graph data, Azure Table, and Cassandra. Currently, you must create a separate account for each API.

SETTING	VALUE	DESCRIPTION
Capacity mode	Provisioned throughput or Serverless	Select **Provisioned throughput** to create an account in provisioned throughput mode. Select **Serverless** to create an account in serverless mode.
Apply Free Tier Discount	Apply or Do not apply	With Azure Cosmos DB free tier, you will get the first 400 RU/s and 5 GB of storage for free in an account. Learn more about free tier.
Location	The region closest to your users	Select a geographic location to host your Azure Cosmos DB account. Use the location that is closest to your users to give them the fastest access to the data.
Account Type	Production or Non-Production	Select **Production** if the account will be used for a production workload. Select **Non-Production** if the account will be used for non-production, e.g. development, testing, QA, or staging. This is an Azure resource tag setting that tunes the Portal experience but does not affect the underlying Azure Cosmos DB account. You can change this value anytime.

NOTE

You can have up to one free tier Azure Cosmos DB account per Azure subscription and must opt-in when creating the account. If you do not see the option to apply the free tier discount, this means another account in the subscription has already been enabled with free tier.

NOTE

The following options are not available if you select **Serverless** as the **Capacity mode**:

- Apply Free Tier Discount
- Geo-redundancy
- Multi-region Writes

5. Select **Review + create**. You can skip the **Network** and **Tags** sections.

6. Review the account settings, and then select **Create**. It takes a few minutes to create the account. Wait for the portal page to display **Your deployment is complete**.

7. Select **Go to resource** to go to the Azure Cosmos DB account page.

Add a container

You can now use the Data Explorer tool in the Azure portal to create a database and container.

1. Select **Data Explorer** > **New Container**.

   The **Add Container** area is displayed on the far right, you may need to scroll right to see it.

2. In the **Add container** page, enter the settings for the new container.

SETTING	SUGGESTED VALUE	DESCRIPTION

SETTING	SUGGESTED VALUE	DESCRIPTION
Database ID	Tasks	Enter *Tasks* as the name for the new database. Database names must contain from 1 through 255 characters, and they cannot contain `/, \\, #, ?`, or a trailing space. Check the **Provision database throughput** option, it allows you to share the throughput provisioned to the database across all the containers within the database. This option also helps with cost savings.
Throughput	400	Leave the throughput at 400 request units per second (RU/s). If you want to reduce latency, you can scale up the throughput later.
Container ID	Items	Enter *Items* as the name for your new container. Container IDs have the same character requirements as database names.
Partition key	/category	The sample described in this article uses */category* as the partition key.

In addition to the preceding settings, you can optionally add **Unique keys** for the container. Let's leave the field empty in this example. Unique keys provide developers with the ability to add a layer of data integrity to the database. By creating a unique key policy while creating a container, you ensure the uniqueness of one or more values per partition key. To learn more, refer to the Unique keys in Azure Cosmos DB article.

Select **OK**. The Data Explorer displays the new database and container.

Add sample data

You can now add data to your new container using Data Explorer.

1. From the **Data Explorer**, expand the **Tasks** database, expand the **Items** container. Select **Items**, and then select **New Item**.

2. Now add a document to the container with the following structure.

```
{
    "id": "1",
    "category": "personal",
    "name": "groceries",
    "description": "Pick up apples and strawberries.",
    "isComplete": false
}
```

3. Once you've added the json to the **Documents** tab, select **Save**.

4. Create and save one more document where you insert a unique value for the `id` property, and change the other properties as you see fit. Your new documents can have any structure you want as Azure Cosmos DB doesn't impose any schema on your data.

Query your data

You can use queries in Data Explorer to retrieve and filter your data.

1. At the top of the **Items** tab in Data Explorer, review the default query `SELECT * FROM c`. This query retrieves and displays all documents from the container ordered by ID.

2. To change the query, select **Edit Filter**, replace the default query with `ORDER BY c._ts DESC`, and then select **Apply Filter**.

The modified query displays the documents in descending order based on their time stamp, so now your second document is listed first.

If you're familiar with SQL syntax, you can enter any supported SQL queries in the query predicate box. You can also use Data Explorer to create stored procedures, UDFs, and triggers for server-side business logic.

Data Explorer provides easy Azure portal access to all of the built-in programmatic data access features available in the APIs. You also use the portal to scale throughput, get keys and connection strings, and review metrics and SLAs for your Azure Cosmos DB account.

Clone the sample application

Now let's clone the Xamarin SQL API app from GitHub, review the code, obtain the API keys, and run it. You'll see how easy it is to work with data programmatically.

1. Open a command prompt, create a new folder named git-samples, then close the command prompt.

```
mkdir "C:\git-samples"
```

2. Open a git terminal window, such as git bash, and use the `cd` command to change to the new folder to install the sample app.

```
cd "C:\git-samples"
```

3. Run the following command to clone the sample repository. This command creates a copy of the sample app on your computer.

```
git clone https://github.com/Azure-Samples/azure-cosmos-db-sql-xamarin-getting-started.git
```

4. In Visual Studio, open **C:\git-samples\azure-cosmos-db-sql-xamarin-getting-started\src\ToDoItems.sln**

Obtain your API keys

Go back to the Azure portal to get your API key information and copy it into the app.

1. In the Azure portal, in your Azure Cosmos DB SQL API account, in the left navigation click **Keys**, and then click **Read-write Keys**. You'll use the copy buttons on the right side of the screen to copy the URI and Primary Key into the APIKeys.cs file in the next step.

2. In Visual Studio, open **ToDoItems.Core/Helpers/APIKeys.cs**.

3. In the Azure Portal, using the copy button, copy the **URI** value and make it the value of the `CosmosEndpointUrl` variable in APIKeys.cs.

```
//#error Enter the URL of your Azure Cosmos DB endpoint here
        public static readonly string CosmosEndpointUrl = "[URI Copied from Azure Portal]";
```

4. In the Azure Portal, using the copy button, copy the **PRIMARY KEY** value and make it the value of the `Cosmos Auth Key` in APIKeys.cs.

```
//#error Enter the read/write authentication key of your Azure Cosmos DB endpoint here
        public static readonly string CosmosAuthKey = "[PRIMARY KEY copied from Azure Portal";
```

> **IMPORTANT**
>
> This quick start hard codes the Azure Cosmos DB authentication key for the sake of demonstration purposes. It's not recommended to hard code an authentication key when you are using it in a production app. To learn how to access Azure Cosmos DB in a securely by using a resource token, view the Authenticating users with Azure Cosmos DB article.

Review the code

This solution demonstrates how to create a ToDo app using the Azure Cosmos DB SQL API and Xamarin.Forms. The app has two tabs, the first tab contains a list view showing todo items that are not yet complete. The second tab displays todo items that have been completed. In addition to viewing not completed todo items in the first tab, you can also add new todo items, edit existing ones, and mark items as completed.

The code in the ToDoItems solution contains:

- **ToDoItems.Core**
  - This is a .NET Standard project holding a Xamarin.Forms project and shared application logic code that maintains todo items within Azure Cosmos DB.
- **ToDoItems.Android**
  - This project contains the Android app.
- **ToDoItems.iOS**
  - This project contains the iOS app.

Now let's take a quick review of how the app communicates with Azure Cosmos DB.

- The Microsoft.Azure.DocumentDb.Core NuGet package is required to be added to all projects.
- The `ToDoItem` class in the **ToDoItems.Core/Models** folder models the documents in the **Items** container created above. Note that property naming is case-sensitive.
- The `CosmosDBService` class in the **ToDoItems.Core/Services** folder encapsulates the communication to Azure Cosmos DB.
- Within the `CosmosDBService` class there is a `DocumentClient` type variable. The `DocumentClient` is used to configure and execute requests against the Azure Cosmos DB account, and is instantiated:

  ```
  docClient = new DocumentClient(new Uri(APIKeys.CosmosEndpointUrl), APIKeys.CosmosAuthKey);
  ```

- When querying a container for documents, the `DocumentClient.CreateDocumentQuery<T>` method is used, as seen here in the `CosmosDBService.GetToDoItems` function:

```
/// <summary>
/// </summary>
/// <returns></returns>
public async static Task<List<ToDoItem>> GetToDoItems()
{
    var todos = new List<ToDoItem>();

    if (!await Initialize())
        return todos;

    var todoQuery = docClient.CreateDocumentQuery<ToDoItem>(
        UriFactory.CreateDocumentCollectionUri(databaseName, collectionName),
        new FeedOptions { MaxItemCount = -1, EnableCrossPartitionQuery = true })
        .Where(todo => todo.Completed == false)
        .AsDocumentQuery();

    while (todoQuery.HasMoreResults)
    {
        var queryResults = await todoQuery.ExecuteNextAsync<ToDoItem>();

        todos.AddRange(queryResults);
    }

    return todos;
}
```

The `CreateDocumentQuery<T>` takes a URI that points to the container created in the previous section. And you are also able to specify LINQ operators such as a `Where` clause. In this case only todo items that are not completed are returned.

The `CreateDocumentQuery<T>` function is executed synchronously, and returns an `IQueryable<T>`. However, the `AsDocumentQuery` method converts the `IQueryable<T>` to an `IDocumentQuery<T>` object which can be executed asynchronously. Thus not blocking the UI thread for mobile applications.

The `IDocumentQuery<T>.ExecuteNextAsync<T>` function retrieves the page of results from Azure Cosmos DB, which `HasMoreResults` checking to see if additional results remain to be returned.

> **TIP**
> Several functions that operate on Azure Cosmos containers and documents take an URI as a parameter which specifies the address of the container or document. This URI is constructed using the `URIFactory` class. URIs for databases, containers, and documents can all be created with this class.

- The `ComsmosDBService.InsertToDoItem` function demonstrates how to insert a new document:

```
/// <summary>
/// </summary>
/// <returns></returns>
public async static Task InsertToDoItem(ToDoItem item)
{
    if (!await Initialize())
        return;

    await docClient.CreateDocumentAsync(
        UriFactory.CreateDocumentCollectionUri(databaseName, collectionName),
        item);
}
```

The item URI is specified as well as the item to be inserted.

- The `CosmosDBService.UpdateToDoItem` function demonstrates how to replace an existing document with a new one:

  ```
  /// <summary>
  /// </summary>
  /// <returns></returns>
  public async static Task UpdateToDoItem(ToDoItem item)
  {
      if (!await Initialize())
          return;

      var docUri = UriFactory.CreateDocumentUri(databaseName, collectionName, item.Id);
      await docClient.ReplaceDocumentAsync(docUri, item);
  }
  ```

  Here a new URI is needed to uniquely identify the document to replace and is obtained by using `UriFactory.CreateDocumentUri` and passing it the database and container names and the ID of the document.

  The `DocumentClient.ReplaceDocumentAsync` replaces the document identified by the URI with the one specified as a parameter.

- Deleting an item is demonstrated with the `CosmosDBService.DeleteToDoItem` function:

  ```
  /// <summary>
  /// </summary>
  /// <returns></returns>
  public async static Task DeleteToDoItem(ToDoItem item)
  {
      if (!await Initialize())
          return;

      var docUri = UriFactory.CreateDocumentUri(databaseName, collectionName, item.Id);
      await docClient.DeleteDocumentAsync(docUri);
  }
  ```

  Again note the unique document URI being created and passed to the `DocumentClient.DeleteDocumentAsync` function.

Run the app

You've now updated your app with all the info it needs to communicate with Azure Cosmos DB.

The following steps will demonstrate how to run the app using the Visual Studio for Mac debugger.

> **NOTE**
>
> Usage of the Android version app is exactly the same, any differences will be called out in the steps below. If you wish to debug with Visual Studio on Windows, documentation todo so can be found for iOS here and Android here.

1. First select the platform you wish to target by clicking on the dropdown highlighted and selecting either ToDoItems.iOS for iOS or ToDoItems.Android for Android.

2. To start debugging the app, either press cmd+Enter or click the play button.

3. When the iOS simulator or Android emulator finishes launching, the app will display 2 tabs at the bottom of the screen for iOS and the top of the screen for Android. The first shows todo items which are not completed, the second shows todo items which are completed.

4. To complete a todo item on iOS, slide it to the left > tap on the **Complete** button. To complete a todo item on Android, long press the item > then tap on the complete button.

5. To edit a todo item > tap on the item > a new screen appears letting you enter new values. Tapping the save button will persist the changes to Azure Cosmos DB.

6. To add a todo item > tap on the **Add** button on the upper right of the home screen > a new, blank, edit page will appear.

Review SLAs in the Azure portal

The Azure portal monitors your Cosmos DB account throughput, storage, availability, latency, and consistency. Charts for metrics associated with an Azure Cosmos DB Service Level Agreement (SLA) show the SLA value compared to actual performance. This suite of metrics makes monitoring your SLAs transparent.

To review metrics and SLAs:

1. Select **Metrics** in your Cosmos DB account's navigation menu.

2. Select a tab such as **Latency**, and select a timeframe on the right. Compare the **Actual** and **SLA** lines on the charts.

3. Review the metrics on the other tabs.

Clean up resources

When you're done with your app and Azure Cosmos DB account, you can delete the Azure resources you created so you don't incur more charges. To delete the resources:

1. In the Azure portal Search bar, search for and select **Resource groups**.

2. From the list, select the resource group you created for this quickstart.

3. On the resource group **Overview** page, select **Delete resource group**.

4. In the next window, enter the name of the resource group to delete, and then select **Delete**.

Next steps

In this quickstart, you've learned how to create an Azure Cosmos account, create a container using the Data Explorer, and build and deploy a Xamarin app. You can now import additional data to your Azure Cosmos account.

Import data into Azure Cosmos DB

Tutorial: Build a .NET console app to manage data in Azure Cosmos DB SQL API account

11/2/2020 • 19 minutes to read • Edit Online

APPLIES TO: SQL API

Welcome to the Azure Cosmos DB SQL API get started tutorial. After following this tutorial, you'll have a console application that creates and queries Azure Cosmos DB resources.

This tutorial uses version 3.0 or later of the Azure Cosmos DB .NET SDK. You can work with .NET Framework or .NET Core.

This tutorial covers:

- Creating and connecting to an Azure Cosmos account
- Configuring your project in Visual Studio
- Creating a database and a container
- Adding items to the container
- Querying the container
- Performing create, read, update, and delete (CRUD) operations on the item
- Deleting the database

Don't have time? Don't worry! The complete solution is available on GitHub. Jump to the Get the complete tutorial solution section for quick instructions.

Now let's get started!

Prerequisites

- An active Azure account. If you don't have one, you can sign up for a free account.

  You can Try Azure Cosmos DB for free without an Azure subscription, free of charge and commitments, or create an Azure Cosmos DB free tier account, with the first 400 RU/s and 5 GB of storage for free. You can also use the Azure Cosmos DB Emulator with a URI of `https://localhost:8081`. For the key to use with the emulator, see Authenticating requests.

- Download and use the free Visual Studio 2019 Community Edition. Make sure that you enable the **Azure development** workload during the Visual Studio setup.

Step 1: Create an Azure Cosmos DB account

Let's create an Azure Cosmos DB account. If you already have an account you want to use, skip this section. To use the Azure Cosmos DB Emulator, follow the steps at Azure Cosmos DB Emulator to set up the emulator. Then skip ahead to Step 2: Set up your Visual Studio project.

1. From the Azure portal menu or the **Home page**, select **Create a resource**.

2. On the **New** page, search for and select **Azure Cosmos DB**.

3. On the **Azure Cosmos DB** page, select **Create**.

4. On the **Create Azure Cosmos DB Account** page, enter the basic settings for the new Azure Cosmos

account.

SETTING	VALUE	DESCRIPTION
Subscription	Subscription name	Select the Azure subscription that you want to use for this Azure Cosmos account.
Resource Group	Resource group name	Select a resource group, or select **Create new**, then enter a unique name for the new resource group.
Account Name	A unique name	Enter a name to identify your Azure Cosmos account. Because *documents.azure.com* is appended to the name that you provide to create your URI, use a unique name. The name can only contain lowercase letters, numbers, and the hyphen (-) character. It must be between 3-44 characters in length.
API	The type of account to create	Select **Core (SQL)** to create a document database and query by using SQL syntax. The API determines the type of account to create. Azure Cosmos DB provides five APIs: Core (SQL) and MongoDB for document data, Gremlin for graph data, Azure Table, and Cassandra. Currently, you must create a separate account for each API.
Capacity mode	Provisioned throughput or Serverless	Select **Provisioned throughput** to create an account in provisioned throughput mode. Select **Serverless** to create an account in serverless mode.
Apply Free Tier Discount	Apply or Do not apply	With Azure Cosmos DB free tier, you will get the first 400 RU/s and 5 GB of storage for free in an account. Learn more about free tier.
Location	The region closest to your users	Select a geographic location to host your Azure Cosmos DB account. Use the location that is closest to your users to give them the fastest access to the data.

SETTING	VALUE	DESCRIPTION
Account Type	Production or Non-Production	Select **Production** if the account will be used for a production workload. Select **Non-Production** if the account will be used for non-production, e.g. development, testing, QA, or staging. This is an Azure resource tag setting that tunes the Portal experience but does not affect the underlying Azure Cosmos DB account. You can change this value anytime.

> **NOTE**
> You can have up to one free tier Azure Cosmos DB account per Azure subscription and must opt-in when creating the account. If you do not see the option to apply the free tier discount, this means another account in the subscription has already been enabled with free tier.

> **NOTE**
> The following options are not available if you select **Serverless** as the **Capacity mode**:
>
> - Apply Free Tier Discount
> - Geo-redundancy
> - Multi-region Writes

5. Select **Review + create**. You can skip the **Network** and **Tags** sections.

6. Review the account settings, and then select **Create**. It takes a few minutes to create the account. Wait for the portal page to display **Your deployment is complete**.

7. Select **Go to resource** to go to the Azure Cosmos DB account page.

Step 2: Set up your Visual Studio project

1. Open Visual Studio and select **Create a new project**.
2. In **Create a new project**, choose **Console App (.NET Framework)** for C#, then select **Next**.
3. Name your project *CosmosGettingStartedTutorial*, and then select **Create**.

4. In the **Solution Explorer**, right-click your new console application, which is under your Visual Studio solution, and select **Manage NuGet Packages**.
5. In the **NuGet Package Manager**, select **Browse** and search for *Microsoft.Azure.Cosmos*. Choose **Microsoft.Azure.Cosmos** and select **Install**.

The package ID for the Azure Cosmos DB SQL API Client Library is Microsoft Azure Cosmos DB Client Library.

Great! Now that we finished the setup, let's start writing some code. For the completed project of this tutorial, see Developing a .NET console app using Azure Cosmos DB.

Step 3: Connect to an Azure Cosmos DB account

1. Replace the references at the beginning of your C# application in the *Program.cs* file with these references:

```
using System;
using System.Threading.Tasks;
using System.Configuration;
using System.Collections.Generic;
using System.Net;
using Microsoft.Azure.Cosmos;
```

2. Add these constants and variables into your `Program` class.

```
public class Program
{
    // ADD THIS PART TO YOUR CODE

    // The Azure Cosmos DB endpoint for running this sample.
    private static readonly string EndpointUri = "<your endpoint here>";
    // The primary key for the Azure Cosmos account.
    private static readonly string PrimaryKey = "<your primary key>";

    // The Cosmos client instance
    private CosmosClient cosmosClient;

    // The database we will create
    private Database database;

    // The container we will create.
    private Container container;

    // The name of the database and container we will create
    private string databaseId = "FamilyDatabase";
    private string containerId = "FamilyContainer";
}
```

> **NOTE**
>
> If you're familiar with the previous version of the .NET SDK, you may be familiar with the terms *collection* and *document*. Because Azure Cosmos DB supports multiple API models, version 3.0 of the .NET SDK uses the generic terms *container* and *item*. A *container* can be a collection, graph, or table. An *item* can be a document, edge/vertex, or row, and is the content inside a container. For more information, see Work with databases, containers, and items in Azure Cosmos DB.

3. Open the Azure portal. Find your Azure Cosmos DB account, and then select **Keys**.

4. In *Program.cs*, replace `<your endpoint URL>` with the value of **URI**. Replace `<your primary key>` with the value of **PRIMARY KEY**.

5. Below the **Main** method, add a new asynchronous task called **GetStartedDemoAsync**, which instantiates our new `CosmosClient`.

```
public static async Task Main(string[] args)
{
}

// ADD THIS PART TO YOUR CODE
/*
    Entry point to call methods that operate on Azure Cosmos DB resources in this sample
*/
public async Task GetStartedDemoAsync()
{
    // Create a new instance of the Cosmos Client
    this.cosmosClient = new CosmosClient(EndpointUri, PrimaryKey);
}
```

   We use **GetStartedDemoAsync** as the entry point that calls methods that operate on Azure Cosmos DB resources.

6. Add the following code to run the **GetStartedDemoAsync** asynchronous task from your **Main** method. The **Main** method catches exceptions and writes them to the console.

```
public static async Task Main(string[] args)
{
    try
    {
        Console.WriteLine("Beginning operations...\n");
        Program p = new Program();
        await p.GetStartedDemoAsync();

    }
    catch (CosmosException de)
    {
        Exception baseException = de.GetBaseException();
        Console.WriteLine("{0} error occurred: {1}", de.StatusCode, de);
    }
    catch (Exception e)
    {
        Console.WriteLine("Error: {0}", e);
    }
    finally
    {
        Console.WriteLine("End of demo, press any key to exit.");
        Console.ReadKey();
    }
}
```

7. Select F5 to run your application.

   The console displays the message: **End of demo, press any key to exit.** This message confirms that your application made a connection to Azure Cosmos DB. You can then close the console window.

Congratulations! You've successfully connected to an Azure Cosmos DB account.

Step 4: Create a database

A database is the logical container of items partitioned across containers. Either the `CreateDatabaseIfNotExistsAsync` or `CreateDatabaseAsync` method of the CosmosClient class can create a database.

1. Copy and paste the `CreateDatabaseAsync` method below your `GetStartedDemoAsync` method.

```
/// <summary>
/// Create the database if it does not exist
/// </summary>
private async Task CreateDatabaseAsync()
{
    // Create a new database
    this.database = await this.cosmosClient.CreateDatabaseIfNotExistsAsync(databaseId);
    Console.WriteLine("Created Database: {0}\n", this.database.Id);
}
```

   `CreateDatabaseAsync` creates a new database with ID `FamilyDatabase` if it doesn't already exist, that has the ID specified from the `databaseId` field.

2. Copy and paste the code below where you instantiate the CosmosClient to call the **CreateDatabaseAsync** method you just added.

```
public async Task GetStartedDemoAsync()
{
    // Create a new instance of the Cosmos Client
    this.cosmosClient = new CosmosClient(EndpointUri, PrimaryKey);

    //ADD THIS PART TO YOUR CODE
    await this.CreateDatabaseAsync();
}
```

Your *Program.cs* should now look like this, with your endpoint and primary key filled in.

```
using System;
using System.Threading.Tasks;
using System.Configuration;
using System.Collections.Generic;
using System.Net;
using Microsoft.Azure.Cosmos;

namespace CosmosGettingStartedTutorial
{
    class Program
    {
        // The Azure Cosmos DB endpoint for running this sample.
        private static readonly string EndpointUri = "<your endpoint here>";
        // The primary key for the Azure Cosmos account.
        private static readonly string PrimaryKey = "<your primary key>";

        // The Cosmos client instance
        private CosmosClient cosmosClient;

        // The database we will create
        private Database database;

        // The container we will create.
        private Container container;

        // The name of the database and container we will create
        private string databaseId = "FamilyDatabase";
        private string containerId = "FamilyContainer";

        public static async Task Main(string[] args)
        {
            try
            {
                Console.WriteLine("Beginning operations...");
                Program p = new Program();
                await p.GetStartedDemoAsync();
            }
            catch (CosmosException de)
            {
                Exception baseException = de.GetBaseException();
                Console.WriteLine("{0} error occurred: {1}\n", de.StatusCode, de);
            }
            catch (Exception e)
            {
                Console.WriteLine("Error: {0}\n", e);
            }
            finally
            {
                Console.WriteLine("End of demo, press any key to exit.");
                Console.ReadKey();
            }
        }

        /// <summary>
        /// Entry point to call methods that operate on Azure Cosmos DB resources in this sample
```

```
        /// </summary>
        public async Task GetStartedDemoAsync()
        {
            // Create a new instance of the Cosmos Client
            this.cosmosClient = new CosmosClient(EndpointUri, PrimaryKey);
            await this.CreateDatabaseAsync();
        }

        /// <summary>
        /// Create the database if it does not exist
        /// </summary>
        private async Task CreateDatabaseAsync()
        {
            // Create a new database
            this.database = await this.cosmosClient.CreateDatabaseIfNotExistsAsync(databaseId);
            Console.WriteLine("Created Database: {0}\n", this.database.Id);
        }
    }
}
```

3. Select F5 to run your application.

> **NOTE**
>
> If you get a "503 service unavailable exception" error, it's possible that the required ports for direct connectivity mode are blocked by a firewall. To fix this issue, either open the required ports or use the gateway mode connectivity as shown in the following code:

```
// Create a new instance of the Cosmos Client in Gateway mode
this.cosmosClient = new CosmosClient(EndpointUri, PrimaryKey, new CosmosClientOptions()
       {
           ConnectionMode = ConnectionMode.Gateway
       });
```

Congratulations! You've successfully created an Azure Cosmos database.

Step 5: Create a container

> **WARNING**
>
> The method `CreateContainerIfNotExistsAsync` creates a new container, which has pricing implications. For more details, please visit our pricing page.

A container can be created by using either the **CreateContainerIfNotExistsAsync** or **CreateContainerAsync** method in the `CosmosDatabase` class. A container consists of items (JSON documents if SQL API) and associated server-side application logic in JavaScript, for example, stored procedures, user-defined functions, and triggers.

1. Copy and paste the `CreateContainerAsync` method below your `CreateDatabaseAsync` method. `CreateContainerAsync` creates a new container with the ID `FamilyContainer` if it doesn't already exist, by using the ID specified from the `containerId` field partitioned by `LastName` property.

```
/// <summary>
/// Create the container if it does not exist.
/// Specifiy "/LastName" as the partition key since we're storing family information, to ensure good
distribution of requests and storage.
/// </summary>
/// <returns></returns>
private async Task CreateContainerAsync()
{
    // Create a new container
    this.container = await this.database.CreateContainerIfNotExistsAsync(containerId, "/LastName");
    Console.WriteLine("Created Container: {0}\n", this.container.Id);
}
```

2. Copy and paste the code below where you instantiated the CosmosClient to call the **CreateContainer** method you just added.

```
public async Task GetStartedDemoAsync()
{
    // Create a new instance of the Cosmos Client
    this.cosmosClient = new CosmosClient(EndpointUri, PrimaryKey);
    await this.CreateDatabaseAsync();

    //ADD THIS PART TO YOUR CODE
    await this.CreateContainerAsync();
}
```

3. Select F5 to run your application.

Congratulations! You've successfully created an Azure Cosmos container.

Step 6: Add items to the container

The **CreateItemAsync** method of the `CosmosContainer` class can create an item. When using the SQL API, items are projected as documents, which are user-defined arbitrary JSON content. You can now insert an item into your Azure Cosmos container.

First, let's create a `Family` class that represents objects stored within Azure Cosmos DB in this sample. We'll also create `Parent`, `Child`, `Pet`, `Address` subclasses that are used within `Family`. The item must have an `Id` property serialized as `id` in JSON.

1. Select Ctrl+Shift+A to open **Add New Item**. Add a new class `Family.cs` to your project.

2. Copy and paste the `Family`, `Parent`, `Child`, `Pet`, and `Address` class into `Family.cs`.

```
using Newtonsoft.Json;

namespace CosmosGettingStartedTutorial
{
    public class Family
    {
        [JsonProperty(PropertyName = "id")]
        public string Id { get; set; }
        public string LastName { get; set; }
        public Parent[] Parents { get; set; }
        public Child[] Children { get; set; }
        public Address Address { get; set; }
        public bool IsRegistered { get; set; }
        public override string ToString()
        {
            return JsonConvert.SerializeObject(this);
        }
    }

    public class Parent
    {
        public string FamilyName { get; set; }
        public string FirstName { get; set; }
    }

    public class Child
    {
        public string FamilyName { get; set; }
        public string FirstName { get; set; }
        public string Gender { get; set; }
        public int Grade { get; set; }
        public Pet[] Pets { get; set; }
    }

    public class Pet
    {
        public string GivenName { get; set; }
    }

    public class Address
    {
        public string State { get; set; }
        public string County { get; set; }
        public string City { get; set; }
    }
}
```

3. Back in *Program.cs*, add the `AddItemsToContainerAsync` method after your `CreateContainerAsync` method.

```
/// <summary>
/// Add Family items to the container
/// </summary>
private async Task AddItemsToContainerAsync()
{
    // Create a family object for the Andersen family
    Family andersenFamily = new Family
    {
        Id = "Andersen.1",
        LastName = "Andersen",
        Parents = new Parent[]
        {
            new Parent { FirstName = "Thomas" },
            new Parent { FirstName = "Mary Kay" }
        },
```

```
        Children = new Child[]
        {
            new Child
            {
                FirstName = "Henriette Thaulow",
                Gender = "female",
                Grade = 5,
                Pets = new Pet[]
                {
                    new Pet { GivenName = "Fluffy" }
                }
            }
        },
        Address = new Address { State = "WA", County = "King", City = "Seattle" },
        IsRegistered = false
    };

    try
    {
        // Read the item to see if it exists.
        ItemResponse<Family> andersenFamilyResponse = await this.container.ReadItemAsync<Family>
(andersenFamily.Id, new PartitionKey(andersenFamily.LastName));
        Console.WriteLine("Item in database with id: {0} already exists\n",
andersenFamilyResponse.Resource.Id);
    }
    catch(CosmosException ex) when (ex.StatusCode == HttpStatusCode.NotFound)
    {
        // Create an item in the container representing the Andersen family. Note we provide the value
of the partition key for this item, which is "Andersen"
        ItemResponse<Family> andersenFamilyResponse = await this.container.CreateItemAsync<Family>
(andersenFamily, new PartitionKey(andersenFamily.LastName));

        // Note that after creating the item, we can access the body of the item with the Resource
property off the ItemResponse. We can also access the RequestCharge property to see the amount of RUs
consumed on this request.
        Console.WriteLine("Created item in database with id: {0} Operation consumed {1} RUs.\n",
andersenFamilyResponse.Resource.Id, andersenFamilyResponse.RequestCharge);
    }

    // Create a family object for the Wakefield family
    Family wakefieldFamily = new Family
    {
        Id = "Wakefield.7",
        LastName = "Wakefield",
        Parents = new Parent[]
        {
            new Parent { FamilyName = "Wakefield", FirstName = "Robin" },
            new Parent { FamilyName = "Miller", FirstName = "Ben" }
        },
        Children = new Child[]
        {
            new Child
            {
                FamilyName = "Merriam",
                FirstName = "Jesse",
                Gender = "female",
                Grade = 8,
                Pets = new Pet[]
                {
                    new Pet { GivenName = "Goofy" },
                    new Pet { GivenName = "Shadow" }
                }
            },
            new Child
            {
                FamilyName = "Miller",
                FirstName = "Lisa",
                Gender = "female",
                Grade = 1
```

```
            }
        },
        Address = new Address { State = "NY", County = "Manhattan", City = "NY" },
        IsRegistered = true
    };

    try
    {
        // Read the item to see if it exists
        ItemResponse<Family> wakefieldFamilyResponse = await this.container.ReadItemAsync<Family>
(wakefieldFamily.Id, new PartitionKey(wakefieldFamily.LastName));
        Console.WriteLine("Item in database with id: {0} already exists\n",
wakefieldFamilyResponse.Resource.Id);
    }
    catch(CosmosException ex) when (ex.StatusCode == HttpStatusCode.NotFound)
    {
        // Create an item in the container representing the Wakefield family. Note we provide the value
of the partition key for this item, which is "Wakefield"
        ItemResponse<Family> wakefieldFamilyResponse = await this.container.CreateItemAsync<Family>
(wakefieldFamily, new PartitionKey(wakefieldFamily.LastName));

        // Note that after creating the item, we can access the body of the item with the Resource
property off the ItemResponse. We can also access the RequestCharge property to see the amount of RUs
consumed on this request.
        Console.WriteLine("Created item in database with id: {0} Operation consumed {1} RUs.\n",
wakefieldFamilyResponse.Resource.Id, wakefieldFamilyResponse.RequestCharge);
    }
}
```

The code checks to make sure an item with the same ID doesn't already exist. We'll insert two items, one each for the *Andersen Family* and the *Wakefield Family*.

4. Add a call to `AddItemsToContainerAsync` in the `GetStartedDemoAsync` method.

```
public async Task GetStartedDemoAsync()
{
    // Create a new instance of the Cosmos Client
    this.cosmosClient = new CosmosClient(EndpointUri, PrimaryKey);
    await this.CreateDatabaseAsync();
    await this.CreateContainerAsync();

    //ADD THIS PART TO YOUR CODE
    await this.AddItemsToContainerAsync();
}
```

5. Select F5 to run your application.

Congratulations! You've successfully created two Azure Cosmos items.

Step 7: Query Azure Cosmos DB resources

Azure Cosmos DB supports rich queries against JSON documents stored in each container. For more information, see Getting started with SQL queries. The following sample code shows how to run a query against the items we inserted in the previous step.

1. Copy and paste the `QueryItemsAsync` method after your `AddItemsToContainerAsync` method.

```
/// <summary>
/// Run a query (using Azure Cosmos DB SQL syntax) against the container
/// </summary>
private async Task QueryItemsAsync()
{
    var sqlQueryText = "SELECT * FROM c WHERE c.LastName = 'Andersen'";

    Console.WriteLine("Running query: {0}\n", sqlQueryText);

    QueryDefinition queryDefinition = new QueryDefinition(sqlQueryText);
    FeedIterator<Family> queryResultSetIterator = this.container.GetItemQueryIterator<Family>
(queryDefinition);

    List<Family> families = new List<Family>();

    while (queryResultSetIterator.HasMoreResults)
    {
        FeedResponse<Family> currentResultSet = await queryResultSetIterator.ReadNextAsync();
        foreach (Family family in currentResultSet)
        {
            families.Add(family);
            Console.WriteLine("\tRead {0}\n", family);
        }
    }
}
```

2. Add a call to `QueryItemsAsync` in the `GetStartedDemoAsync` method.

```
public async Task GetStartedDemoAsync()
{
    // Create a new instance of the Cosmos Client
    this.cosmosClient = new CosmosClient(EndpointUri, PrimaryKey);
    await this.CreateDatabaseAsync();
    await this.CreateContainerAsync();
    await this.AddItemsToContainerAsync();

    //ADD THIS PART TO YOUR CODE
    await this.QueryItemsAsync();
}
```

3. Select F5 to run your application.

Congratulations! You've successfully queried an Azure Cosmos container.

Step 8: Replace a JSON item

Now, we'll update an item in Azure Cosmos DB. We'll change the `IsRegistered` property of the `Family` and the `Grade` of one of the children.

1. Copy and paste the `ReplaceFamilyItemAsync` method after your `QueryItemsAsync` method.

```
/// <summary>
/// Replace an item in the container
/// </summary>
private async Task ReplaceFamilyItemAsync()
{
    ItemResponse<Family> wakefieldFamilyResponse = await this.container.ReadItemAsync<Family>
("Wakefield.7", new PartitionKey("Wakefield"));
    var itemBody = wakefieldFamilyResponse.Resource;

    // update registration status from false to true
    itemBody.IsRegistered = true;
    // update grade of child
    itemBody.Children[0].Grade = 6;

    // replace the item with the updated content
    wakefieldFamilyResponse = await this.container.ReplaceItemAsync<Family>(itemBody, itemBody.Id, new
PartitionKey(itemBody.LastName));
    Console.WriteLine("Updated Family [{0},{1}].\n \tBody is now: {2}\n", itemBody.LastName,
itemBody.Id, wakefieldFamilyResponse.Resource);
}
```

2. Add a call to `ReplaceFamilyItemAsync` in the `GetStartedDemoAsync` method.

```
public async Task GetStartedDemoAsync()
{
    // Create a new instance of the Cosmos Client
    this.cosmosClient = new CosmosClient(EndpointUri, PrimaryKey);
    await this.CreateDatabaseAsync();
    await this.CreateContainerAsync();
    await this.AddItemsToContainerAsync();
    await this.QueryItemsAsync();

    //ADD THIS PART TO YOUR CODE
    await this.ReplaceFamilyItemAsync();
}
```

3. Select F5 to run your application.

Congratulations! You've successfully replaced an Azure Cosmos item.

Step 9: Delete item

Now, we'll delete an item in Azure Cosmos DB.

1. Copy and paste the `DeleteFamilyItemAsync` method after your `ReplaceFamilyItemAsync` method.

```
/// <summary>
/// Delete an item in the container
/// </summary>
private async Task DeleteFamilyItemAsync()
{
    var partitionKeyValue = "Wakefield";
    var familyId = "Wakefield.7";

    // Delete an item. Note we must provide the partition key value and id of the item to delete
    ItemResponse<Family> wakefieldFamilyResponse = await this.container.DeleteItemAsync<Family>
(familyId,new PartitionKey(partitionKeyValue));
    Console.WriteLine("Deleted Family [{0},{1}]\n", partitionKeyValue, familyId);
}
```

2. Add a call to `DeleteFamilyItemAsync` in the `GetStartedDemoAsync` method.

```
public async Task GetStartedDemoAsync()
{
    // Create a new instance of the Cosmos Client
    this.cosmosClient = new CosmosClient(EndpointUri, PrimaryKey);
    await this.CreateDatabaseAsync();
    await this.CreateContainerAsync();
    await this.AddItemsToContainerAsync();
    await this.QueryItemsAsync();
    await this.ReplaceFamilyItemAsync();

    //ADD THIS PART TO YOUR CODE
    await this.DeleteFamilyItemAsync();
}
```

3. Select F5 to run your application.

Congratulations! You've successfully deleted an Azure Cosmos item.

Step 10: Delete the database

Now we'll delete our database. Deleting the created database removes the database and all children resources. The resources include containers, items, and any stored procedures, user-defined functions, and triggers. We also dispose of the `CosmosClient` instance.

1. Copy and paste the `DeleteDatabaseAndCleanupAsync` method after your `DeleteFamilyItemAsync` method.

```
/// <summary>
/// Delete the database and dispose of the Cosmos Client instance
/// </summary>
private async Task DeleteDatabaseAndCleanupAsync()
{
    DatabaseResponse databaseResourceResponse = await this.database.DeleteAsync();
    // Also valid: await this.cosmosClient.Databases["FamilyDatabase"].DeleteAsync();

    Console.WriteLine("Deleted Database: {0}\n", this.databaseId);

    //Dispose of CosmosClient
    this.cosmosClient.Dispose();
}
```

2. Add a call to `DeleteDatabaseAndCleanupAsync` in the `GetStartedDemoAsync` method.

```
/// <summary>
/// Entry point to call methods that operate on Azure Cosmos DB resources in this sample
/// </summary>
public async Task GetStartedDemoAsync()
{
    // Create a new instance of the Cosmos Client
    this.cosmosClient = new CosmosClient(EndpointUri, PrimaryKey);
    await this.CreateDatabaseAsync();
    await this.CreateContainerAsync();
    await this.AddItemsToContainerAsync();
    await this.QueryItemsAsync();
    await this.ReplaceFamilyItemAsync();
    await this.DeleteFamilyItemAsync();
    await this.DeleteDatabaseAndCleanupAsync();
}
```

3. Select F5 to run your application.

Congratulations! You've successfully deleted an Azure Cosmos database.

Step 11: Run your C# console application all together!

Select F5 in Visual Studio to build and run the application in debug mode.

You should see the output of your entire app in a console window. The output shows the results of the queries we added. It should match the example text below.

```
Beginning operations...

Created Database: FamilyDatabase

Created Container: FamilyContainer

Created item in database with id: Andersen.1 Operation consumed 11.43 RUs.

Created item in database with id: Wakefield.7 Operation consumed 14.29 RUs.

Running query: SELECT * FROM c WHERE c.LastName = 'Andersen'

        Read {"id":"Andersen.1","LastName":"Andersen","Parents":[{"FamilyName":null,"FirstName":"Thomas"},
{"FamilyName":null,"FirstName":"Mary Kay"}],"Children":[{"FamilyName":null,"FirstName":"Henriette
Thaulow","Gender":"female","Grade":5,"Pets":[{"GivenName":"Fluffy"}]}],"Address":
{"State":"WA","County":"King","City":"Seattle"},"IsRegistered":false}

Updated Family [Wakefield,Wakefield.7].
        Body is now: {"id":"Wakefield.7","LastName":"Wakefield","Parents":
[{"FamilyName":"Wakefield","FirstName":"Robin"},{"FamilyName":"Miller","FirstName":"Ben"}],"Children":
[{"FamilyName":"Merriam","FirstName":"Jesse","Gender":"female","Grade":6,"Pets":[{"GivenName":"Goofy"},
{"GivenName":"Shadow"}]},
{"FamilyName":"Miller","FirstName":"Lisa","Gender":"female","Grade":1,"Pets":null}],"Address":
{"State":"NY","County":"Manhattan","City":"NY"},"IsRegistered":true}

Deleted Family [Wakefield,Wakefield.7]

Deleted Database: FamilyDatabase

End of demo, press any key to exit.
```

Congratulations! You've completed the tutorial and have a working C# console application!

Get the complete tutorial solution

If you didn't have time to complete the steps in this tutorial, or just want to download the code samples, you can download it.

To build the `GetStarted` solution, you need the following prerequisites:

- An active Azure account. If you don't have one, you can sign up for a free account.
- An Azure Cosmos DB account.
- The GetStarted solution available on GitHub.

To restore the references to the Azure Cosmos DB .NET SDK in Visual Studio, right-click the solution in **Solution Explorer**, and then select **Restore NuGet Packages**. Next, in the *App.config* file, update the `EndPointUri` and `PrimaryKey` values as described in Step 3: Connect to an Azure Cosmos DB account.

That's it, build it, and you're on your way!

Next steps

- Want a more complex ASP.NET MVC tutorial? See Tutorial: Develop an ASP.NET Core MVC web application with Azure Cosmos DB by using .NET SDK.

- Want to do scale and performance testing with Azure Cosmos DB? See Performance and scale testing with Azure Cosmos DB.
- To learn how to monitor Azure Cosmos DB requests, usage, and storage, see Monitor performance and storage metrics in Azure Cosmos DB.
- To run queries against our sample dataset, see the Query Playground.
- To learn more about Azure Cosmos DB, see Welcome to Azure Cosmos DB.

Quickstart: Build a Java app to manage Azure Cosmos DB SQL API data

11/2/2020 • 15 minutes to read • Edit Online

APPLIES TO: SQL API

In this quickstart, you create and manage an Azure Cosmos DB SQL API account from the Azure portal, and by using a Java app cloned from GitHub. First, you create an Azure Cosmos DB SQL API account using the Azure portal, then create a Java app using the SQL Java SDK, and then add resources to your Cosmos DB account by using the Java application. Azure Cosmos DB is a multi-model database service that lets you quickly create and query document, table, key-value, and graph databases with global distribution and horizontal scale capabilities.

> **IMPORTANT**
>
> This quickstart is for Azure Cosmos DB Java SDK v4 only. Please view the Azure Cosmos DB Java SDK v4 Release notes, Maven repository, Azure Cosmos DB Java SDK v4 performance tips, and Azure Cosmos DB Java SDK v4 troubleshooting guide for more information. If you are currently using an older version than v4, see the Migrate to Azure Cosmos DB Java SDK v4 guide for help upgrading to v4.

Prerequisites

- An Azure account with an active subscription. Create one for free. Or try Azure Cosmos DB for free without an Azure subscription. You can also use the Azure Cosmos DB Emulator with a URI of `https://localhost:8081` and the key `C2y6yDjf5/R+ob0N8A7Cgv30VRDJIWEHLM+4QDU5DE2nQ9nDuVTqobD4b8mGGyPMbIZnqyMsEcaGQy67XIw/Jw==`.
- Java Development Kit (JDK) 8. Point your `JAVA_HOME` environment variable to the folder where the JDK is installed.
- A Maven binary archive. On Ubuntu, run `apt-get install maven` to install Maven.
- Git. On Ubuntu, run `sudo apt-get install git` to install Git.

Introductory notes

The structure of a Cosmos DB account. Irrespective of API or programming language, a Cosmos DB *account* contains zero or more *databases*, a *database* (DB) contains zero or more *containers*, and a *container* contains zero or more items, as shown in the diagram below:

You may read more about databases, containers and items here. A few important properties are defined at the level of the container, among them *provisioned throughput* and *partition key*.

The provisioned throughput is measured in Request Units (*RUs*) which have a monetary price and are a substantial determining factor in the operating cost of the account. Provisioned throughput can be selected at per-container granularity or per-database granularity, however container-level throughput specification is typically preferred. You may read more about throughput provisioning here.

As items are inserted into a Cosmos DB container, the database grows horizontally by adding more storage and compute to handle requests. Storage and compute capacity are added in discrete units known as *partitions*, and you must choose one field in your documents to be the partition key which maps each document to a partition. The way partitions are managed is that each partition is assigned a roughly equal slice out of the range of partition key values; therefore you are advised to choose a partition key which is relatively random or evenly-distributed. Otherwise, some partitions will see substantially more requests (*hot partition*) while other partitions see substantially fewer requests (*cold partition*), and this is to be avoided. You may learn more about partitioning here.

Create a database account

Before you can create a document database, you need to create a SQL API account with Azure Cosmos DB.

1. From the Azure portal menu or the **Home page**, select **Create a resource**.
2. On the **New** page, search for and select **Azure Cosmos DB**.
3. On the **Azure Cosmos DB** page, select **Create**.
4. On the **Create Azure Cosmos DB Account** page, enter the basic settings for the new Azure Cosmos account.

SETTING	VALUE	DESCRIPTION
Subscription	Subscription name	Select the Azure subscription that you want to use for this Azure Cosmos account.
Resource Group	Resource group name	Select a resource group, or select **Create new**, then enter a unique name for the new resource group.

SETTING	VALUE	DESCRIPTION
Account Name	A unique name	Enter a name to identify your Azure Cosmos account. Because *documents.azure.com* is appended to the name that you provide to create your URI, use a unique name. The name can only contain lowercase letters, numbers, and the hyphen (-) character. It must be between 3-44 characters in length.
API	The type of account to create	Select **Core (SQL)** to create a document database and query by using SQL syntax. The API determines the type of account to create. Azure Cosmos DB provides five APIs: Core (SQL) and MongoDB for document data, Gremlin for graph data, Azure Table, and Cassandra. Currently, you must create a separate account for each API.
Capacity mode	Provisioned throughput or Serverless	Select **Provisioned throughput** to create an account in provisioned throughput mode. Select **Serverless** to create an account in serverless mode.
Apply Free Tier Discount	Apply or Do not apply	With Azure Cosmos DB free tier, you will get the first 400 RU/s and 5 GB of storage for free in an account. Learn more about free tier.
Location	The region closest to your users	Select a geographic location to host your Azure Cosmos DB account. Use the location that is closest to your users to give them the fastest access to the data.
Account Type	Production or Non-Production	Select **Production** if the account will be used for a production workload. Select **Non-Production** if the account will be used for non-production, e.g. development, testing, QA, or staging. This is an Azure resource tag setting that tunes the Portal experience but does not affect the underlying Azure Cosmos DB account. You can change this value anytime.

> **NOTE**
>
> You can have up to one free tier Azure Cosmos DB account per Azure subscription and must opt-in when creating the account. If you do not see the option to apply the free tier discount, this means another account in the subscription has already been enabled with free tier.

> **NOTE**
>
> The following options are not available if you select **Serverless** as the **Capacity mode**:
>
> - Apply Free Tier Discount
> - Geo-redundancy
> - Multi-region Writes

5. Select **Review + create**. You can skip the **Network** and **Tags** sections.

6. Review the account settings, and then select **Create**. It takes a few minutes to create the account. Wait for the portal page to display **Your deployment is complete**.

7. Select **Go to resource** to go to the Azure Cosmos DB account page.

Add a container

You can now use the Data Explorer tool in the Azure portal to create a database and container.

1. Select **Data Explorer** > **New Container**.

   The **Add Container** area is displayed on the far right, you may need to scroll right to see it.

2. In the **Add container** page, enter the settings for the new container.

SETTING	SUGGESTED VALUE	DESCRIPTION
Database ID	Tasks	Enter *Tasks* as the name for the new database. Database names must contain from 1 through 255 characters, and they cannot contain `/, \\, #, ?`, or a trailing space. Check the **Provision database throughput** option, it allows you to share the throughput provisioned to the database across all the containers within the database. This option also helps with cost savings.
Throughput	400	Leave the throughput at 400 request units per second (RU/s). If you want to reduce latency, you can scale up the throughput later.
Container ID	Items	Enter *Items* as the name for your new container. Container IDs have the same character requirements as database names.
Partition key	/category	The sample described in this article uses */category* as the partition key.

In addition to the preceding settings, you can optionally add **Unique keys** for the container. Let's leave the field empty in this example. Unique keys provide developers with the ability to add a layer of data integrity to the database. By creating a unique key policy while creating a container, you ensure the uniqueness of one or

more values per partition key. To learn more, refer to the Unique keys in Azure Cosmos DB article.

Select **OK**. The Data Explorer displays the new database and container.

Add sample data

You can now add data to your new container using Data Explorer.

1. From the **Data Explorer**, expand the **Tasks** database, expand the **Items** container. Select **Items**, and then select **New Item**.

2. Now add a document to the container with the following structure.

```
{
    "id": "1",
    "category": "personal",
    "name": "groceries",
    "description": "Pick up apples and strawberries.",
    "isComplete": false
}
```

3. Once you've added the json to the **Documents** tab, select **Save**.

4. Create and save one more document where you insert a unique value for the `id` property, and change the other properties as you see fit. Your new documents can have any structure you want as Azure Cosmos DB doesn't impose any schema on your data.

Query your data

You can use queries in Data Explorer to retrieve and filter your data.

1. At the top of the **Items** tab in Data Explorer, review the default query `SELECT * FROM c`. This query retrieves and displays all documents from the container ordered by ID.

2. To change the query, select **Edit Filter**, replace the default query with `ORDER BY c._ts DESC`, and then select **Apply Filter**.

The modified query displays the documents in descending order based on their time stamp, so now your second document is listed first.

If you're familiar with SQL syntax, you can enter any supported SQL queries in the query predicate box. You can also use Data Explorer to create stored procedures, UDFs, and triggers for server-side business logic.

Data Explorer provides easy Azure portal access to all of the built-in programmatic data access features available in the APIs. You also use the portal to scale throughput, get keys and connection strings, and review metrics and SLAs for your Azure Cosmos DB account.

Clone the sample application

Now let's switch to working with code. Let's clone a SQL API app from GitHub, set the connection string, and run it. You'll see how easy it is to work with data programmatically.

Run the following command to clone the sample repository. This command creates a copy of the sample app on your computer.

```
git clone https://github.com/Azure-Samples/azure-cosmos-java-getting-started.git
```

Review the code

This step is optional. If you're interested in learning how the database resources are created in the code, you can review the following snippets. Otherwise, you can skip ahead to Run the app .

- Sync API
- Async API

Managing database resources using the synchronous (sync) API

- `CosmosClient` initialization. The `CosmosClient` provides client-side logical representation for the Azure Cosmos database service. This client is used to configure and execute requests against the service.

```
client = new CosmosClientBuilder()
    .endpoint(AccountSettings.HOST)
    .key(AccountSettings.MASTER_KEY)
    //  Setting the preferred location to Cosmos DB Account region
    //  West US is just an example. User should set preferred location to the Cosmos DB region closest
to the application
    .preferredRegions(Collections.singletonList("West US"))
    .consistencyLevel(ConsistencyLevel.EVENTUAL)
    .buildClient();
```

- `CosmosDatabase` creation.

```
CosmosDatabaseResponse cosmosDatabaseResponse = client.createDatabaseIfNotExists(databaseName);
database = client.getDatabase(cosmosDatabaseResponse.getProperties().getId());
```

- `CosmosContainer` creation.

```
CosmosContainerProperties containerProperties =
    new CosmosContainerProperties(containerName, "/lastName");

//  Create container with 400 RU/s
CosmosContainerResponse cosmosContainerResponse =
    database.createContainerIfNotExists(containerProperties,
ThroughputProperties.createManualThroughput(400));
container = database.getContainer(cosmosContainerResponse.getProperties().getId());
```

- Item creation by using the `createItem` method.

```
//  Create item using container that we created using sync client

//  Use lastName as partitionKey for cosmos item
//  Using appropriate partition key improves the performance of database operations
CosmosItemRequestOptions cosmosItemRequestOptions = new CosmosItemRequestOptions();
CosmosItemResponse<Family> item = container.createItem(family, new PartitionKey(family.getLastName()),
cosmosItemRequestOptions);
```

- Point reads are performed using `readItem` method.

```
try {
    CosmosItemResponse<Family> item = container.readItem(family.getId(), new
PartitionKey(family.getLastName()), Family.class);
    double requestCharge = item.getRequestCharge();
    Duration requestLatency = item.getDuration();
    logger.info("Item successfully read with id {} with a charge of {} and within duration {}",
        item.getItem().getId(), requestCharge, requestLatency);
} catch (CosmosException e) {
    logger.error("Read Item failed with", e);
}
```

- SQL queries over JSON are performed using the `queryItems` method.

```
// Set some common query options
CosmosQueryRequestOptions queryOptions = new CosmosQueryRequestOptions();
//queryOptions.setEnableCrossPartitionQuery(true); //No longer necessary in SDK v4
//  Set query metrics enabled to get metrics around query executions
queryOptions.setQueryMetricsEnabled(true);

CosmosPagedIterable<Family> familiesPagedIterable = container.queryItems(
    "SELECT * FROM Family WHERE Family.lastName IN ('Andersen', 'Wakefield', 'Johnson')", queryOptions,
Family.class);

familiesPagedIterable.iterableByPage(10).forEach(cosmosItemPropertiesFeedResponse -> {
    logger.info("Got a page of query result with {} items(s) and request charge of {}",
            cosmosItemPropertiesFeedResponse.getResults().size(),
cosmosItemPropertiesFeedResponse.getRequestCharge());

    logger.info("Item Ids {}", cosmosItemPropertiesFeedResponse
        .getResults()
        .stream()
        .map(Family::getId)
        .collect(Collectors.toList()));
});
```

Run the app

Now go back to the Azure portal to get your connection string information and launch the app with your endpoint information. This enables your app to communicate with your hosted database.

1. In the git terminal window, `cd` to the sample code folder.

   ```
   cd azure-cosmos-java-getting-started
   ```

2. In the git terminal window, use the following command to install the required Java packages.

   ```
   mvn package
   ```

3. In the git terminal window, use the following command to start the Java application (replace SYNCASYNCMODE with `sync` or `async` depending on which sample code you would like to run, replace YOUR_COSMOS_DB_HOSTNAME with the quoted URI value from the portal, and replace YOUR_COSMOS_DB_MASTER_KEY with the quoted primary key from portal)

   ```
   mvn exec:java@SYNCASYNCMODE -DACCOUNT_HOST=YOUR_COSMOS_DB_HOSTNAME -
   DACCOUNT_KEY=YOUR_COSMOS_DB_MASTER_KEY
   ```

   The terminal window displays a notification that the FamilyDB database was created.

4. The app creates database with name `AzureSampleFamilyDB`
5. The app creates container with name `FamilyContainer`
6. The app will perform point reads using object IDs and partition key value (which is lastName in our sample).
7. The app will query items to retrieve all families with last name in ('Andersen', 'Wakefield', 'Johnson')
8. The app doesn't delete the created resources. Switch back to the portal to clean up the resources. from your account so that you don't incur charges.

Review SLAs in the Azure portal

The Azure portal monitors your Cosmos DB account throughput, storage, availability, latency, and consistency. Charts for metrics associated with an Azure Cosmos DB Service Level Agreement (SLA) show the SLA value compared to actual performance. This suite of metrics makes monitoring your SLAs transparent.

To review metrics and SLAs:

1. Select **Metrics** in your Cosmos DB account's navigation menu.
2. Select a tab such as **Latency**, and select a timeframe on the right. Compare the **Actual** and **SLA** lines on the charts.

3. Review the metrics on the other tabs.

Clean up resources

When you're done with your app and Azure Cosmos DB account, you can delete the Azure resources you created so you don't incur more charges. To delete the resources:

1. In the Azure portal Search bar, search for and select **Resource groups**.
2. From the list, select the resource group you created for this quickstart.

3. On the resource group **Overview** page, select **Delete resource group**.

4. In the next window, enter the name of the resource group to delete, and then select **Delete**.

Next steps

In this quickstart, you've learned how to create an Azure Cosmos DB SQL API account, create a document database and container using the Data Explorer, and run a Java app to do the same thing programmatically. You can now import additional data into your Azure Cosmos DB account.

Import data into Azure Cosmos DB

Quickstart: Build a Java app to manage Azure Cosmos DB SQL API data

11/2/2020 • 15 minutes to read • Edit Online

APPLIES TO: SQL API

In this quickstart, you create and manage an Azure Cosmos DB SQL API account from the Azure portal, and by using a Java app cloned from GitHub. First, you create an Azure Cosmos DB SQL API account using the Azure portal, then create a Java app using the SQL Java SDK, and then add resources to your Cosmos DB account by using the Java application. Azure Cosmos DB is a multi-model database service that lets you quickly create and query document, table, key-value, and graph databases with global distribution and horizontal scale capabilities.

> **IMPORTANT**
>
> This quickstart is for Azure Cosmos DB Java SDK v4 only. Please view the Azure Cosmos DB Java SDK v4 Release notes, Maven repository, Azure Cosmos DB Java SDK v4 performance tips, and Azure Cosmos DB Java SDK v4 troubleshooting guide for more information. If you are currently using an older version than v4, see the Migrate to Azure Cosmos DB Java SDK v4 guide for help upgrading to v4.

Prerequisites

- An Azure account with an active subscription. Create one for free. Or try Azure Cosmos DB for free without an Azure subscription. You can also use the Azure Cosmos DB Emulator with a URI of `https://localhost:8081` and the key `C2y6yDjf5/R+ob0N8A7Cgv30VRDJIWEHLM+4QDU5DE2nQ9nDuVTqobD4b8mGGyPMbIZnqyMsEcaGQy67XIw/Jw==`.
- Java Development Kit (JDK) 8. Point your `JAVA_HOME` environment variable to the folder where the JDK is installed.
- A Maven binary archive. On Ubuntu, run `apt-get install maven` to install Maven.
- Git. On Ubuntu, run `sudo apt-get install git` to install Git.

Introductory notes

The structure of a Cosmos DB account. Irrespective of API or programming language, a Cosmos DB *account* contains zero or more *databases*, a *database* (DB) contains zero or more *containers*, and a *container* contains zero or more items, as shown in the diagram below:

You may read more about databases, containers and items here. A few important properties are defined at the level of the container, among them *provisioned throughput* and *partition key*.

The provisioned throughput is measured in Request Units (*RUs*) which have a monetary price and are a substantial determining factor in the operating cost of the account. Provisioned throughput can be selected at per-container granularity or per-database granularity, however container-level throughput specification is typically preferred. You may read more about throughput provisioning here.

As items are inserted into a Cosmos DB container, the database grows horizontally by adding more storage and compute to handle requests. Storage and compute capacity are added in discrete units known as *partitions*, and you must choose one field in your documents to be the partition key which maps each document to a partition. The way partitions are managed is that each partition is assigned a roughly equal slice out of the range of partition key values; therefore you are advised to choose a partition key which is relatively random or evenly-distributed. Otherwise, some partitions will see substantially more requests (*hot partition*) while other partitions see substantially fewer requests (*cold partition*), and this is to be avoided. You may learn more about partitioning here.

Create a database account

Before you can create a document database, you need to create a SQL API account with Azure Cosmos DB.

1. From the Azure portal menu or the **Home page**, select **Create a resource**.
2. On the **New** page, search for and select **Azure Cosmos DB**.
3. On the **Azure Cosmos DB** page, select **Create**.
4. On the **Create Azure Cosmos DB Account** page, enter the basic settings for the new Azure Cosmos account.

SETTING	VALUE	DESCRIPTION
Subscription	Subscription name	Select the Azure subscription that you want to use for this Azure Cosmos account.
Resource Group	Resource group name	Select a resource group, or select **Create new**, then enter a unique name for the new resource group.

SETTING	VALUE	DESCRIPTION
Account Name	A unique name	Enter a name to identify your Azure Cosmos account. Because *documents.azure.com* is appended to the name that you provide to create your URI, use a unique name. The name can only contain lowercase letters, numbers, and the hyphen (-) character. It must be between 3-44 characters in length.
API	The type of account to create	Select **Core (SQL)** to create a document database and query by using SQL syntax. The API determines the type of account to create. Azure Cosmos DB provides five APIs: Core (SQL) and MongoDB for document data, Gremlin for graph data, Azure Table, and Cassandra. Currently, you must create a separate account for each API.
Capacity mode	Provisioned throughput or Serverless	Select **Provisioned throughput** to create an account in provisioned throughput mode. Select **Serverless** to create an account in serverless mode.
Apply Free Tier Discount	Apply or Do not apply	With Azure Cosmos DB free tier, you will get the first 400 RU/s and 5 GB of storage for free in an account. Learn more about free tier.
Location	The region closest to your users	Select a geographic location to host your Azure Cosmos DB account. Use the location that is closest to your users to give them the fastest access to the data.
Account Type	Production or Non-Production	Select **Production** if the account will be used for a production workload. Select **Non-Production** if the account will be used for non-production, e.g. development, testing, QA, or staging. This is an Azure resource tag setting that tunes the Portal experience but does not affect the underlying Azure Cosmos DB account. You can change this value anytime.

> **NOTE**
>
> You can have up to one free tier Azure Cosmos DB account per Azure subscription and must opt-in when creating the account. If you do not see the option to apply the free tier discount, this means another account in the subscription has already been enabled with free tier.

> **NOTE**
>
> The following options are not available if you select **Serverless** as the **Capacity mode**:
>
> - Apply Free Tier Discount
> - Geo-redundancy
> - Multi-region Writes

5. Select **Review + create**. You can skip the **Network** and **Tags** sections.

6. Review the account settings, and then select **Create**. It takes a few minutes to create the account. Wait for the portal page to display **Your deployment is complete**.

7. Select **Go to resource** to go to the Azure Cosmos DB account page.

Add a container

You can now use the Data Explorer tool in the Azure portal to create a database and container.

1. Select **Data Explorer** > **New Container**.

   The **Add Container** area is displayed on the far right, you may need to scroll right to see it.

2. In the **Add container** page, enter the settings for the new container.

SETTING	SUGGESTED VALUE	DESCRIPTION
Database ID	Tasks	Enter *Tasks* as the name for the new database. Database names must contain from 1 through 255 characters, and they cannot contain `/, \\, #, ?`, or a trailing space. Check the **Provision database throughput** option, it allows you to share the throughput provisioned to the database across all the containers within the database. This option also helps with cost savings.
Throughput	400	Leave the throughput at 400 request units per second (RU/s). If you want to reduce latency, you can scale up the throughput later.
Container ID	Items	Enter *Items* as the name for your new container. Container IDs have the same character requirements as database names.
Partition key	/category	The sample described in this article uses */category* as the partition key.

In addition to the preceding settings, you can optionally add **Unique keys** for the container. Let's leave the field empty in this example. Unique keys provide developers with the ability to add a layer of data integrity to the database. By creating a unique key policy while creating a container, you ensure the

uniqueness of one or more values per partition key. To learn more, refer to the Unique keys in Azure Cosmos DB article.

Select **OK**. The Data Explorer displays the new database and container.

Add sample data

You can now add data to your new container using Data Explorer.

1. From the **Data Explorer**, expand the **Tasks** database, expand the **Items** container. Select **Items**, and then select **New Item**.

2. Now add a document to the container with the following structure.

```
{
    "id": "1",
    "category": "personal",
    "name": "groceries",
    "description": "Pick up apples and strawberries.",
    "isComplete": false
}
```

3. Once you've added the json to the **Documents** tab, select **Save**.

4. Create and save one more document where you insert a unique value for the `id` property, and change the other properties as you see fit. Your new documents can have any structure you want as Azure Cosmos DB doesn't impose any schema on your data.

Query your data

You can use queries in Data Explorer to retrieve and filter your data.

1. At the top of the **Items** tab in Data Explorer, review the default query `SELECT * FROM c`. This query retrieves and displays all documents from the container ordered by ID.

2. To change the query, select **Edit Filter**, replace the default query with `ORDER BY c._ts DESC`, and then select **Apply Filter**.

The modified query displays the documents in descending order based on their time stamp, so now your second document is listed first.

If you're familiar with SQL syntax, you can enter any supported SQL queries in the query predicate box. You can also use Data Explorer to create stored procedures, UDFs, and triggers for server-side business logic.

Data Explorer provides easy Azure portal access to all of the built-in programmatic data access features available in the APIs. You also use the portal to scale throughput, get keys and connection strings, and review metrics and SLAs for your Azure Cosmos DB account.

Clone the sample application

Now let's switch to working with code. Let's clone a SQL API app from GitHub, set the connection string, and run it. You'll see how easy it is to work with data programmatically.

Run the following command to clone the sample repository. This command creates a copy of the sample app on your computer.

```
git clone https://github.com/Azure-Samples/azure-cosmos-java-getting-started.git
```

Review the code

This step is optional. If you're interested in learning how the database resources are created in the code, you can review the following snippets. Otherwise, you can skip ahead to Run the app .

- Sync API
- Async API

Managing database resources using the synchronous (sync) API

- `CosmosClient` initialization. The `CosmosClient` provides client-side logical representation for the Azure Cosmos database service. This client is used to configure and execute requests against the service.

```
client = new CosmosClientBuilder()
    .endpoint(AccountSettings.HOST)
    .key(AccountSettings.MASTER_KEY)
    //  Setting the preferred location to Cosmos DB Account region
    //  West US is just an example. User should set preferred location to the Cosmos DB region
closest to the application
    .preferredRegions(Collections.singletonList("West US"))
    .consistencyLevel(ConsistencyLevel.EVENTUAL)
    .buildClient();
```

- `CosmosDatabase` creation.

```
CosmosDatabaseResponse cosmosDatabaseResponse = client.createDatabaseIfNotExists(databaseName);
database = client.getDatabase(cosmosDatabaseResponse.getProperties().getId());
```

- `CosmosContainer` creation.

```
CosmosContainerProperties containerProperties =
    new CosmosContainerProperties(containerName, "/lastName");

//  Create container with 400 RU/s
CosmosContainerResponse cosmosContainerResponse =
    database.createContainerIfNotExists(containerProperties,
ThroughputProperties.createManualThroughput(400));
container = database.getContainer(cosmosContainerResponse.getProperties().getId());
```

- Item creation by using the `createItem` method.

```
//  Create item using container that we created using sync client

//  Use lastName as partitionKey for cosmos item
//  Using appropriate partition key improves the performance of database operations
CosmosItemRequestOptions cosmosItemRequestOptions = new CosmosItemRequestOptions();
CosmosItemResponse<Family> item = container.createItem(family, new
PartitionKey(family.getLastName()), cosmosItemRequestOptions);
```

- Point reads are performed using `readItem` method.

```
try {
    CosmosItemResponse<Family> item = container.readItem(family.getId(), new
PartitionKey(family.getLastName()), Family.class);
    double requestCharge = item.getRequestCharge();
    Duration requestLatency = item.getDuration();
    logger.info("Item successfully read with id {} with a charge of {} and within duration {}",
        item.getItem().getId(), requestCharge, requestLatency);
} catch (CosmosException e) {
    logger.error("Read Item failed with", e);
}
```

- SQL queries over JSON are performed using the `queryItems` method.

```
// Set some common query options
CosmosQueryRequestOptions queryOptions = new CosmosQueryRequestOptions();
//queryOptions.setEnableCrossPartitionQuery(true); //No longer necessary in SDK v4
//  Set query metrics enabled to get metrics around query executions
queryOptions.setQueryMetricsEnabled(true);

CosmosPagedIterable<Family> familiesPagedIterable = container.queryItems(
    "SELECT * FROM Family WHERE Family.lastName IN ('Andersen', 'Wakefield', 'Johnson')",
queryOptions, Family.class);

familiesPagedIterable.iterableByPage(10).forEach(cosmosItemPropertiesFeedResponse -> {
    logger.info("Got a page of query result with {} items(s) and request charge of {}",
            cosmosItemPropertiesFeedResponse.getResults().size(),
cosmosItemPropertiesFeedResponse.getRequestCharge());

    logger.info("Item Ids {}", cosmosItemPropertiesFeedResponse
        .getResults()
        .stream()
        .map(Family::getId)
        .collect(Collectors.toList()));
});
```

Run the app

Now go back to the Azure portal to get your connection string information and launch the app with your endpoint information. This enables your app to communicate with your hosted database.

1. In the git terminal window, `cd` to the sample code folder.

   ```
   cd azure-cosmos-java-getting-started
   ```

2. In the git terminal window, use the following command to install the required Java packages.

   ```
   mvn package
   ```

3. In the git terminal window, use the following command to start the Java application (replace SYNCASYNCMODE with `sync` or `async` depending on which sample code you would like to run, replace YOUR_COSMOS_DB_HOSTNAME with the quoted URI value from the portal, and replace YOUR_COSMOS_DB_MASTER_KEY with the quoted primary key from portal)

   ```
   mvn exec:java@SYNCASYNCMODE -DACCOUNT_HOST=YOUR_COSMOS_DB_HOSTNAME -
   DACCOUNT_KEY=YOUR_COSMOS_DB_MASTER_KEY
   ```

   The terminal window displays a notification that the FamilyDB database was created.

4. The app creates database with name `AzureSampleFamilyDB`

5. The app creates container with name `FamilyContainer`

6. The app will perform point reads using object IDs and partition key value (which is lastName in our sample).

7. The app will query items to retrieve all families with last name in ('Andersen', 'Wakefield', 'Johnson')

8. The app doesn't delete the created resources. Switch back to the portal to clean up the resources. from your account so that you don't incur charges.

Review SLAs in the Azure portal

The Azure portal monitors your Cosmos DB account throughput, storage, availability, latency, and consistency. Charts for metrics associated with an Azure Cosmos DB Service Level Agreement (SLA) show the SLA value compared to actual performance. This suite of metrics makes monitoring your SLAs transparent.

To review metrics and SLAs:

1. Select **Metrics** in your Cosmos DB account's navigation menu.
2. Select a tab such as **Latency**, and select a timeframe on the right. Compare the **Actual** and **SLA** lines on the charts.

3. Review the metrics on the other tabs.

Clean up resources

When you're done with your app and Azure Cosmos DB account, you can delete the Azure resources you created so you don't incur more charges. To delete the resources:

1. In the Azure portal Search bar, search for and select **Resource groups**.
2. From the list, select the resource group you created for this quickstart.

3. On the resource group **Overview** page, select **Delete resource group**.

4. In the next window, enter the name of the resource group to delete, and then select **Delete**.

Next steps

In this quickstart, you've learned how to create an Azure Cosmos DB SQL API account, create a document database and container using the Data Explorer, and run a Java app to do the same thing programmatically. You can now import additional data into your Azure Cosmos DB account.

Import data into Azure Cosmos DB

Tutorial: Build a Node.js console app with the JavaScript SDK to manage Azure Cosmos DB SQL API data

11/2/2020 • 11 minutes to read • Edit Online

APPLIES TO: SQL API

As a developer, you might have applications that use NoSQL document data. You can use a SQL API account in Azure Cosmos DB to store and access this document data. This tutorial shows you how to build a Node.js console application to create Azure Cosmos DB resources and query them.

In this tutorial, you will:

- Create and connect to an Azure Cosmos DB account.
- Set up your application.
- Create a database.
- Create a container.
- Add items to the container.
- Perform basic operations on the items, container, and database.

Prerequisites

Make sure you have the following resources:

- An active Azure account. If you don't have one, you can sign up for a Free Azure Trial.

  You can Try Azure Cosmos DB for free without an Azure subscription, free of charge and commitments, or create an Azure Cosmos DB free tier account, with the first 400 RU/s and 5 GB of storage for free. You can also use the Azure Cosmos DB Emulator with a URI of `https://localhost:8081`. For the key to use with the emulator, see Authenticating requests.

- Node.js v6.0.0 or higher.

Create Azure Cosmos DB account

Let's create an Azure Cosmos DB account. If you already have an account you want to use, you can skip ahead to Set up your Node.js application. If you are using the Azure Cosmos DB Emulator, follow the steps at Azure Cosmos DB Emulator to set up the emulator and skip ahead to Set up your Node.js application.

1. From the Azure portal menu or the **Home page**, select **Create a resource**.
2. On the **New** page, search for and select **Azure Cosmos DB**.
3. On the **Azure Cosmos DB** page, select **Create**.
4. On the **Create Azure Cosmos DB Account** page, enter the basic settings for the new Azure Cosmos account.

SETTING	VALUE	DESCRIPTION

SETTING	VALUE	DESCRIPTION
Subscription	Subscription name	Select the Azure subscription that you want to use for this Azure Cosmos account.
Resource Group	Resource group name	Select a resource group, or select **Create new**, then enter a unique name for the new resource group.
Account Name	A unique name	Enter a name to identify your Azure Cosmos account. Because *documents.azure.com* is appended to the name that you provide to create your URI, use a unique name. The name can only contain lowercase letters, numbers, and the hyphen (-) character. It must be between 3-44 characters in length.
API	The type of account to create	Select **Core (SQL)** to create a document database and query by using SQL syntax. The API determines the type of account to create. Azure Cosmos DB provides five APIs: Core (SQL) and MongoDB for document data, Gremlin for graph data, Azure Table, and Cassandra. Currently, you must create a separate account for each API.
Capacity mode	Provisioned throughput or Serverless	Select **Provisioned throughput** to create an account in provisioned throughput mode. Select **Serverless** to create an account in serverless mode.
Apply Free Tier Discount	Apply or Do not apply	With Azure Cosmos DB free tier, you will get the first 400 RU/s and 5 GB of storage for free in an account. Learn more about free tier.
Location	The region closest to your users	Select a geographic location to host your Azure Cosmos DB account. Use the location that is closest to your users to give them the fastest access to the data.

SETTING	VALUE	DESCRIPTION
Account Type	Production or Non-Production	Select **Production** if the account will be used for a production workload. Select **Non-Production** if the account will be used for non-production, e.g. development, testing, QA, or staging. This is an Azure resource tag setting that tunes the Portal experience but does not affect the underlying Azure Cosmos DB account. You can change this value anytime.

> **NOTE**
> You can have up to one free tier Azure Cosmos DB account per Azure subscription and must opt-in when creating the account. If you do not see the option to apply the free tier discount, this means another account in the subscription has already been enabled with free tier.

> **NOTE**
> The following options are not available if you select **Serverless** as the **Capacity mode**:
>
> - Apply Free Tier Discount
> - Geo-redundancy
> - Multi-region Writes

5. Select **Review + create**. You can skip the **Network** and **Tags** sections.

6. Review the account settings, and then select **Create**. It takes a few minutes to create the account. Wait for the portal page to display **Your deployment is complete**.

7. Select **Go to resource** to go to the Azure Cosmos DB account page.

Set up your Node.js application

Before you start writing code to build the application, you can build the framework for your app. Run the following steps to set up your Node.js application that has the framework code:

1. Open your favorite terminal.
2. Locate the folder or directory where you'd like to save your Node.js application.
3. Create empty JavaScript files with the following commands:
   - Windows:
     - `fsutil file createnew app.js 0`
     - `fsutil file createnew config.js 0`
     - `md data`
     - `fsutil file createnew data\databaseContext.js 0`
   - Linux/OS X:
     - `touch app.js`
     - `touch config.js`
     - `mkdir data`
     - `touch data/databaseContext.js`
4. Create and initialize a `package.json` file. Use the following command:
   - `npm init -y`
5. Install the @azure/cosmos module via npm. Use the following command:
   - `npm install @azure/cosmos --save`

Set your app's configurations

Now that your app exists, you need to make sure it can talk to Azure Cosmos DB. By updating a few configuration settings, as shown in the following steps, you can set your app to talk to Azure Cosmos DB:

1. Open the *config.js* file in your favorite text editor.
2. Copy and paste the following code snippet into the *config.js* file and set the properties `endpoint` and `key` to your Azure Cosmos DB endpoint URI and primary key. The database, container names are set to **Tasks**

and **Items**. The partition key you will use for this application is **/category**.

```
// @ts-check

const config = {
  endpoint: "<Your Azure Cosmos account URI>",
  key: "<Your Azure Cosmos account key>",
  databaseId: "Tasks",
  containerId: "Items",
  partitionKey: { kind: "Hash", paths: ["/category"] }
};

module.exports = config;
```

You can find the endpoint and key details in the **Keys** pane of the Azure portal.

The JavaScript SDK uses the generic terms *container* and *item*. A container can be a collection, graph, or table. An item can be a document, edge/vertex, or row, and is the content inside a container. In the previous code snippet, the `module.exports = config;` code is used to export the config object, so that you can reference it within the *app.js* file.

Create a database and a container

1. Open the *databaseContext.js* file in your favorite text editor.

2. Copy and paste the following code to the *databaseContext.js* file. This code defines a function that creates the "Tasks", "Items" database and the container if they don't already exist in your Azure Cosmos account:

```
const config = require("../config");
const CosmosClient = require("@azure/cosmos").CosmosClient;

/*
// This script ensures that the database is setup and populated correctly
*/
async function create(client, databaseId, containerId) {
  const partitionKey = config.partitionKey;

  /**
   * Create the database if it does not exist
   */
  const { database } = await client.databases.createIfNotExists({
    id: databaseId
  });
  console.log(`Created database:\n${database.id}\n`);

  /**
   * Create the container if it does not exist
   */
  const { container } = await client
    .database(databaseId)
    .containers.createIfNotExists(
      { id: containerId, partitionKey },
      { offerThroughput: 400 }
    );

  console.log(`Created container:\n${container.id}\n`);
}

module.exports = { create };
```

A database is the logical container of items partitioned across containers. You create a database by using either the `createIfNotExists` or create function of the **Databases** class. A container consists of items which in the case of the SQL API is JSON documents. You create a container by using either the `createIfNotExists` or create function from the **Containers** class. After creating a container, you can store and query the data.

> **WARNING**
>
> Creating a container has pricing implications. Visit our pricing page so you know what to expect.

Import the configuration

1. Open the *app.js* file in your favorite text editor.
2. Copy and paste the code below to import the `@azure/cosmos` module, the configuration, and the databaseContext that you defined in the previous steps.

```
const CosmosClient = require("@azure/cosmos").CosmosClient;
const config = require("./config");
const dbContext = require("./data/databaseContext");
```

Connect to the Azure Cosmos account

In the *app.js* file, copy and paste the following code to use the previously saved endpoint and key to create a new CosmosClient object.

```
const { endpoint, key, databaseId, containerId } = config;

const client = new CosmosClient({ endpoint, key });

const database = client.database(databaseId);
const container = database.container(containerId);

// Make sure Tasks database is already setup. If not, create it.
await dbContext.create(client, databaseId, containerId);
```

NOTE

If connecting to the **Cosmos DB Emulator**, disable TLS verification for your node process:

```
process.env.NODE_TLS_REJECT_UNAUTHORIZED = "0";
const client = new CosmosClient({ endpoint, key });
```

Now that you have the code to initialize the Azure Cosmos DB client, let's take a look at how to work with Azure Cosmos DB resources.

Query items

Azure Cosmos DB supports rich queries against JSON items stored in each container. The following sample code shows a query that you can run against the items in your container.You can query the items by using the query function of the `Items` class. Add the following code to the *app.js* file to query the items from your Azure Cosmos account:

```
console.log(`Querying container: Items`);

// query to return all items
const querySpec = {
  query: "SELECT * from c"
};

// read all items in the Items container
const { resources: items } = await container.items
  .query(querySpec)
  .fetchAll();

items.forEach(item => {
  console.log(`${item.id} - ${item.description}`);
});
```

Create an item

An item can be created by using the create function of the `Items` class. When you're using the SQL API, items are projected as documents, which are user-defined (arbitrary) JSON content. In this tutorial, you create a new item within the tasks database.

1. In the app.js file, define the item definition:

```
const newItem = {
  id: "3",
  category: "fun",
  name: "Cosmos DB",
  description: "Complete Cosmos DB Node.js Quickstart ⚡",
  isComplete: false
};
```

2. Add the following code to create the previously defined item:

```
/** Create new item
 * newItem is defined at the top of this file
 */
const { resource: createdItem } = await container.items.create(newItem);

console.log(`\r\nCreated new item: ${createdItem.id} - ${createdItem.description}\r\n`);
```

Update an item

Azure Cosmos DB supports replacing the contents of items. Copy and paste the following code to *app.js* file. This code gets an item from the container and updates the *isComplete* field to true.

```
/** Update item
 * Pull the id and partition key value from the newly created item.
 * Update the isComplete field to true.
 */
const { id, category } = createdItem;

createdItem.isComplete = true;

const { resource: updatedItem } = await container
  .item(id, category)
  .replace(createdItem);

console.log(`Updated item: ${updatedItem.id} - ${updatedItem.description}`);
console.log(`Updated isComplete to ${updatedItem.isComplete}\r\n`);
```

Delete an item

Azure Cosmos DB supports deleting JSON items. The following code shows how to get an item by its ID and delete it. Copy and paste the following code to *app.js* file:

```
/**
 * Delete item
 * Pass the id and partition key value to delete the item
 */
const { resource: result } = await container.item(id, category).delete();
console.log(`Deleted item with id: ${id}`);
```

Run your Node.js application

Altogether, your code should look like this:

```
// @ts-check
//  <ImportConfiguration>
const CosmosClient = require("@azure/cosmos").CosmosClient;
const config = require("./config");
```

```
const dbContext = require("./data/databaseContext");
//  </ImportConfiguration>

//  <DefineNewItem>
const newItem = {
  id: "3",
  category: "fun",
  name: "Cosmos DB",
  description: "Complete Cosmos DB Node.js Quickstart ⚡",
  isComplete: false
};
//  </DefineNewItem>

async function main() {

  // <CreateClientObjectDatabaseContainer>
  const { endpoint, key, databaseId, containerId } = config;

  const client = new CosmosClient({ endpoint, key });

  const database = client.database(databaseId);
  const container = database.container(containerId);

  // Make sure Tasks database is already setup. If not, create it.
  await dbContext.create(client, databaseId, containerId);
  // </CreateClientObjectDatabaseContainer>

  try {
    // <QueryItems>
    console.log(`Querying container: Items`);

    // query to return all items
    const querySpec = {
      query: "SELECT * from c"
    };

    // read all items in the Items container
    const { resources: items } = await container.items
      .query(querySpec)
      .fetchAll();

    items.forEach(item => {
      console.log(`${item.id} - ${item.description}`);
    });
    // </QueryItems>

    // <CreateItem>
    /** Create new item
     * newItem is defined at the top of this file
     */
    const { resource: createdItem } = await container.items.create(newItem);

    console.log(`\r\nCreated new item: ${createdItem.id} - ${createdItem.description}\r\n`);
    // </CreateItem>

    // <UpdateItem>
    /** Update item
     * Pull the id and partition key value from the newly created item.
     * Update the isComplete field to true.
     */
    const { id, category } = createdItem;

    createdItem.isComplete = true;

    const { resource: updatedItem } = await container
      .item(id, category)
      .replace(createdItem);

    console.log(`Updated item: ${updatedItem.id} - ${updatedItem.description}`);
```

```
    console.log(`Updated item: ${updatedItem.id} - ${updatedItem.description}`);
    console.log(`Updated isComplete to ${updatedItem.isComplete}\r\n`);
    // </UpdateItem>

    // <DeleteItem>
    /**
     * Delete item
     * Pass the id and partition key value to delete the item
     */
    const { resource: result } = await container.item(id, category).delete();
    console.log(`Deleted item with id: ${id}`);
    // </DeleteItem>

  } catch (err) {
    console.log(err.message);
  }
}

main();
```

In your terminal, locate your `app.js` file and run the command:

```
node app.js
```

You should see the output of your get started app. The output should match the example text below.

```
Created database:
Tasks

Created container:
Items

Querying container: Items
1 - Pick up apples and strawberries.

Created new item: 3 - Complete Cosmos DB Node.js Quickstart ▯

Updated item: 3 - Complete Cosmos DB Node.js Quickstart ▯
Updated isComplete to true

Deleted item with id: 3
```

Get the complete Node.js tutorial solution

If you didn't have time to complete the steps in this tutorial, or just want to download the code, you can get it from GitHub.

To run the getting started solution that contains all the code in this article, you will need:

- An Azure Cosmos DB account.
- The Getting Started solution available on GitHub.

Install the project's dependencies via npm. Use the following command:

- `npm install`

Next, in the `config.js` file, update the config.endpoint and config.key values as described in Step 3: Set your app's configurations.

Then in your terminal, locate your `app.js` file and run the command:

```
node app.js
```

Clean up resources

When these resources are no longer needed, you can delete the resource group, Azure Cosmos DB account, and all the related resources. To do so, select the resource group that you used for the Azure Cosmos DB account, select **Delete**, and then confirm the name of the resource group to delete.

Next steps

Monitor an Azure Cosmos DB account

Tutorial: Develop an ASP.NET Core MVC web application with Azure Cosmos DB by using .NET SDK

12/4/2020 • 17 minutes to read • Edit Online

APPLIES TO: SQL API

This tutorial shows you how to use Azure Cosmos DB to store and access data from an ASP.NET MVC application that is hosted on Azure. In this tutorial, you use the .NET SDK V3. The following image shows the web page that you'll build by using the sample in this article:

If you don't have time to complete the tutorial, you can download the complete sample project from GitHub.

This tutorial covers:

- Creating an Azure Cosmos account
- Creating an ASP.NET Core MVC app
- Connecting the app to Azure Cosmos DB
- Performing create, read, update, and delete (CRUD) operations on the data

> **TIP**
> This tutorial assumes that you have prior experience using ASP.NET Core MVC and Azure App Service. If you are new to ASP.NET Core or the prerequisite tools, we recommend you to download the complete sample project from GitHub, add the required NuGet packages, and run it. Once you build the project, you can review this article to gain insight on the code in the context of the project.

Prerequisites

Before following the instructions in this article, make sure that you have the following resources:

- An active Azure account. If you don't have an Azure subscription, create a free account before you begin.

  You can Try Azure Cosmos DB for free without an Azure subscription, free of charge and commitments, or create an Azure Cosmos DB free tier account, with the first 400 RU/s and 5 GB of storage for free. You can also use the Azure Cosmos DB Emulator with a URI of `https://localhost:8081`. For the key to use with the emulator, see Authenticating requests.

- Visual Studio 2019. Download and use the free Visual Studio 2019 Community Edition. Make sure that you enable the **Azure development** workload during the Visual Studio setup.

All the screenshots in this article are from Microsoft Visual Studio Community 2019. If you use a different version, your screens and options may not match entirely. The solution should work if you meet the prerequisites.

Step 1: Create an Azure Cosmos account

Let's start by creating an Azure Cosmos account. If you already have an Azure Cosmos DB SQL API account or if you're using the Azure Cosmos DB Emulator, skip to Step 2: Create a new ASP.NET MVC application.

1. From the Azure portal menu or the **Home page**, select **Create a resource**.
2. On the **New** page, search for and select **Azure Cosmos DB**.
3. On the **Azure Cosmos DB** page, select **Create**.
4. On the **Create Azure Cosmos DB Account** page, enter the basic settings for the new Azure Cosmos account.

SETTING	VALUE	DESCRIPTION
Subscription	Subscription name	Select the Azure subscription that you want to use for this Azure Cosmos account.
Resource Group	Resource group name	Select a resource group, or select **Create new**, then enter a unique name for the new resource group.
Account Name	A unique name	Enter a name to identify your Azure Cosmos account. Because *documents.azure.com* is appended to the name that you provide to create your URI, use a unique name. The name can only contain lowercase letters, numbers, and the hyphen (-) character. It must be between 3-44 characters in length.
API	The type of account to create	Select **Core (SQL)** to create a document database and query by using SQL syntax. The API determines the type of account to create. Azure Cosmos DB provides five APIs: Core (SQL) and MongoDB for document data, Gremlin for graph data, Azure Table, and Cassandra. Currently, you must create a separate account for each API.
Capacity mode	Provisioned throughput or Serverless	Select **Provisioned throughput** to create an account in provisioned throughput mode. Select **Serverless** to create an account in serverless mode.

SETTING	VALUE	DESCRIPTION
Apply Free Tier Discount	Apply or Do not apply	With Azure Cosmos DB free tier, you will get the first 400 RU/s and 5 GB of storage for free in an account. Learn more about free tier.
Location	The region closest to your users	Select a geographic location to host your Azure Cosmos DB account. Use the location that is closest to your users to give them the fastest access to the data.
Account Type	Production or Non-Production	Select **Production** if the account will be used for a production workload. Select **Non-Production** if the account will be used for non-production, e.g. development, testing, QA, or staging. This is an Azure resource tag setting that tunes the Portal experience but does not affect the underlying Azure Cosmos DB account. You can change this value anytime.

NOTE

You can have up to one free tier Azure Cosmos DB account per Azure subscription and must opt-in when creating the account. If you do not see the option to apply the free tier discount, this means another account in the subscription has already been enabled with free tier.

NOTE

The following options are not available if you select **Serverless** as the **Capacity mode**:

- Apply Free Tier Discount
- Geo-redundancy
- Multi-region Writes

5. Select **Review + create**. You can skip the **Network** and **Tags** sections.

6. Review the account settings, and then select **Create**. It takes a few minutes to create the account. Wait for the portal page to display **Your deployment is complete**.

7. Select **Go to resource** to go to the Azure Cosmos DB account page.

Go to the Azure Cosmos DB account page, and select **Keys**. Copy the values to use in the web application you create next.

In the next section, you create a new ASP.NET Core MVC application.

Step 2: Create a new ASP.NET Core MVC application

1. Open Visual Studio and select **Create a new project**.

2. In **Create a new project**, find and select **ASP.NET Core Web Application** for C#. Select **Next** to continue.

3. In **Configure your new project**, name the project *todo* and select **Create**.

4. In **Create a new ASP.NET Core Web Application**, choose **Web Application (Model-View-Controller)**. Select **Create** to continue.

   Visual Studio creates an empty MVC application.

5. Select **Debug** > **Start Debugging** or F5 to run your ASP.NET application locally.

Step 3: Add Azure Cosmos DB NuGet package to the project

Now that we have most of the ASP.NET Core MVC framework code that we need for this solution, let's add the NuGet packages required to connect to Azure Cosmos DB.

1. In **Solution Explorer**, right-click your project and select **Manage NuGet Packages**.

2. In the **NuGet Package Manager**, search for and select **Microsoft.Azure.Cosmos**. Select **Install**.

   Visual Studio downloads and installs the Azure Cosmos DB package and its dependencies.

   You can also use **Package Manager Console** to install the NuGet package. To do so, select **Tools** > **NuGet Package Manager** > **Package Manager Console**. At the prompt, type the following command:

```
Install-Package Microsoft.Azure.Cosmos
```

Step 4: Set up the ASP.NET Core MVC application

Now let's add the models, the views, and the controllers to this MVC application.

Add a model

1. In **Solution Explorer**, right-click the **Models** folder, select **Add** > **Class**.
2. In **Add New Item**, name your new class *Item.cs* and select **Add**.
3. Replace the contents of *Item.cs* class with the following code:

```
namespace todo.Models
{
    using Newtonsoft.Json;

    public class Item
    {
        [JsonProperty(PropertyName = "id")]
        public string Id { get; set; }

        [JsonProperty(PropertyName = "name")]
        public string Name { get; set; }

        [JsonProperty(PropertyName = "description")]
        public string Description { get; set; }

        [JsonProperty(PropertyName = "isComplete")]
        public bool Completed { get; set; }
    }
}
```

Azure Cosmos DB uses JSON to move and store data. You can use the `JsonProperty` attribute to control how JSON serializes and deserializes objects. The `Item` class demonstrates the `JsonProperty` attribute. This code controls the format of the property name that goes into JSON. It also renames the .NET property `Completed`.

Add views

Next, let's add the following views.

- A create item view
- A delete item view
- A view to get an item details
- An edit item view
- A view to list all the items

Create item view

1. In **Solution Explorer**, right-click the **Views** folder and select **Add** > **New Folder**. Name the folder *Item*.
2. Right-click the empty **Item** folder, then select **Add** > **View**.
3. In **Add MVC View**, make the following changes:
   - In **View name**, enter *Create*.
   - In **Template**, select **Create**.
   - In **Model class**, select **Item (todo.Models)**.
   - Select **Use a layout page** and enter *~/Views/Shared/_Layout.cshtml*.

- Select **Add**.

4. Next select **Add** and let Visual Studio create a new template view. Replace the code in the generated file with the following contents:

```
@model todo.Models.Item

@{
    ViewBag.Title = "Create";
    Layout = "~/Views/Shared/_Layout.cshtml";
}

<h2>Create a new To-Do Item</h2>


@using (Html.BeginForm())
{
    @Html.AntiForgeryToken()

    <div class="form-horizontal">
        <hr />
        @Html.ValidationSummary(true, "", new { @class = "text-danger" })
        <div class="form-group">
            @Html.LabelFor(model => model.Name, htmlAttributes: new { @class = "control-label col-md-2"
})
            <div class="col-md-10">
                @Html.EditorFor(model => model.Name, new { htmlAttributes = new { @class = "form-
control" } })
                @Html.ValidationMessageFor(model => model.Name, "", new { @class = "text-danger" })
            </div>
        </div>

        <div class="form-group">
            @Html.LabelFor(model => model.Description, htmlAttributes: new { @class = "control-label
col-md-2" })
            <div class="col-md-10">
                @Html.EditorFor(model => model.Description, new { htmlAttributes = new { @class =
"form-control" } })
                @Html.ValidationMessageFor(model => model.Description, "", new { @class = "text-danger"
})
            </div>
        </div>

        <div class="form-group">
            @Html.LabelFor(model => model.Completed, htmlAttributes: new { @class = "control-label col-
md-2" })
            <div class="col-md-10">
                <div class="checkbox">
                    @Html.EditorFor(model => model.Completed)
                    @Html.ValidationMessageFor(model => model.Completed, "", new { @class = "text-
danger" })
                </div>
            </div>
        </div>

        <div class="form-group">
            <div class="col-md-offset-2 col-md-10">
                <input type="submit" value="Create" class="btn btn-default" />
            </div>
        </div>
    </div>
}

<div>
    @Html.ActionLink("Back to List", "Index")
</div>
<script src="~/bundles/jqueryval"></script>
```

Delete item view

1. From the **Solution Explorer**, right-click the **Item** folder again, select **Add** > **View**.

2. In **Add MVC View**, make the following changes:
   - In the **View name** box, type *Delete*.
   - In the **Template** box, select **Delete**.
   - In the **Model class** box, select **Item (todo.Models)**.
   - Select **Use a layout page** and enter *~/Views/Shared/_Layout.cshtml*.
   - Select **Add**.
3. Next select **Add** and let Visual Studio create a new template view. Replace the code in the generated file with the following contents:

```
@model todo.Models.Item

@{
    ViewBag.Title = "Delete";
    Layout = "~/Views/Shared/_Layout.cshtml";
}

<h2>Delete a To-Do Item</h2>

<h3>Are you sure you want to delete this?</h3>
<div>
    <hr />
    <dl class="dl-horizontal">
        <dt>
            @Html.DisplayNameFor(model => model.Name)
        </dt>

        <dd>
            @Html.DisplayFor(model => model.Name)
        </dd>

        <dt>
            @Html.DisplayNameFor(model => model.Description)
        </dt>

        <dd>
            @Html.DisplayFor(model => model.Description)
        </dd>

        <dt>
            @Html.DisplayNameFor(model => model.Completed)
        </dt>

        <dd>
            @Html.DisplayFor(model => model.Completed)
        </dd>
    </dl>

    @using (Html.BeginForm()) {
        @Html.AntiForgeryToken()

        <div class="form-actions no-color">
            <input type="submit" value="Delete" class="btn btn-default" /> |
            @Html.ActionLink("Back to List", "Index")
        </div>
    }
</div>
```

Add a view to get an item details

1. In **Solution Explorer**, right-click the **Item** folder again, select **Add** > **View**.
2. In **Add MVC View**, provide the following values:

- In **View name**, enter *Details*.
- In **Template**, select **Details**.
- In **Model class**, select **Item (todo.Models)**.
- Select **Use a layout page** and enter *~/Views/Shared/_Layout.cshtml*.

3. Next select **Add** and let Visual Studio create a new template view. Replace the code in the generated file with the following contents:

```
@model todo.Models.Item

<h2>View To-Do Item Details</h2>

<div>
    <h4>Item</h4>
    <hr />
    <dl class="dl-horizontal">
        <dt>
            @Html.DisplayNameFor(model => model.Name)
        </dt>

        <dd>
            @Html.DisplayFor(model => model.Name)
        </dd>

        <dt>
            @Html.DisplayNameFor(model => model.Description)
        </dt>

        <dd>
            @Html.DisplayFor(model => model.Description)
        </dd>

        <dt>
            @Html.DisplayNameFor(model => model.Completed)
        </dt>

        <dd>
            @Html.DisplayFor(model => model.Completed)
        </dd>

    </dl>
</div>
<p>
    @Html.ActionLink("Edit", "Edit", new { id = Model.Id }) |
    @Html.ActionLink("Back to List", "Index")
</p>
```

Add an edit item view

1. From the **Solution Explorer**, right-click the **Item** folder again, select **Add** > **View**.

2. In **Add MVC View**, make the following changes:

   - In the **View name** box, type *Edit*.
   - In the **Template** box, select **Edit**.
   - In the **Model class** box, select **Item (todo.Models)**.
   - Select **Use a layout page** and enter *~/Views/Shared/_Layout.cshtml*.
   - Select **Add**.

3. Next select **Add** and let Visual Studio create a new template view. Replace the code in the generated file with the following contents:

```
@model todo.Models.Item

<h2>Edit a To-Do Item</h2>

@using (Html.BeginForm())
{
    @Html.AntiForgeryToken()

    <div class="form-horizontal">
        <h4>Item</h4>
        <hr />
        @Html.ValidationSummary(true, "", new { @class = "text-danger" })
        @Html.HiddenFor(model => model.Id)

        <div class="form-group">
            @Html.LabelFor(model => model.Name, htmlAttributes: new { @class = "control-label col-md-2"
})
            <div class="col-md-10">
                @Html.EditorFor(model => model.Name, new { htmlAttributes = new { @class = "form-
control" } })
                @Html.ValidationMessageFor(model => model.Name, "", new { @class = "text-danger" })
            </div>
        </div>

        <div class="form-group">
            @Html.LabelFor(model => model.Description, htmlAttributes: new { @class = "control-label
col-md-2" })
            <div class="col-md-10">
                @Html.EditorFor(model => model.Description, new { htmlAttributes = new { @class =
"form-control" } })
                @Html.ValidationMessageFor(model => model.Description, "", new { @class = "text-danger"
})
            </div>
        </div>

        <div class="form-group">
            @Html.LabelFor(model => model.Completed, htmlAttributes: new { @class = "control-label col-
md-2" })
            <div class="col-md-10">
                <div class="checkbox">
                    @Html.EditorFor(model => model.Completed)
                    @Html.ValidationMessageFor(model => model.Completed, "", new { @class = "text-
danger" })
                </div>
            </div>
        </div>

        <div class="form-group">
            <div class="col-md-offset-2 col-md-10">
                <input type="submit" value="Save" class="btn btn-default" />
            </div>
        </div>
    </div>
}

<div>
    @Html.ActionLink("Back to List", "Index")
</div>

<script src="~/bundles/jqueryval"></script>
```

Add a view to list all the items

And finally, add a view to get all the items with the following steps:

1. From the **Solution Explorer**, right-click the **Item** folder again, select **Add** > **View**.

2. In **Add MVC View**, make the following changes:
   - In the **View name** box, type *Index*.
   - In the **Template** box, select **List**.
   - In the **Model class** box, select **Item (todo.Models)**.
   - Select **Use a layout page** and enter *~/Views/Shared/_Layout.cshtml*.
   - Select **Add**.
3. Next select **Add** and let Visual Studio create a new template view. Replace the code in the generated file with the following contents:

```
@model IEnumerable<todo.Models.Item>

@{
    ViewBag.Title = "List of To-Do Items";
    Layout = "~/Views/Shared/_Layout.cshtml";
}

<h2>List of To-Do Items</h2>

<table class="table">
    <tr>
        <th>
            @Html.DisplayNameFor(model => model.Name)
        </th>
        <th>
            @Html.DisplayNameFor(model => model.Description)
        </th>
        <th>
            @Html.DisplayNameFor(model => model.Completed)
        </th>
        <th></th>
    </tr>

@foreach (var item in Model) {
    <tr>
        <td>
            @Html.DisplayFor(modelItem => item.Name)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.Description)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.Completed)
        </td>
        <td>
            @Html.ActionLink("Edit", "Edit", new { id=item.Id }) |
            @Html.ActionLink("Details", "Details", new { id=item.Id }) |
            @Html.ActionLink("Delete", "Delete", new { id=item.Id })
        </td>
    </tr>
}

</table>

<p>
    @Html.ActionLink("Create New", "Create")
</p>
```

Once you complete these steps, close all the *cshtml* documents in Visual Studio.

Declare and initialize services

First, we'll add a class that contains the logic to connect to and use Azure Cosmos DB. For this tutorial, we'll

encapsulate this logic into a class called `CosmosDbService` and an interface called `ICosmosDbService`. This service does the CRUD operations. It also does read feed operations such as listing incomplete items, creating, editing, and deleting the items.

1. In **Solution Explorer**, right-click your project and select **Add** > **New Folder**. Name the folder *Services*.
2. Right-click the **Services** folder, select **Add** > **Class**. Name the new class *CosmosDbService* and select **Add**.
3. Replace the contents of *CosmosDbService.cs* with the following code:

```
namespace todo
{
    using System.Collections.Generic;
    using System.Linq;
    using System.Threading.Tasks;
    using todo.Models;
    using Microsoft.Azure.Cosmos;
    using Microsoft.Azure.Cosmos.Fluent;
    using Microsoft.Extensions.Configuration;

    public class CosmosDbService : ICosmosDbService
    {
        private Container _container;

        public CosmosDbService(
            CosmosClient dbClient,
            string databaseName,
            string containerName)
        {
            this._container = dbClient.GetContainer(databaseName, containerName);
        }

        public async Task AddItemAsync(Item item)
        {
            await this._container.CreateItemAsync<Item>(item, new PartitionKey(item.Id));
        }

        public async Task DeleteItemAsync(string id)
        {
            await this._container.DeleteItemAsync<Item>(id, new PartitionKey(id));
        }

        public async Task<Item> GetItemAsync(string id)
        {
            try
            {
                ItemResponse<Item> response = await this._container.ReadItemAsync<Item>(id, new
PartitionKey(id));
                return response.Resource;
            }
            catch(CosmosException ex) when (ex.StatusCode == System.Net.HttpStatusCode.NotFound)
            {
                return null;
            }

        }

        public async Task<IEnumerable<Item>> GetItemsAsync(string queryString)
        {
            var query = this._container.GetItemQueryIterator<Item>(new QueryDefinition(queryString));
            List<Item> results = new List<Item>();
            while (query.HasMoreResults)
            {
                var response = await query.ReadNextAsync();

                results.AddRange(response.ToList());
            }
```

```
            return results;
        }

        public async Task UpdateItemAsync(string id, Item item)
        {
            await this._container.UpsertItemAsync<Item>(item, new PartitionKey(id));
        }
    }
}
```

4. Right-click the **Services** folder, select **Add** > **Class**. Name the new class *ICosmosDbService* and select **Add**.

5. Add the following code to *ICosmosDbService* class:

```
namespace todo
{
    using System.Collections.Generic;
    using System.Threading.Tasks;
    using todo.Models;

    public interface ICosmosDbService
    {
        Task<IEnumerable<Item>> GetItemsAsync(string query);
        Task<Item> GetItemAsync(string id);
        Task AddItemAsync(Item item);
        Task UpdateItemAsync(string id, Item item);
        Task DeleteItemAsync(string id);
    }
}
```

6. Open the *Startup.cs* file in your solution and add the following method **InitializeCosmosClientInstanceAsync**, which reads the configuration and initializes the client.

```
/// <summary>
/// Creates a Cosmos DB database and a container with the specified partition key.
/// </summary>
/// <returns></returns>
private static async Task<CosmosDbService> InitializeCosmosClientInstanceAsync(IConfigurationSection
configurationSection)
{
    string databaseName = configurationSection.GetSection("DatabaseName").Value;
    string containerName = configurationSection.GetSection("ContainerName").Value;
    string account = configurationSection.GetSection("Account").Value;
    string key = configurationSection.GetSection("Key").Value;
    Microsoft.Azure.Cosmos.CosmosClient client = new Microsoft.Azure.Cosmos.CosmosClient(account, key);
    CosmosDbService cosmosDbService = new CosmosDbService(client, databaseName, containerName);
    Microsoft.Azure.Cosmos.DatabaseResponse database = await
client.CreateDatabaseIfNotExistsAsync(databaseName);
    await database.Database.CreateContainerIfNotExistsAsync(containerName, "/id");

    return cosmosDbService;
}
```

7. On that same file, replace the `ConfigureServices` method with:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddControllersWithViews();
    services.AddSingleton<ICosmosDbService>
(InitializeCosmosClientInstanceAsync(Configuration.GetSection("CosmosDb")).GetAwaiter().GetResult());
}
```

The code in this step initializes the client based on the configuration as a singleton instance to be injected through Dependency injection in ASP.NET Core.

And make sure to change the default MVC Controller to `Item` by editing the routes in the `Configure` method of the same file:

```
 app.UseEndpoints(endpoints =>
       {
             endpoints.MapControllerRoute(
                name: "default",
                pattern: "{controller=Item}/{action=Index}/{id?}");
       });
```

8. Define the configuration in the project's *appsettings.json* file as shown in the following snippet:

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Warning"
    }
  },
  "AllowedHosts": "*",
  "CosmosDb": {
    "Account": "<Endpoint URI of your Azure Cosmos account>",
    "Key": "<PRIMARY KEY of your Azure Cosmos account>",
    "DatabaseName": "Tasks",
    "ContainerName": "Item"
  }
}
```

Add a controller

1. In **Solution Explorer**, right-click the **Controllers** folder, select **Add** > **Controller**.

2. In **Add Scaffold**, select **MVC Controller - Empty** and select **Add**.

3. Name your new controller *ItemController*.

4. Replace the contents of *ItemController.cs* with the following code:

```
namespace todo.Controllers
{
    using System;
    using System.Threading.Tasks;
    using Microsoft.AspNetCore.Mvc;
    using todo.Models;

    public class ItemController : Controller
    {
        private readonly ICosmosDbService _cosmosDbService;
        public ItemController(ICosmosDbService cosmosDbService)
        {
            _cosmosDbService = cosmosDbService;
        }

        [ActionName("Index")]
        public async Task<IActionResult> Index()
        {
            return View(await _cosmosDbService.GetItemsAsync("SELECT * FROM c"));
        }

        [ActionName("Create")]
        public IActionResult Create()
        {
            return View();
        }

        [HttpPost]
        [ActionName("Create")]
        [ValidateAntiForgeryToken]
        public async Task<ActionResult> CreateAsync([Bind("Id,Name,Description,Completed")] Item item)
        {
            if (ModelState.IsValid)
            {
                item.Id = Guid.NewGuid().ToString();
```

```
            await _cosmosDbService.AddItemAsync(item);
            return RedirectToAction("Index");
        }

        return View(item);
    }

    [HttpPost]
    [ActionName("Edit")]
    [ValidateAntiForgeryToken]
    public async Task<ActionResult> EditAsync([Bind("Id,Name,Description,Completed")] Item item)
    {
        if (ModelState.IsValid)
        {
            await _cosmosDbService.UpdateItemAsync(item.Id, item);
            return RedirectToAction("Index");
        }

        return View(item);
    }

    [ActionName("Edit")]
    public async Task<ActionResult> EditAsync(string id)
    {
        if (id == null)
        {
            return BadRequest();
        }

        Item item = await _cosmosDbService.GetItemAsync(id);
        if (item == null)
        {
            return NotFound();
        }

        return View(item);
    }

    [ActionName("Delete")]
    public async Task<ActionResult> DeleteAsync(string id)
    {
        if (id == null)
        {
            return BadRequest();
        }

        Item item = await _cosmosDbService.GetItemAsync(id);
        if (item == null)
        {
            return NotFound();
        }

        return View(item);
    }

    [HttpPost]
    [ActionName("Delete")]
    [ValidateAntiForgeryToken]
    public async Task<ActionResult> DeleteConfirmedAsync([Bind("Id")] string id)
    {
        await _cosmosDbService.DeleteItemAsync(id);
        return RedirectToAction("Index");
    }

    [ActionName("Details")]
    public async Task<ActionResult> DetailsAsync(string id)
    {
        return View(await _cosmosDbService.GetItemAsync(id));
    }
```

```
    }
}
```

The **ValidateAntiForgeryToken** attribute is used here to help protect this application against cross-site request forgery attacks. Your views should work with this anti-forgery token as well. For more information and examples, see Preventing Cross-Site Request Forgery (CSRF) Attacks in ASP.NET MVC Application. The source code provided on GitHub has the full implementation in place.

We also use the **Bind** attribute on the method parameter to help protect against over-posting attacks. For more information, see Tutorial: Implement CRUD Functionality with the Entity Framework in ASP.NET MVC.

Step 5: Run the application locally

To test the application on your local computer, use the following steps:

1. Press F5 in Visual Studio to build the application in debug mode. It should build the application and launch a browser with the empty grid page we saw before:

   If the application instead opens to the home page, append `/Item` to the url.

2. Select the **Create New** link and add values to the **Name** and **Description** fields. Leave the **Completed** check box unselected. If you select it, the app adds the new item in a completed state. The item no longer appears on the initial list.

3. Select **Create**. The app sends you back to the **Index** view, and your item appears in the list. You can add a few more items to your **To-Do** list.

4. Select **Edit** next to an **Item** on the list. The app opens the **Edit** view where you can update any property of your object, including the **Completed** flag. If you select **Completed** and select **Save**, the app displays the **Item** as completed in the list.

5. Verify the state of the data in the Azure Cosmos DB service using Cosmos Explorer or the Azure Cosmos DB Emulator's Data Explorer.

6. Once you've tested the app, select Ctrl+F5 to stop debugging the app. You're ready to deploy!

Step 6: Deploy the application

Now that you have the complete application working correctly with Azure Cosmos DB we're going to deploy this web app to Azure App Service.

1. To publish this application, right-click the project in **Solution Explorer** and select **Publish**.

2. In **Pick a publish target**, select **App Service**.

3. To use an existing App Service profile, choose **Select Existing**, then select **Publish**.

4. In **App Service**, select a **Subscription**. Use the **View** filter to sort by resource group or resource type.

5. Find your profile, and then select **OK**. Next search the required Azure App Service and select **OK**.

Another option is to create a new profile:

1. As in the previous procedure, right-click the project in **Solution Explorer** and select **Publish**.
2. In **Pick a publish target**, select **App Service**.
3. In **Pick a publish target**, select **Create New** and select **Publish**.
4. In **App Service**, enter your Web App name and the appropriate subscription, resource group, and hosting plan, then select **Create**.

In a few seconds, Visual Studio publishes your web application and launches a browser where you can see your project running in Azure!

Next steps

In this tutorial, you've learned how to build an ASP.NET Core MVC web application. Your application can access data stored in Azure Cosmos DB. You can now continue with these resources:

- Partitioning in Azure Cosmos DB
- Getting started with SQL queries
- How to model and partition data on Azure Cosmos DB using a real-world example

Tutorial: Build a Java web application using Azure Cosmos DB and the SQL API

12/4/2020 • 23 minutes to read • Edit Online

APPLIES TO: SQL API

This Java web application tutorial shows you how to use the Microsoft Azure Cosmos DB service to store and access data from a Java application hosted on Azure App Service Web Apps. In this article, you will learn:

- How to build a basic JavaServer Pages (JSP) application in Eclipse.
- How to work with the Azure Cosmos DB service using the Azure Cosmos DB Java SDK.

This Java application tutorial shows you how to create a web-based task-management application that enables you to create, retrieve, and mark tasks as complete, as shown in the following image. Each of the tasks in the ToDo list is stored as JSON documents in Azure Cosmos DB.

> **TIP**
> This application development tutorial assumes that you have prior experience using Java. If you are new to Java or the prerequisite tools, we recommend downloading the complete todo project from GitHub and building it using the instructions at the end of this article. Once you have it built, you can review the article to gain insight on the code in the context of the project.

Prerequisites for this Java web application tutorial

Before you begin this application development tutorial, you must have the following:

- If you don't have an Azure subscription, create a free account before you begin.

  You can Try Azure Cosmos DB for free without an Azure subscription, free of charge and commitments, or create an Azure Cosmos DB free tier account, with the first 400 RU/s and 5 GB of storage for free. You can also use the Azure Cosmos DB Emulator with a URI of `https://localhost:8081`. For the key to use with the emulator, see Authenticating requests.

- Java Development Kit (JDK) 7+.
- Eclipse IDE for Java EE Developers.
- An Azure Web Site with a Java runtime environment (e.g. Tomcat or Jetty) enabled.

If you're installing these tools for the first time, coreservlets.com provides a walk-through of the installation process in the quickstart section of their Tutorial: Installing TomCat7 and Using it with Eclipse article.

Create an Azure Cosmos DB account

Let's start by creating an Azure Cosmos DB account. If you already have an account or if you are using the Azure Cosmos DB Emulator for this tutorial, you can skip to Step 2: Create the Java JSP application.

1. From the Azure portal menu or the **Home page**, select **Create a resource**.
2. On the **New** page, search for and select **Azure Cosmos DB**.
3. On the **Azure Cosmos DB** page, select **Create**.
4. On the **Create Azure Cosmos DB Account** page, enter the basic settings for the new Azure Cosmos account.

SETTING	VALUE	DESCRIPTION
Subscription	Subscription name	Select the Azure subscription that you want to use for this Azure Cosmos account.
Resource Group	Resource group name	Select a resource group, or select **Create new**, then enter a unique name for the new resource group.
Account Name	A unique name	Enter a name to identify your Azure Cosmos account. Because *documents.azure.com* is appended to the name that you provide to create your URI, use a unique name. The name can only contain lowercase letters, numbers, and the hyphen (-) character. It must be between 3-44 characters in length.

SETTING	VALUE	DESCRIPTION
API	The type of account to create	Select **Core (SQL)** to create a document database and query by using SQL syntax. The API determines the type of account to create. Azure Cosmos DB provides five APIs: Core (SQL) and MongoDB for document data, Gremlin for graph data, Azure Table, and Cassandra. Currently, you must create a separate account for each API.
Capacity mode	Provisioned throughput or Serverless	Select **Provisioned throughput** to create an account in provisioned throughput mode. Select **Serverless** to create an account in serverless mode.
Apply Free Tier Discount	Apply or Do not apply	With Azure Cosmos DB free tier, you will get the first 400 RU/s and 5 GB of storage for free in an account. Learn more about free tier.
Location	The region closest to your users	Select a geographic location to host your Azure Cosmos DB account. Use the location that is closest to your users to give them the fastest access to the data.
Account Type	Production or Non-Production	Select **Production** if the account will be used for a production workload. Select **Non-Production** if the account will be used for non-production, e.g. development, testing, QA, or staging. This is an Azure resource tag setting that tunes the Portal experience but does not affect the underlying Azure Cosmos DB account. You can change this value anytime.

NOTE

You can have up to one free tier Azure Cosmos DB account per Azure subscription and must opt-in when creating the account. If you do not see the option to apply the free tier discount, this means another account in the subscription has already been enabled with free tier.

NOTE

The following options are not available if you select **Serverless** as the **Capacity mode**:

- Apply Free Tier Discount
- Geo-redundancy
- Multi-region Writes

5. Select **Review + create**. You can skip the **Network** and **Tags** sections.

6. Review the account settings, and then select **Create**. It takes a few minutes to create the account. Wait for the portal page to display **Your deployment is complete**.

7. Select **Go to resource** to go to the Azure Cosmos DB account page.

Go to the Azure Cosmos DB account page, and select **Keys**. Copy the values to use in the web application you create next.

Create the Java JSP application

To create the JSP application:

1. First, we'll start off by creating a Java project. Start Eclipse, then click **File**, click **New**, and then click **Dynamic Web Project**. If you don't see **Dynamic Web Project** listed as an available project, do the following: click **File**, click **New**, click **Project**..., expand **Web**, click **Dynamic Web Project**, and click **Next**.

2. Enter a project name in the **Project name** box, and in the **Target Runtime** drop-down menu, optionally select a value (e.g. Apache Tomcat v7.0), and then click **Finish**. Selecting a target runtime enables you to run your project locally through Eclipse.

3. In Eclipse, in the Project Explorer view, expand your project. Right-click **WebContent**, click **New**, and then click **JSP File**.

4. In the **New JSP File** dialog box, name the file **index.jsp**. Keep the parent folder as **WebContent**, as shown in the following illustration, and then click **Next**.

5. In the **Select JSP Template** dialog box, for the purpose of this tutorial select **New JSP File (html)**, and then click **Finish**.

6. When the *index.jsp* file opens in Eclipse, add text to display **Hello World!** within the existing `<body>` element. The updated `<body>` content should look like the following code:

```
<body>
  <% out.println("Hello World!"); %>
</body>
```

7. Save the *index.jsp* file.

8. If you set a target runtime in step 2, you can click **Project** and then **Run** to run your JSP application locally:

Install the SQL Java SDK

The easiest way to pull in the SQL Java SDK and its dependencies is through Apache Maven. To do this, you need to convert your project to a Maven project by using the following steps:

1. Right-click your project in the Project Explorer, click **Configure**, click **Convert to Maven Project**.

2. In the **Create new POM** window, accept the defaults, and click **Finish**.

3. In **Project Explorer**, open the pom.xml file.

4. On the **Dependencies** tab, in the **Dependencies** pane, click **Add**.

5. In the **Select Dependency** window, do the following:

   - In the **Group Id** box, enter `com.azure`.

- In the **Artifact Id** box, enter `azure-cosmos`.
- In the **Version** box, enter `4.0.1-beta.1`.

Or, you can add the dependency XML for Group ID and Artifact ID directly to the *pom.xml* file:

```
<dependency>
   <groupId>com.azure</groupId>
   <artifactId>azure-cosmos</artifactId>
   <version>4.0.1-beta.1</version>
</dependency>
```

6. Click **OK** and Maven will install the SQL Java SDK or save the pom.xml file.

Use the Azure Cosmos DB service in your Java application

Now let's add the models, the views, and the controllers to your web application.

Add a model

First, let's define a model within a new file *TodoItem.java*. The `TodoItem` class defines the schema of an item along with the getter and setter methods:

```
package com.microsoft.azure.documentdb.sample.model;

//@Data
//@Builder
public class TodoItem {
    private String entityType;
    private String category;
    private boolean complete;
    private String id;
    private String name;

    public String getCategory() {
        return category;
    }

    public void setCategory(String category) {
        this.category = category;
    }

    public String getEntityType() {
        return entityType;
    }

    public void setEntityType(String entityType) {
        this.entityType = entityType;
    }

    public boolean isComplete() {
        return complete;
    }

    public void setComplete(boolean complete) {
        this.complete = complete;
    }

    public String getId() {
        return id;
    }

    public void setId(String id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }


}
```

Add the Data Access Object(DAO) classes

Create a Data Access Object (DAO) to abstract persisting the ToDo items to Azure Cosmos DB. In order to save ToDo items to a collection, the client needs to know which database and collection to persist to (as referenced by self-links). In general, it is best to cache the database and collection when possible to avoid additional round-trips to the database.

1. To invoke the Azure Cosmos DB service, you must instantiate a new `cosmosClient` object. In general, it is best to reuse the `cosmosClient` object rather than constructing a new client for each subsequent request. You can reuse the client by defining it within the `cosmosClientFactory` class. Update the HOST and MASTER_KEY values that you saved in step 1. Replace the HOST variable with with your URI and replace the

MASTER_KEY with your PRIMARY KEY. Use the following code to create the `CosmosClientFactory` class within the *CosmosClientFactory.java* file:

```
package com.microsoft.azure.documentdb.sample.dao;

import com.azure.cosmos.ConnectionPolicy;
import com.azure.cosmos.ConsistencyLevel;
import com.azure.cosmos.CosmosClient;
import com.azure.cosmos.CosmosClientBuilder;

public class CosmosClientFactory {
    private static final String HOST = "https://docdb-java-sample.documents.azure.com:443/";
    private static final String MASTER_KEY = "[YOUR_KEY_HERE]";

    private static CosmosClient cosmosClient = new CosmosClientBuilder()
            .setEndpoint(HOST)
            .setKey(MASTER_KEY)
            .setConnectionPolicy(ConnectionPolicy.getDefaultPolicy())
            .setConsistencyLevel(ConsistencyLevel.EVENTUAL)
            .buildClient();

    public static CosmosClient getCosmosClient() {
        return cosmosClient;
    }

}
```

2. Create a new *TodoDao.java* file and add the `TodoDao` class to create, update, read, and delete the todo items:

```
package com.microsoft.azure.documentdb.sample.dao;

import java.util.List;

import com.microsoft.azure.documentdb.sample.model.TodoItem;

public interface TodoDao {
    /**
     * @return A list of TodoItems
     */
    public List<TodoItem> readTodoItems();

    /**
     * @param todoItem
     * @return whether the todoItem was persisted.
     */
    public TodoItem createTodoItem(TodoItem todoItem);

    /**
     * @param id
     * @return the TodoItem
     */
    public TodoItem readTodoItem(String id);

    /**
     * @param id
     * @return the TodoItem
     */
    public TodoItem updateTodoItem(String id, boolean isComplete);

    /**
     *
     * @param id
     * @return whether the delete was successful.
     */
    public boolean deleteTodoItem(String id);
}
```

3. Create a new *MockDao.java* file and add the `MockDao` class, this class implements the `TodoDao` class to perform CRUD operations on the items:

```
package com.microsoft.azure.documentdb.sample.dao;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

import lombok.NonNull;

import org.apache.commons.lang3.StringUtils;

import com.microsoft.azure.documentdb.sample.model.TodoItem;

public class MockDao implements TodoDao {
    private final Map<String, TodoItem> todoItemMap;

    public MockDao() {
        todoItemMap = new HashMap<String, TodoItem>();
    }

    @Override
    public TodoItem createTodoItem(@NonNull TodoItem todoItem) {
        if (StringUtils.isEmpty(todoItem.getId())) {
            todoItem.setId(generateId());
        }
        todoItemMap.put(todoItem.getId(), todoItem);
        return todoItem;
    }

    @Override
    public TodoItem readTodoItem(@NonNull String id) {
        return todoItemMap.get(id);
    }

    @Override
    public List<TodoItem> readTodoItems() {
        return new ArrayList<TodoItem>(todoItemMap.values());
    }

    @Override
    public TodoItem updateTodoItem(String id, boolean isComplete) {
        todoItemMap.get(id).setComplete(isComplete);
        return todoItemMap.get(id);
    }

    @Override
    public boolean deleteTodoItem(@NonNull String id) {
        todoItemMap.remove(id);
        return true;
    }

    private String generateId() {
        return new Integer(todoItemMap.size()).toString();
    }
}
```

4. Create a new *DocDbDao.java* file and add the `DocDbDao` class. This class defines code to persist the TodoItems into the container, retrieves your database and collection, if it exists, or create a new one if it doesn't exist. This example uses Gson to serialize and de-serialize the TodoItem Plain Old Java Objects (POJOs) to JSON documents. In order to save ToDo items to a collection, the client needs to know which database and collection to persist to (as referenced by self-links). This class also defines helper function to retrieve the documents by another attribute (e.g. "ID") rather than self-link. You can use the helper method to retrieve a TodoItem JSON document by ID and then deserialize it to a POJO.

You can also use the `cosmosClient` client object to get a collection or list of TodoItems using a SQL query. Finally, you define the delete method to delete a TodoItem from your list. The following code shows the contents of the `DocDbDao` class:

```
package com.microsoft.azure.documentdb.sample.dao;

import java.util.ArrayList;
import java.util.List;

import com.google.gson.Gson;
import com.azure.cosmos.ConnectionPolicy;
import com.azure.cosmos.ConsistencyLevel;
import com.azure.cosmos.CosmosClient;
import com.azure.cosmos.CosmosClientBuilder;
import com.azure.cosmos.CosmosClientException;
import com.azure.cosmos.CosmosContainer;
import com.azure.cosmos.CosmosDatabase;
import com.azure.cosmos.implementation.Utils;
import com.azure.cosmos.models.CosmosContainerProperties;
import com.azure.cosmos.models.CosmosItemRequestOptions;
import com.azure.cosmos.models.FeedOptions;
import com.azure.cosmos.models.FeedResponse;
import com.azure.cosmos.models.PartitionKey;
import com.fasterxml.jackson.core.JsonProcessingException;
import com.fasterxml.jackson.databind.JsonNode;
import com.fasterxml.jackson.databind.ObjectMapper;
import com.fasterxml.jackson.databind.node.ObjectNode;
import com.microsoft.azure.documentdb.sample.model.TodoItem;

public class DocDbDao implements TodoDao {
    // The name of our database.
    private static final String DATABASE_ID = "TestDB";

    // The name of our collection.
    private static final String CONTAINER_ID = "TestCollection";

    // We'll use Gson for POJO <=> JSON serialization for this example.
    private static Gson gson = new Gson();

    // The Cosmos DB Client
    private static CosmosClient cosmosClient = CosmosClientFactory
            .getCosmosClient();

    // The Cosmos DB database
    private static CosmosDatabase cosmosDatabase = null;

    // The Cosmos DB container
    private static CosmosContainer cosmosContainer = null;

    // For POJO/JsonNode interconversion
    private static final ObjectMapper OBJECT_MAPPER = Utils.getSimpleObjectMapper();

    @Override
    public TodoItem createTodoItem(TodoItem todoItem) {
        // Serialize the TodoItem as a JSON Document.

        JsonNode todoItemJson = OBJECT_MAPPER.valueToTree(todoItem);

        ((ObjectNode)todoItemJson).put("entityType", "todoItem");

        try {
            // Persist the document using the DocumentClient.
            todoItemJson =
                    getContainerCreateResourcesIfNotExist()
                    .createItem(todoItemJson)
                    .getResource();
        } catch (CosmosClientException e) {
```

```
            System.out.println("Error creating TODO item.\n");
            e.printStackTrace();
            return null;
        }


        try {

            return OBJECT_MAPPER.treeToValue(todoItemJson, TodoItem.class);
            //return todoItem;
        } catch (Exception e) {
            System.out.println("Error deserializing created TODO item.\n");
            e.printStackTrace();

            return null;
        }

    }

    @Override
    public TodoItem readTodoItem(String id) {
        // Retrieve the document by id using our helper method.
        JsonNode todoItemJson = getDocumentById(id);

        if (todoItemJson != null) {
            // De-serialize the document in to a TodoItem.
            try {
            return OBJECT_MAPPER.treeToValue(todoItemJson, TodoItem.class);
            } catch (JsonProcessingException e) {
                System.out.println("Error deserializing read TODO item.\n");
                e.printStackTrace();

                return null;
            }
        } else {
            return null;
        }
    }

    @Override
    public List<TodoItem> readTodoItems() {

        List<TodoItem> todoItems = new ArrayList<TodoItem>();

        String sql = "SELECT * FROM root r WHERE r.entityType = 'todoItem'";
        int maxItemCount = 1000;
        int maxDegreeOfParallelism = 1000;
        int maxBufferedItemCount = 100;

        FeedOptions options = new FeedOptions();
        options.setMaxItemCount(maxItemCount);
        options.setMaxBufferedItemCount(maxBufferedItemCount);
        options.setMaxDegreeOfParallelism(maxDegreeOfParallelism);
        options.setRequestContinuation(null);
        options.setPopulateQueryMetrics(false);

        int error_count = 0;
        int error_limit = 10;

        do {

            for (FeedResponse<JsonNode> pageResponse:
                getContainerCreateResourcesIfNotExist()
                    .queryItems(sql, options, JsonNode.class)
                    .iterableByPage()) {

                options.setRequestContinuation(pageResponse.getContinuationToken());

                for (JsonNode item : pageResponse.getElements()) {
```

```
                    try {
                        todoItems.add(OBJECT_MAPPER.treeToValue(item, TodoItem.class));
                    } catch (JsonProcessingException e) {
                        if (error_count < error_limit) {
                            error_count++;
                            if (error_count >= error_limit) {
                                System.out.println("\n...reached max error count.\n");
                            } else {
                                System.out.println("Error deserializing TODO item JsonNode. " +
                                        "This item will not be returned.");
                                e.printStackTrace();
                            }
                        }
                    }

                }
            }

        } while(options.getRequestContinuation() != null);

        return todoItems;
    }

    @Override
    public TodoItem updateTodoItem(String id, boolean isComplete) {
        // Retrieve the document from the database
        JsonNode todoItemJson = getDocumentById(id);

        // You can update the document as a JSON document directly.
        // For more complex operations - you could de-serialize the document in
        // to a POJO, update the POJO, and then re-serialize the POJO back in to
        // a document.
        ((ObjectNode)todoItemJson).put("complete", isComplete);

        try {
            // Persist/replace the updated document.
            todoItemJson =
                    getContainerCreateResourcesIfNotExist()
                    .replaceItem(todoItemJson, id, new PartitionKey(id), new
CosmosItemRequestOptions())
                    .getResource();
        } catch (CosmosClientException e) {
            System.out.println("Error updating TODO item.\n");
            e.printStackTrace();
            return null;
        }

        // De-serialize the document in to a TodoItem.
        try {
            return OBJECT_MAPPER.treeToValue(todoItemJson, TodoItem.class);
        } catch (JsonProcessingException e) {
            System.out.println("Error deserializing updated item.\n");
            e.printStackTrace();

            return null;
        }
    }

    @Override
    public boolean deleteTodoItem(String id) {
        // CosmosDB refers to documents by self link rather than id.

        // Query for the document to retrieve the self link.
        JsonNode todoItemJson = getDocumentById(id);

        try {
            // Delete the document by self link.
            getContainerCreateResourcesIfNotExist()
```

```
            getContainerCreateResourcesIfNotExist()
                .deleteItem(id, new PartitionKey(id), new CosmosItemRequestOptions());
        } catch (CosmosClientException e) {
            System.out.println("Error deleting TODO item.\n");
            e.printStackTrace();
            return false;
        }

        return true;
    }

    /*

    private CosmosDatabase getTodoDatabase() {
        if (databaseCache == null) {
            // Get the database if it exists
            List<CosmosDatabase> databaseList = cosmosClient
                    .queryDatabases(
                            "SELECT * FROM root r WHERE r.id='" + DATABASE_ID
                                    + "'", null).getQueryIterable().toList();

            if (databaseList.size() > 0) {
                // Cache the database object so we won't have to query for it
                // later to retrieve the selfLink.
                databaseCache = databaseList.get(0);
            } else {
                // Create the database if it doesn't exist.
                try {
                    CosmosDatabase databaseDefinition = new CosmosDatabase();
                    databaseDefinition.setId(DATABASE_ID);

                    databaseCache = cosmosClient.createDatabase(
                            databaseDefinition, null).getResource();
                } catch (CosmosClientException e) {
                    // TODO: Something has gone terribly wrong - the app wasn't
                    // able to query or create the collection.
                    // Verify your connection, endpoint, and key.
                    e.printStackTrace();
                }
            }
        }

        return databaseCache;
    }

    */

    private CosmosContainer getContainerCreateResourcesIfNotExist() {

        try {

            if (cosmosDatabase == null) {
                cosmosDatabase = cosmosClient.createDatabaseIfNotExists(DATABASE_ID).getDatabase();
            }

        } catch(CosmosClientException e) {
            // TODO: Something has gone terribly wrong - the app wasn't
            // able to query or create the collection.
            // Verify your connection, endpoint, and key.
            System.out.println("Something has gone terribly wrong - " +
                               "the app wasn't able to create the Database.\n");
            e.printStackTrace();
        }

        try {

            if (cosmosContainer == null) {
                CosmosContainerProperties properties = new
CosmosContainerProperties(CONTAINER_ID,"/id");
                cosmosContainer = cosmosDatabase.createContainerIfNotExists(properties).getContainer();
```

```
            cosmosContainer = cosmosDatabase.createContainerIfNotExists(properties).getContainer();
        }

    } catch(CosmosClientException e) {
        // TODO: Something has gone terribly wrong - the app wasn't
        // able to query or create the collection.
        // Verify your connection, endpoint, and key.
        System.out.println("Something has gone terribly wrong - " +
                "the app wasn't able to create the Container.\n");
        e.printStackTrace();
    }

    return cosmosContainer;
}

private JsonNode getDocumentById(String id) {

    String sql = "SELECT * FROM root r WHERE r.id='" + id + "'";
    int maxItemCount = 1000;
    int maxDegreeOfParallelism = 1000;
    int maxBufferedItemCount = 100;

    FeedOptions options = new FeedOptions();
    options.setMaxItemCount(maxItemCount);
    options.setMaxBufferedItemCount(maxBufferedItemCount);
    options.setMaxDegreeOfParallelism(maxDegreeOfParallelism);
    options.setRequestContinuation(null);
    options.setPopulateQueryMetrics(false);

    List<JsonNode> itemList = new ArrayList();

    do {

        for (FeedResponse<JsonNode> pageResponse:
            getContainerCreateResourcesIfNotExist()
                .queryItems(sql, options, JsonNode.class)
                .iterableByPage()) {

            options.setRequestContinuation(pageResponse.getContinuationToken());

            for (JsonNode item : pageResponse.getElements()) {
                itemList.add(item);
            }
        }

    } while(options.getRequestContinuation() != null);

    if (itemList.size() > 0) {
        return itemList.get(0);
    } else {
        return null;
    }
}

}
```

5. Next, create a new *TodoDaoFactory.java* file and add the `TodoDaoFactory` class that creates a new DocDbDao object:

```
package com.microsoft.azure.documentdb.sample.dao;

public class TodoDaoFactory {
    private static TodoDao myTodoDao = new DocDbDao();

    public static TodoDao getDao() {
        return myTodoDao;
    }
}
```

Add a controller

Add the *TodoItemController* controller to your application. In this project, you are using Project Lombok to generate the constructor, getters, setters, and a builder. Alternatively, you can write this code manually or have the IDE generate it.:

```
package com.microsoft.azure.documentdb.sample.controller;

import java.util.List;
import java.util.UUID;

import lombok.NonNull;

import com.microsoft.azure.documentdb.sample.dao.TodoDao;
import com.microsoft.azure.documentdb.sample.dao.TodoDaoFactory;
import com.microsoft.azure.documentdb.sample.model.TodoItem;

public class TodoItemController {
    public static TodoItemController getInstance() {
        if (todoItemController == null) {
            todoItemController = new TodoItemController(TodoDaoFactory.getDao());
        }
        return todoItemController;
    }

    private static TodoItemController todoItemController;

    private final TodoDao todoDao;

    TodoItemController(TodoDao todoDao) {
        this.todoDao = todoDao;
    }

    public TodoItem createTodoItem(@NonNull String name,
            @NonNull String category, boolean isComplete) {
        TodoItem todoItem = new TodoItem();

        todoItem.setName(name);
        todoItem.setCategory(category);
        todoItem.setComplete(isComplete);
        todoItem.setId(UUID.randomUUID().toString());

        return todoDao.createTodoItem(todoItem);
    }

    public boolean deleteTodoItem(@NonNull String id) {
        return todoDao.deleteTodoItem(id);
    }

    public TodoItem getTodoItemById(@NonNull String id) {
        return todoDao.readTodoItem(id);
    }

    public List<TodoItem> getTodoItems() {
        return todoDao.readTodoItems();
    }

    public TodoItem updateTodoItem(@NonNull String id, boolean isComplete) {
        return todoDao.updateTodoItem(id, isComplete);
    }
}
```

Create a servlet

Next, create a servlet to route HTTP requests to the controller. Create the *ApiServlet.java* file and define the following code under it:

```
package com.microsoft.azure.documentdb.sample;

import java.io.IOException;

import javax.servlet.ServletException;
```

```
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.apache.commons.lang3.StringUtils;

import com.google.gson.Gson;
import com.microsoft.azure.documentdb.sample.controller.TodoItemController;

/**
 * API Frontend Servlet
 */
@WebServlet("/api")
public class ApiServlet extends HttpServlet {
    // API Keys
    public static final String API_METHOD = "method";

    // API Methods
    public static final String CREATE_TODO_ITEM = "createTodoItem";
    public static final String GET_TODO_ITEMS = "getTodoItems";
    public static final String UPDATE_TODO_ITEM = "updateTodoItem";

    // API Parameters
    public static final String TODO_ITEM_ID = "todoItemId";
    public static final String TODO_ITEM_NAME = "todoItemName";
    public static final String TODO_ITEM_CATEGORY = "todoItemCategory";
    public static final String TODO_ITEM_COMPLETE = "todoItemComplete";

    public static final String MESSAGE_ERROR_INVALID_METHOD = "{'error': 'Invalid method'}";

    private static final long serialVersionUID = 1L;
    private static final Gson gson = new Gson();

    @Override
    protected void doGet(HttpServletRequest request,
            HttpServletResponse response) throws ServletException, IOException {

        String apiResponse = MESSAGE_ERROR_INVALID_METHOD;

        TodoItemController todoItemController = TodoItemController
                .getInstance();

        String id = request.getParameter(TODO_ITEM_ID);
        String name = request.getParameter(TODO_ITEM_NAME);
        String category = request.getParameter(TODO_ITEM_CATEGORY);
        boolean isComplete = StringUtils.equalsIgnoreCase("true",
                request.getParameter(TODO_ITEM_COMPLETE)) ? true : false;

        switch (request.getParameter(API_METHOD)) {
        case CREATE_TODO_ITEM:
            apiResponse = gson.toJson(todoItemController.createTodoItem(name,
                    category, isComplete));
            break;
        case GET_TODO_ITEMS:
            apiResponse = gson.toJson(todoItemController.getTodoItems());
            break;
        case UPDATE_TODO_ITEM:
            apiResponse = gson.toJson(todoItemController.updateTodoItem(id,
                    isComplete));
            break;
        default:
            break;
        }

        response.setCharacterEncoding("UTF-8");
        response.getWriter().println(apiResponse);
    }
```

```
    @Override
    protected void doPost(HttpServletRequest request,
            HttpServletResponse response) throws ServletException, IOException {
        doGet(request, response);
    }
}
```

Wire the rest of the of Java app together

Now that we've finished the fun bits, all that's left is to build a quick user interface and wire it up to your DAO.

1. You need a web user interface to display to the user. Let's re-write the *index.jsp* we created earlier with the following code:

```
<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge;" />
  <title>Azure DocumentDB Java Sample</title>

  <!-- Bootstrap -->
  <link href="//ajax.aspnetcdn.com/ajax/bootstrap/3.2.0/css/bootstrap.min.css" rel="stylesheet">

  <style>
    /* Add padding to body for fixed nav bar */
    body {
      padding-top: 50px;
    }
  </style>
</head>
<body>
  <!-- Nav Bar -->
  <div class="navbar navbar-inverse navbar-fixed-top" role="navigation">
    <div class="container">
      <div class="navbar-header">
        <a class="navbar-brand" href="#">My Tasks</a>
      </div>
    </div>
  </div>

  <!-- Body -->
  <div class="container">
    <h1>My ToDo List</h1>

    <hr/>

    <!-- The ToDo List -->
    <div class = "todoList">
      <table class="table table-bordered table-striped" id="todoItems">
        <thead>
          <tr>
            <th>Name</th>
            <th>Category</th>
            <th>Complete</th>
          </tr>
        </thead>
        <tbody>
        </tbody>
      </table>

      <!-- Update Button -->
      <div class="todoUpdatePanel">
        <form class="form-horizontal" role="form">
          <button type="button" class="btn btn-primary">Update Tasks</button>
```

```
          <button type="button" class="btn btn-primary">Update Tasks</button>
        </form>
      </div>

    </div>

    <hr/>

    <!-- Item Input Form -->
    <div class="todoForm">
      <form class="form-horizontal" role="form">
        <div class="form-group">
          <label for="inputItemName" class="col-sm-2">Task Name</label>
          <div class="col-sm-10">
            <input type="text" class="form-control" id="inputItemName" placeholder="Enter name">
          </div>
        </div>

        <div class="form-group">
          <label for="inputItemCategory" class="col-sm-2">Task Category</label>
          <div class="col-sm-10">
            <input type="text" class="form-control" id="inputItemCategory" placeholder="Enter
category">
          </div>
        </div>

        <button type="button" class="btn btn-primary">Add Task</button>
      </form>
    </div>

  </div>

  <!-- Placed at the end of the document so the pages load faster -->
  <script src="//ajax.aspnetcdn.com/ajax/jQuery/jquery-2.1.1.min.js"></script>
  <script src="//ajax.aspnetcdn.com/ajax/bootstrap/3.2.0/bootstrap.min.js"></script>
  <script src="assets/todo.js"></script>
</body>
</html>
```

2. Finally, write some client-side JavaScript to tie the web user interface and the servlet together:

```
/**
 * ToDo App
 */

var todoApp = {
  /*
   * API methods to call Java backend.
   */
  apiEndpoint: "api",

  createTodoItem: function(name, category, isComplete) {
    $.post(todoApp.apiEndpoint, {
        "method": "createTodoItem",
        "todoItemName": name,
        "todoItemCategory": category,
        "todoItemComplete": isComplete
      },
      function(data) {
        var todoItem = data;
        todoApp.addTodoItemToTable(todoItem.id, todoItem.name, todoItem.category, todoItem.complete);
      },
      "json");
  },

  getTodoItems: function() {
    $.post(todoApp.apiEndpoint, {
        "method": "getTodoItems"
```

```
        "method": "getTodoItems"
      },
      function(data) {
        var todoItemArr = data;
        $.each(todoItemArr, function(index, value) {
          todoApp.addTodoItemToTable(value.id, value.name, value.category, value.complete);
        });
      },
      "json");
  },

  updateTodoItem: function(id, isComplete) {
    $.post(todoApp.apiEndpoint, {
        "method": "updateTodoItem",
        "todoItemId": id,
        "todoItemComplete": isComplete
      },
      function(data) {},
      "json");
  },

  /*
   * UI Methods
   */
  addTodoItemToTable: function(id, name, category, isComplete) {
    var rowColor = isComplete ? "active" : "warning";

    todoApp.ui_table().append($("<tr>")
      .append($("<td>").text(name))
      .append($("<td>").text(category))
      .append($("<td>")
        .append($("<input>")
          .attr("type", "checkbox")
          .attr("id", id)
          .attr("checked", isComplete)
          .attr("class", "isComplete")
        ))
      .addClass(rowColor)
    );
  },

  /*
   * UI Bindings
   */
  bindCreateButton: function() {
    todoApp.ui_createButton().click(function() {
      todoApp.createTodoItem(todoApp.ui_createNameInput().val(),
todoApp.ui_createCategoryInput().val(), false);
      todoApp.ui_createNameInput().val("");
      todoApp.ui_createCategoryInput().val("");
    });
  },

  bindUpdateButton: function() {
    todoApp.ui_updateButton().click(function() {
      // Disable button temporarily.
      var myButton = $(this);
      var originalText = myButton.text();
      $(this).text("Updating...");
      $(this).prop("disabled", true);

      // Call api to update todo items.
      $.each(todoApp.ui_updateId(), function(index, value) {
        todoApp.updateTodoItem(value.name, value.value);
        $(value).remove();
      });

      // Re-enable button.
      setTimeout(function() {
        myButton.prop("disabled", false);
```

```
      myButton.prop("disabled", false);
      myButton.text(originalText);
    }, 500);
  });
},

bindUpdateCheckboxes: function() {
  todoApp.ui_table().on("click", ".isComplete", function(event) {
    var checkboxElement = $(event.currentTarget);
    var rowElement = $(event.currentTarget).parents('tr');
    var id = checkboxElement.attr('id');
    var isComplete = checkboxElement.is(':checked');

    // Togle table row color
    if (isComplete) {
      rowElement.addClass("active");
      rowElement.removeClass("warning");
    } else {
      rowElement.removeClass("active");
      rowElement.addClass("warning");
    }

    // Update hidden inputs for update panel.
    todoApp.ui_updateForm().children("input[name='" + id + "']").remove();

    todoApp.ui_updateForm().append($("<input>")
      .attr("type", "hidden")
      .attr("class", "updateComplete")
      .attr("name", id)
      .attr("value", isComplete));

  });
},

/*
 * UI Elements
 */
ui_createNameInput: function() {
  return $(".todoForm #inputItemName");
},

ui_createCategoryInput: function() {
  return $(".todoForm #inputItemCategory");
},

ui_createButton: function() {
  return $(".todoForm button");
},

ui_table: function() {
  return $(".todoList table tbody");
},

ui_updateButton: function() {
  return $(".todoUpdatePanel button");
},

ui_updateForm: function() {
  return $(".todoUpdatePanel form");
},

ui_updateId: function() {
  return $(".todoUpdatePanel .updateComplete");
},

/*
 * Install the TodoApp
 */
install: function() {
```

```
    todoApp.bindCreateButton();
    todoApp.bindUpdateButton();
    todoApp.bindUpdateCheckboxes();

    todoApp.getTodoItems();
  }
};

$(document).ready(function() {
  todoApp.install();
});
```

3. Now all that's left is to test the application. Run the application locally, and add some Todo items by filling in the item name and category and clicking **Add Task**. After the item appears, you can update whether it's complete by toggling the checkbox and clicking **Update Tasks**.

Deploy your Java application to Azure Web Sites

Azure Web Sites makes deploying Java applications as simple as exporting your application as a WAR file and either uploading it via source control (e.g. Git) or FTP.

1. To export your application as a WAR file, right-click on your project in **Project Explorer**, click **Export**, and then click **WAR File**.

2. In the **WAR Export** window, do the following:

   - In the Web project box, enter azure-documentdb-java-sample.
   - In the Destination box, choose a destination to save the WAR file.
   - Click **Finish**.
3. Now that you have a WAR file in hand, you can simply upload it to your Azure Web Site's **webapps** directory. For instructions on uploading the file, see Add a Java application to Azure App Service Web Apps. After the WAR file is uploaded to the webapps directory, the runtime environment will detect that you've added it and will automatically load it.

4. To view your finished product, navigate to `http://YOUR\_SITE\_NAME.azurewebsites.net/azure-java-sample/` and start adding your tasks!

Get the project from GitHub

All the samples in this tutorial are included in the todo project on GitHub. To import the todo project into Eclipse, ensure you have the software and resources listed in the Prerequisites section, then do the following:

1. Install Project Lombok. Lombok is used to generate constructors, getters, setters in the project. Once you have downloaded the lombok.jar file, double-click it to install it or install it from the command line.

2. If Eclipse is open, close it and restart it to load Lombok.

3. In Eclipse, on the **File** menu, click **Import**.

4. In the **Import** window, click **Git**, click **Projects from Git**, and then click **Next**.

5. On the **Select Repository Source** screen, click **Clone URI**.

6. On the **Source Git Repository** screen, in the **URI** box, enter https://github.com/Azure-Samples/documentdb-java-todo-app.git, and then click **Next**.

7. On the **Branch Selection** screen, ensure that **main** is selected, and then click **Next**.

8. On the **Local Destination** screen, click **Browse** to select a folder where the repository can be copied, and then click **Next**.

9. On the **Select a wizard to use for importing projects** screen, ensure that **Import existing projects** is selected, and then click **Next**.
10. On the **Import Projects** screen, unselect the **DocumentDB** project, and then click **Finish**. The DocumentDB project contains the Azure Cosmos DB Java SDK, which we will add as a dependency instead.
11. In **Project Explorer**, navigate to azure-documentdb-java-sample\src\com.microsoft.azure.documentdb.sample.dao\DocumentClientFactory.java and replace the HOST and MASTER_KEY values with the URI and PRIMARY KEY for your Azure Cosmos DB account, and then save the file. For more information, see Step 1. Create an Azure Cosmos database account.
12. In **Project Explorer**, right-click the **azure-documentdb-java-sample**, click **Build Path**, and then click **Configure Build Path**.
13. On the **Java Build Path** screen, in the right pane, select the **Libraries** tab, and then click **Add External JARs**. Navigate to the location of the lombok.jar file, and click **Open**, and then click **OK**.
14. Use step 12 to open the **Properties** window again, and then in the left pane click **Targeted Runtimes**.
15. On the **Targeted Runtimes** screen, click **New**, select **Apache Tomcat v7.0**, and then click **OK**.
16. Use step 12 to open the **Properties** window again, and then in the left pane click **Project Facets**.
17. On the **Project Facets** screen, select **Dynamic Web Module** and **Java**, and then click **OK**.
18. On the **Servers** tab at the bottom of the screen, right-click **Tomcat v7.0 Server at localhost** and then click **Add and Remove**.
19. On the **Add and Remove** window, move **azure-documentdb-java-sample** to the **Configured** box, and then click **Finish**.
20. In the **Servers** tab, right-click **Tomcat v7.0 Server at localhost**, and then click **Restart**.
21. In a browser, navigate to `http://localhost:8080/azure-documentdb-java-sample/` and start adding to your task list. Note that if you changed your default port values, change 8080 to the value you selected.
22. To deploy your project to an Azure web site, see Step 6. Deploy your application to Azure Web Sites.

Next steps

Build a node.js application with Azure Cosmos DB

Tutorial: Build a Node.js web app using the JavaScript SDK to manage a SQL API account in Azure Cosmos DB

11/2/2020 • 13 minutes to read • Edit Online

APPLIES TO: SQL API

As a developer, you might have applications that use NoSQL document data. You can use a SQL API account in Azure Cosmos DB to store and access this document data. This Node.js tutorial shows you how to store and access data from a SQL API account in Azure Cosmos DB by using a Node.js Express application that is hosted on the Web Apps feature of Microsoft Azure App Service. In this tutorial, you will build a web-based application (Todo app) that allows you to create, retrieve, and complete tasks. The tasks are stored as JSON documents in Azure Cosmos DB.

This tutorial demonstrates how to create a SQL API account in Azure Cosmos DB by using the Azure portal. You then build and run a web app that is built on the Node.js SDK to create a database and container, and add items to the container. This tutorial uses JavaScript SDK version 3.0.

This tutorial covers the following tasks:

- Create an Azure Cosmos DB account
- Create a new Node.js application
- Connect the application to Azure Cosmos DB
- Run and deploy the application to Azure

Prerequisites

Before following the instructions in this article, ensure that you have the following resources:

- If you don't have an Azure subscription, create a free account before you begin.

  You can Try Azure Cosmos DB for free without an Azure subscription, free of charge and commitments, or create an Azure Cosmos DB free tier account, with the first 400 RU/s and 5 GB of storage for free. You can also use the Azure Cosmos DB Emulator with a URI of `https://localhost:8081`. For the key to use with the emulator, see Authenticating requests.

- Node.js version 6.10 or higher.
- Express generator (you can install Express via `npm install express-generator -g`)
- Install Git on your local workstation.

Create an Azure Cosmos DB account

Let's start by creating an Azure Cosmos DB account. If you already have an account or if you are using the Azure Cosmos DB Emulator for this tutorial, you can skip to Step 2: Create a new Node.js application.

1. From the Azure portal menu or the **Home page**, select **Create a resource**.
2. On the **New** page, search for and select **Azure Cosmos DB**.
3. On the **Azure Cosmos DB** page, select **Create**.

4. On the **Create Azure Cosmos DB Account** page, enter the basic settings for the new Azure Cosmos account.

SETTING	VALUE	DESCRIPTION
Subscription	Subscription name	Select the Azure subscription that you want to use for this Azure Cosmos account.
Resource Group	Resource group name	Select a resource group, or select **Create new**, then enter a unique name for the new resource group.
Account Name	A unique name	Enter a name to identify your Azure Cosmos account. Because *documents.azure.com* is appended to the name that you provide to create your URI, use a unique name. The name can only contain lowercase letters, numbers, and the hyphen (-) character. It must be between 3-44 characters in length.
API	The type of account to create	Select **Core (SQL)** to create a document database and query by using SQL syntax. The API determines the type of account to create. Azure Cosmos DB provides five APIs: Core (SQL) and MongoDB for document data, Gremlin for graph data, Azure Table, and Cassandra. Currently, you must create a separate account for each API.
Capacity mode	Provisioned throughput or Serverless	Select **Provisioned throughput** to create an account in provisioned throughput mode. Select **Serverless** to create an account in serverless mode.
Apply Free Tier Discount	Apply or Do not apply	With Azure Cosmos DB free tier, you will get the first 400 RU/s and 5 GB of storage for free in an account. Learn more about free tier.
Location	The region closest to your users	Select a geographic location to host your Azure Cosmos DB account. Use the location that is closest to your users to give them the fastest access to the data.

SETTING	VALUE	DESCRIPTION
Account Type	Production or Non-Production	Select **Production** if the account will be used for a production workload. Select **Non-Production** if the account will be used for non-production, e.g. development, testing, QA, or staging. This is an Azure resource tag setting that tunes the Portal experience but does not affect the underlying Azure Cosmos DB account. You can change this value anytime.

NOTE

You can have up to one free tier Azure Cosmos DB account per Azure subscription and must opt-in when creating the account. If you do not see the option to apply the free tier discount, this means another account in the subscription has already been enabled with free tier.

NOTE

The following options are not available if you select **Serverless** as the **Capacity mode**:

- Apply Free Tier Discount
- Geo-redundancy
- Multi-region Writes

5. Select **Review + create**. You can skip the **Network** and **Tags** sections.

6. Review the account settings, and then select **Create**. It takes a few minutes to create the account. Wait for the portal page to display **Your deployment is complete**.

7. Select **Go to resource** to go to the Azure Cosmos DB account page.

Go to the Azure Cosmos DB account page, and select **Keys**. Copy the values to use in the web application you create next.

Create a new Node.js application

Now let's learn to create a basic Hello World Node.js project using the Express framework.

1. Open your favorite terminal, such as the Node.js command prompt.

2. Navigate to the directory in which you'd like to store the new application.

3. Use the express generator to generate a new application called **todo**.

```
express todo
```

4. Open the **todo** directory and install dependencies.

```
cd todo
npm install
```

5. Run the new application.

```
npm start
```

6. You can view your new application by navigating your browser to `http://localhost:3000`.

Stop the application by using CTRL+C in the terminal window, and select **y** to terminate the batch job.

Install the required modules

The **package.json** file is one of the files created in the root of the project. This file contains a list of additional modules that are required for your Node.js application. When you deploy this application to Azure, this file is used to determine which modules should be installed on Azure to support your application. Install two more packages for this tutorial.

1. Install the **@azure/cosmos** module via npm.

```
npm install @azure/cosmos
```

Connect the Node.js application to Azure Cosmos DB

Now that you have completed the initial setup and configuration, next you will write code that is required by the todo application to communicate with Azure Cosmos DB.

Create the model

1. At the root of your project directory, create a new directory named **models**.

2. In the **models** directory, create a new file named **taskDao.js**. This file contains code required to create the database and container. It also defines methods to read, update, create, and find tasks in Azure Cosmos DB.

3. Copy the following code into the **taskDao.js** file:

```
// @ts-check
const CosmosClient = require('@azure/cosmos').CosmosClient
const debug = require('debug')('todo:taskDao')

// For simplicity we'll set a constant partition key
const partitionKey = undefined
class TaskDao {
  /**
```

```
     * Manages reading, adding, and updating Tasks in Cosmos DB
     * @param {CosmosClient} cosmosClient
     * @param {string} databaseId
     * @param {string} containerId
     */
    constructor(cosmosClient, databaseId, containerId) {
      this.client = cosmosClient
      this.databaseId = databaseId
      this.collectionId = containerId

      this.database = null
      this.container = null
    }

    async init() {
      debug('Setting up the database...')
      const dbResponse = await this.client.databases.createIfNotExists({
        id: this.databaseId
      })
      this.database = dbResponse.database
      debug('Setting up the database...done!')
      debug('Setting up the container...')
      const coResponse = await this.database.containers.createIfNotExists({
        id: this.collectionId
      })
      this.container = coResponse.container
      debug('Setting up the container...done!')
    }

    async find(querySpec) {
      debug('Querying for items from the database')
      if (!this.container) {
        throw new Error('Collection is not initialized.')
      }
      const { resources } = await this.container.items.query(querySpec).fetchAll()
      return resources
    }

    async addItem(item) {
      debug('Adding an item to the database')
      item.date = Date.now()
      item.completed = false
      const { resource: doc } = await this.container.items.create(item)
      return doc
    }

    async updateItem(itemId) {
      debug('Update an item in the database')
      const doc = await this.getItem(itemId)
      doc.completed = true

      const { resource: replaced } = await this.container
        .item(itemId, partitionKey)
        .replace(doc)
      return replaced
    }

    async getItem(itemId) {
      debug('Getting an item from the database')
      const { resource } = await this.container.item(itemId, partitionKey).read()
      return resource
    }
  }

  module.exports = TaskDao
```

4. Save and close the **taskDao.js** file.

Create the controller

1. In the **routes** directory of your project, create a new file named **tasklist.js**.

2. Add the following code to **tasklist.js**. This code loads the CosmosClient and async modules, which are used by **tasklist.js**. This code also defines the **TaskList** class, which is passed as an instance of the **TaskDao** object we defined earlier:

```
const TaskDao = require("../models/TaskDao");

class TaskList {
  /**
   * Handles the various APIs for displaying and managing tasks
   * @param {TaskDao} taskDao
   */
  constructor(taskDao) {
    this.taskDao = taskDao;
  }
  async showTasks(req, res) {
    const querySpec = {
      query: "SELECT * FROM root r WHERE r.completed=@completed",
      parameters: [
        {
          name: "@completed",
          value: false
        }
      ]
    };

    const items = await this.taskDao.find(querySpec);
    res.render("index", {
      title: "My ToDo List ",
      tasks: items
    });
  }

  async addTask(req, res) {
    const item = req.body;

    await this.taskDao.addItem(item);
    res.redirect("/");
  }

  async completeTask(req, res) {
    const completedTasks = Object.keys(req.body);
    const tasks = [];

    completedTasks.forEach(task => {
      tasks.push(this.taskDao.updateItem(task));
    });

    await Promise.all(tasks);

    res.redirect("/");
  }
}

module.exports = TaskList;
```

3. Save and close the **tasklist.js** file.

Add config.js

1. At the root of your project directory, create a new file named **config.js**.

2. Add the following code to **config.js** file. This code defines configuration settings and values needed for our

application.

```
const config = {};

config.host = process.env.HOST || "[the endpoint URI of your Azure Cosmos DB account]";
config.authKey =
  process.env.AUTH_KEY || "[the PRIMARY KEY value of your Azure Cosmos DB account";
config.databaseId = "ToDoList";
config.containerId = "Items";

if (config.host.includes("https://localhost:")) {
  console.log("Local environment detected");
  console.log("WARNING: Disabled checking of self-signed certs. Do not have this code in production.");
  process.env.NODE_TLS_REJECT_UNAUTHORIZED = "0";
  console.log(`Go to http://localhost:${process.env.PORT || '3000'} to try the sample.`);
}

module.exports = config;
```

3. In the **config.js** file, update the values of HOST and AUTH_KEY using the values found in the Keys page of your Azure Cosmos DB account on the Azure portal.

4. Save and close the **config.js** file.

Modify app.js

1. In the project directory, open the **app.js** file. This file was created earlier when the Express web application was created.

2. Add the following code to the **app.js** file. This code defines the config file to be used, and loads the values into some variables that you will use in the next sections.

```
const CosmosClient = require('@azure/cosmos').CosmosClient
const config = require('./config')
const TaskList = require('./routes/tasklist')
const TaskDao = require('./models/taskDao')

const express = require('express')
const path = require('path')
const logger = require('morgan')
const cookieParser = require('cookie-parser')
const bodyParser = require('body-parser')

const app = express()

// view engine setup
app.set('views', path.join(__dirname, 'views'))
app.set('view engine', 'jade')

// uncomment after placing your favicon in /public
//app.use(favicon(path.join(__dirname, 'public', 'favicon.ico')));
app.use(logger('dev'))
app.use(bodyParser.json())
app.use(bodyParser.urlencoded({ extended: false }))
app.use(cookieParser())
app.use(express.static(path.join(__dirname, 'public')))

//Todo App:
const cosmosClient = new CosmosClient({
  endpoint: config.host,
  key: config.authKey
})
const taskDao = new TaskDao(cosmosClient, config.databaseId, config.containerId)
const taskList = new TaskList(taskDao)
taskDao
```

```
  .init(err => {
    console.error(err)
  })
  .catch(err => {
    console.error(err)
    console.error(
      'Shutting down because there was an error settinig up the database.'
    )
    process.exit(1)
  })

app.get('/', (req, res, next) => taskList.showTasks(req, res).catch(next))
app.post('/addtask', (req, res, next) => taskList.addTask(req, res).catch(next))
app.post('/completetask', (req, res, next) =>
  taskList.completeTask(req, res).catch(next)
)
app.set('view engine', 'jade')

// catch 404 and forward to error handler
app.use(function(req, res, next) {
  const err = new Error('Not Found')
  err.status = 404
  next(err)
})

// error handler
app.use(function(err, req, res, next) {
  // set locals, only providing error in development
  res.locals.message = err.message
  res.locals.error = req.app.get('env') === 'development' ? err : {}

  // render the error page
  res.status(err.status || 500)
  res.render('error')
})

module.exports = app
```

3. Finally, save and close the **app.js** file.

Build a user interface

Now let's build the user interface so that a user can interact with the application. The Express application we created in the previous sections uses **Jade** as the view engine.

1. The **layout.jade** file in the **views** directory is used as a global template for other **.jade** files. In this step you will modify it to use Twitter Bootstrap, which is a toolkit used to design a website.

2. Open the **layout.jade** file found in the **views** folder and replace the contents with the following code:

```
doctype html
html
  head
    title= title
    link(rel='stylesheet', href='//ajax.aspnetcdn.com/ajax/bootstrap/3.3.2/css/bootstrap.min.css')
    link(rel='stylesheet', href='/stylesheets/style.css')
  body
    nav.navbar.navbar-inverse.navbar-fixed-top
      div.navbar-header
        a.navbar-brand(href='#') My Tasks
    block content
    script(src='//ajax.aspnetcdn.com/ajax/jQuery/jquery-1.11.2.min.js')
    script(src='//ajax.aspnetcdn.com/ajax/bootstrap/3.3.2/bootstrap.min.js')
```

This code tells the **Jade** engine to render some HTML for our application, and creates a **block** called **content** where we can supply the layout for our content pages. Save and close the **layout.jade** file.

3. Now open the **index.jade** file, the view that will be used by our application, and replace the content of the file with the following code:

```
extends layout
block content
     h1 #{title}
     br

     form(action="/completetask", method="post")
      table.table.table-striped.table-bordered
         tr
           td Name
           td Category
           td Date
           td Complete
         if (typeof tasks === "undefined")
           tr
             td
         else
           each task in tasks
             tr
               td #{task.name}
               td #{task.category}
               - var date  = new Date(task.date);
               - var day   = date.getDate();
               - var month = date.getMonth() + 1;
               - var year  = date.getFullYear();
               td #{month + "/" + day + "/" + year}
               td
                if(task.completed)
                 input(type="checkbox", name="#{task.id}", value="#{!task.completed}",
checked=task.completed)
                else
                 input(type="checkbox", name="#{task.id}", value="#{!task.completed}",
checked=task.completed)
       button.btn.btn-primary(type="submit") Update tasks
     hr
     form.well(action="/addtask", method="post")
       label Item Name:
       input(name="name", type="textbox")
       label Item Category:
       input(name="category", type="textbox")
       br
       button.btn(type="submit") Add item
```

This code extends layout, and provides content for the **content** placeholder we saw in the **layout.jade** file earlier. In this layout, we created two HTML forms.

The first form contains a table for your data and a button that allows you to update items by posting to **/completeTask** method of the controller.

The second form contains two input fields and a button that allows you to create a new item by posting to **/addtask** method of the controller. That's all we need for the application to work.

Run your application locally

Now that you have built the application, you can run it locally by using the following steps:

1. To test the application on your local machine, run `npm start` in the terminal to start your application, and then refresh the `http://localhost:3000` browser page. The page should now look as shown in the following

screenshot:

> **TIP**
> If you receive an error about the indent in the layout.jade file or the index.jade file, ensure that the first two lines in both files are left-justified, with no spaces. If there are spaces before the first two lines, remove them, save both files, and then refresh your browser window.

2. Use the Item, Item Name, and Category fields to enter a new task, and then select **Add Item**. It creates a document in Azure Cosmos DB with those properties.

3. The page should update to display the newly created item in the ToDo list.

4. To complete a task, select the check box in the Complete column, and then select **Update tasks**. It updates the document you already created and removes it from the view.

5. To stop the application, press CTRL+C in the terminal window and then select **Y** to terminate the batch job.

Deploy your application to Web Apps

After your application succeeds locally, you can deploy it to Azure by using the following steps:

1. If you haven't already done so, enable a git repository for your Web Apps application.

2. Add your Web Apps application as a git remote.

```
git remote add azure https://username@your-azure-website.scm.azurewebsites.net:443/your-azure-website.git
```

3. Deploy the application by pushing it to the remote.

```
git push azure master
```

4. In a few seconds, your web application is published and launched in a browser.

Clean up resources

When these resources are no longer needed, you can delete the resource group, Azure Cosmos DB account, and all the related resources. To do so, select the resource group that you used for the Azure Cosmos DB account, select **Delete**, and then confirm the name of the resource group to delete.

Next steps

Tutorial: Build mobile applications with Xamarin and Azure Cosmos DB

12/4/2020 • 5 minutes to read • Edit Online

APPLIES TO: SQL API

Most mobile apps need to store data in the cloud, and Azure Cosmos DB is a cloud database for mobile apps. It has everything a mobile developer need. It is a fully managed database as a service that scales on demand. It can bring your data to your application transparently, wherever your users are located around the globe. By using the Azure Cosmos DB .NET Core SDK, you can enable Xamarin mobile apps to interact directly with Azure Cosmos DB, without a middle tier.

This article provides a tutorial for building mobile apps with Xamarin and Azure Cosmos DB. You can find the complete source code for the tutorial at Xamarin and Azure Cosmos DB on GitHub, including how to manage users and permissions.

Azure Cosmos DB capabilities for mobile apps

Azure Cosmos DB provides the following key capabilities for mobile app developers:

- Rich queries over schemaless data. Azure Cosmos DB stores data as schemaless JSON documents in heterogeneous collections. It offers rich and fast queries without the need to worry about schemas or indexes.
- Fast throughput. It takes only a few milliseconds to read and write documents with Azure Cosmos DB. Developers can specify the throughput they need, and Azure Cosmos DB honors it with 99.99% availability SLA for all single region accounts and all multi-region accounts with relaxed consistency, and 99.999% read availability on all multi-region database accounts.
- Limitless scale. Your Azure Cosmos containers grow as your app grows. You can start with small data size and throughput of hundreds of requests per second. Your collections or databases can grow to petabytes of data and arbitrarily large throughput with hundreds of millions of requests per second.
- Globally distributed. Mobile app users are on the go, often across the world. Azure Cosmos DB is a globally distributed database. Click the map to make your data accessible to your users.
- Built-in rich authorization. With Azure Cosmos DB, you can easily implement popular patterns like per-user data or multiuser shared data, without complex custom authorization code.
- Geospatial queries. Many mobile apps offer geo-contextual experiences today. With first-class support for geospatial types, Azure Cosmos DB makes creating these experiences easy to accomplish.
- Binary attachments. Your app data often includes binary blobs. Native support for attachments makes it easier

to use Azure Cosmos DB as a one-stop shop for your app data.

Azure Cosmos DB and Xamarin tutorial

The following tutorial shows how to build a mobile application by using Xamarin and Azure Cosmos DB. You can find the complete source code for the tutorial at Xamarin and Azure Cosmos DB on GitHub.

Get started

It's easy to get started with Azure Cosmos DB. Go to the Azure portal, and create a new Azure Cosmos DB account. Click the **Quick start** tab. Download the Xamarin Forms to-do list sample that is already connected to your Azure Cosmos DB account.

Or if you have an existing Xamarin app, you can add the Azure Cosmos DB NuGet package. Azure Cosmos DB supports Xamarin.iOS, Xamarin.Android, and Xamarin Forms shared libraries.

Work with data

Your data records are stored in Azure Cosmos DB as schemaless JSON documents in heterogeneous collections. You can store documents with different structures in the same collection:

```
var result = await client.CreateDocumentAsync(collectionLink, todoItem);
```

In your Xamarin projects, you can use language-integrated queries over schemaless data:

```
var query = await client.CreateDocumentQuery<ToDoItem>(collectionLink)
                .Where(todoItem => todoItem.Complete == false)
                .AsDocumentQuery();

Items = new List<TodoItem>();
while (query.HasMoreResults) {
    Items.AddRange(await query.ExecuteNextAsync<TodoItem>());
}
```

Add users

Like many get started samples, the Azure Cosmos DB sample you downloaded authenticates to the service by using a primary key hardcoded in the app's code. This default is not a good practice for an app you intend to run anywhere except on your local emulator. If an unauthorized user obtained the primary key, all the data across your Azure Cosmos DB account could be compromised. Instead, you want your app to access only the records for the signed-in user. Azure Cosmos DB allows developers to grant application read or read/write permission to a collection, a set of documents grouped by a partition key, or a specific document.

Follow these steps to modify the to-do list app to a multiuser to-do list app:

1. Add Login to your app by using Facebook, Active Directory, or any other provider.
2. Create an Azure Cosmos DB UserItems collection with **/userId** as the partition key. Specifying the partition key for your collection allows Azure Cosmos DB to scale infinitely as the number of your app users grows, while continuing to offer fast queries.
3. Add Azure Cosmos DB Resource Token Broker. This simple Web API authenticates users and issues short-lived tokens to signed-in users with access only to the documents within their partition. In this example, Resource Token Broker is hosted in App Service.
4. Modify the app to authenticate to Resource Token Broker with Facebook, and request the resource tokens for the signed-in Facebook users. You can then access their data in the UserItems collection.

You can find a complete code sample of this pattern at Resource Token Broker on GitHub. This diagram illustrates the solution:

If you want two users to have, access to the same to-do list, you can add additional permissions to the access

token in Resource Token Broker.

Scale on demand

Azure Cosmos DB is a managed database as a service. As your user base grows, you don't need to worry about provisioning VMs or increasing cores. All you need to tell Azure Cosmos DB is how many operations per second (throughput) your app needs. You can specify the throughput via the **Scale** tab by using a measure of throughput called Request Units (RUs) per second. For example, a read operation on a 1-KB document requires one RU. You can also add alerts to the **Throughput** metric to monitor the traffic growth and programmatically change the throughput as alerts fire.

Go planet scale

As your app gains popularity, you might gain users across the globe. Or maybe you want to be prepared for unforeseen events. Go to the Azure portal, and open your Azure Cosmos DB account. Click the map to make your data continuously replicate to any number of regions across the world. This capability makes your data available wherever your users are. You can also add failover policies to be prepared for contingencies.

Congratulations. You have completed the solution and have a mobile app with Xamarin and Azure Cosmos DB. Follow similar steps to build Cordova apps by using the Azure Cosmos DB JavaScript SDK and native iOS/Android apps by using Azure Cosmos DB REST APIs.

Next steps

- View the source code for Xamarin and Azure Cosmos DB on GitHub.
- Download the Azure Cosmos DB .NET Core SDK.
- Find more code samples for .NET applications.
- Learn about Azure Cosmos DB rich query capabilities.
- Learn about geospatial support in Azure Cosmos DB.

Tutorial: Use Data migration tool to migrate your data to Azure Cosmos DB

12/4/2020 • 25 minutes to read • Edit Online

APPLIES TO: SQL API

This tutorial provides instructions on using the Azure Cosmos DB Data Migration tool, which can import data from various sources into Azure Cosmos containers and tables. You can import from JSON files, CSV files, SQL, MongoDB, Azure Table storage, Amazon DynamoDB, and even Azure Cosmos DB SQL API collections. You migrate that data to collections and tables for use with Azure Cosmos DB. The Data Migration tool can also be used when migrating from a single partition collection to a multi-partition collection for the SQL API.

> **NOTE**
> The Azure Cosmos DB Data Migration tool is an open source tool designed for small migrations. For larger migrations, view our guide for ingesting data.

- **SQL API** - You can use any of the source options provided in the Data Migration tool to import data at a small scale. Learn about migration options for importing data at a large scale.
- **Table API** - You can use the Data Migration tool or AzCopy to import data. For more information, see Import data for use with the Azure Cosmos DB Table API.
- **Azure Cosmos DB's API for MongoDB** - The Data Migration tool doesn't support Azure Cosmos DB's API for MongoDB either as a source or as a target. If you want to migrate the data in or out of collections in Azure Cosmos DB, refer to How to migrate MongoDB data to a Cosmos database with Azure Cosmos DB's API for MongoDB for instructions. You can still use the Data Migration tool to export data from MongoDB to Azure Cosmos DB SQL API collections for use with the SQL API.
- **Cassandra API** - The Data Migration tool isn't a supported import tool for Cassandra API accounts. Learn about migration options for importing data into Cassandra API
- **Gremlin API** - The Data Migration tool isn't a supported import tool for Gremlin API accounts at this time. Learn about migration options for importing data into Gremlin API

This tutorial covers the following tasks:

- Installing the Data Migration tool
- Importing data from different data sources
- Exporting from Azure Cosmos DB to JSON

Prerequisites

Before following the instructions in this article, ensure that you do the following steps:

- **Install** Microsoft .NET Framework 4.51 or higher.
- **Increase throughput:** The duration of your data migration depends on the amount of throughput you set up for an individual collection or a set of collections. Be sure to increase the throughput for larger data migrations. After you've completed the migration, decrease the throughput to save costs. For more information about increasing throughput in the Azure portal, see performance levels and pricing tiers in Azure Cosmos DB.
- **Create Azure Cosmos DB resources:** Before you start the migrating data, pre-create all your collections from the Azure portal. To migrate to an Azure Cosmos DB account that has database level throughput, provide a partition key when you create the Azure Cosmos containers.

> **IMPORTANT**
> To make sure that the Data migration tool uses Transport Layer Security (TLS) 1.2 when connecting to your Azure Cosmos accounts, use the .NET Framework version 4.7 or follow the instructions found in this article.

Overview

The Data Migration tool is an open-source solution that imports data to Azure Cosmos DB from a variety of sources, including:

- JSON files
- MongoDB
- SQL Server
- CSV files
- Azure Table storage
- Amazon DynamoDB
- HBase
- Azure Cosmos containers

While the import tool includes a graphical user interface (dtui.exe), it can also be driven from the command-line (dt.exe). In fact, there's an option to output the associated command after setting up an import through the UI. You can transform tabular source data, such as SQL Server or CSV files, to create hierarchical relationships (subdocuments) during import. Keep reading to learn more about source options, sample commands to import from each source, target options, and viewing import results.

> **NOTE**
> You should only use the Azure Cosmos DB migration tool for small migrations. For large migrations, view our guide for ingesting data.

Installation

The migration tool source code is available on GitHub in this repository. You can download and compile the solution locally, or download a pre-compiled binary, then run either:

- **Dtui.exe**: Graphical interface version of the tool
- **Dt.exe**: Command-line version of the tool

Select data source

Once you've installed the tool, it's time to import your data. What kind of data do you want to import?

- JSON files
- MongoDB
- MongoDB Export files
- SQL Server
- CSV files
- Azure Table storage
- Amazon DynamoDB
- Blob
- Azure Cosmos containers
- HBase
- Azure Cosmos DB bulk import
- Azure Cosmos DB sequential record import

Import JSON files

The JSON file source importer option allows you to import one or more single document JSON files or JSON files that each have an array of JSON documents. When adding folders that have JSON files to import, you have the option of recursively searching for files in subfolders.

The connection string is in the following format:

```
AccountEndpoint=<CosmosDB Endpoint>;AccountKey=<CosmosDB Key>;Database=<CosmosDB Database>
```

- The `<CosmosDB Endpoint>` is the endpoint URI. You can get this value from the Azure portal. Navigate to your Azure Cosmos account. Open the **Overview** pane and copy the **URI** value.
- The `<AccountKey>` is the "Password" or **PRIMARY KEY**. You can get this value from the Azure portal. Navigate to your Azure Cosmos account. Open the **Connection Strings** or **Keys** pane, and copy the "Password" or **PRIMARY KEY** value.
- The `<CosmosDB Database>` is the CosmosDB database name.

Example:

```
AccountEndpoint=https://myCosmosDBName.documents.azure.com:443/;AccountKey=wJmFRYna6ttQ79ATmrTMKql8vPri84QBiHTt6oinFkZRvoe7Vv81x9sn6zlVlBY10bEPMgGM982wfYXpWXWB9w==;Database=myDatabaseN
```

> **NOTE**
> Use the Verify command to ensure that the Cosmos DB account specified in the connection string field can be accessed.

Here are some command-line samples to import JSON files:

```
#Import a single JSON file
dt.exe /s:JsonFile /s.Files:.\Sessions.json /t:DocumentDBBulk /t.ConnectionString:"AccountEndpoint=<CosmosDB Endpoint>;AccountKey=<CosmosDB Key>;Database=<CosmosDB
Database>;" /t.Collection:Sessions /t.CollectionThroughput:2500

#Import a directory of JSON files
dt.exe /s:JsonFile /s.Files:C:\TESessions\*.json /t:DocumentDBBulk /t.ConnectionString:" AccountEndpoint=<CosmosDB Endpoint>;AccountKey=<CosmosDB Key>;Database=<CosmosDB
Database>;" /t.Collection:Sessions /t.CollectionThroughput:2500

#Import a directory (including sub-directories) of JSON files
dt.exe /s:JsonFile /s.Files:C:\LastFMMusic\**\*.json /t:DocumentDBBulk /t.ConnectionString:" AccountEndpoint=<CosmosDB Endpoint>;AccountKey=<CosmosDB Key>;Database=
<CosmosDB Database>;" /t.Collection:Music /t.CollectionThroughput:2500

#Import a directory (single), directory (recursive), and individual JSON files
dt.exe /s:JsonFile
/s.Files:C:\Tweets\*.*;C:\LargeDocs\**\*.*;C:\TESessions\Session48172.json;C:\TESessions\Session48173.json;C:\TESessions\Session48174.json;C:\TESessions\Session48175.jso
n;C:\TESessions\Session48177.json /t:DocumentDBBulk /t.ConnectionString:"AccountEndpoint=<CosmosDB Endpoint>;AccountKey=<CosmosDB Key>;Database=<CosmosDB Database>;"
/t.Collection:subs /t.CollectionThroughput:2500

#Import a single JSON file and partition the data across 4 collections
dt.exe /s:JsonFile /s.Files:D:\\CompanyData\\Companies.json /t:DocumentDBBulk /t.ConnectionString:"AccountEndpoint=<CosmosDB Endpoint>;AccountKey=<CosmosDB
Key>;Database=<CosmosDB Database>;" /t.Collection:comp[1-4] /t.PartitionKey:name /t.CollectionThroughput:2500
```

Import from MongoDB

> **IMPORTANT**
>
> If you're importing to a Cosmos account configured with Azure Cosmos DB's API for MongoDB, follow these instructions.

With the MongoDB source importer option, you can import from a single MongoDB collection, optionally filter documents using a query, and modify the document structure by using a projection.

The connection string is in the standard MongoDB format:

```
mongodb://<dbuser>:<dbpassword>@<host>:<port>/<database>
```

> **NOTE**
>
> Use the Verify command to ensure that the MongoDB instance specified in the connection string field can be accessed.

Enter the name of the collection from which data will be imported. You may optionally specify or provide a file for a query, such as `{pop: {$gt:5000}}`, or a projection, such as `{loc:0}`, to both filter and shape the data that you're importing.

Here are some command-line samples to import from MongoDB:

```
#Import all documents from a MongoDB collection
dt.exe /s:MongoDB /s.ConnectionString:mongodb://<dbuser>:<dbpassword>@<host>:<port>/<database> /s.Collection:zips /t:DocumentDBBulk /t.ConnectionString:"AccountEndpoint=
<CosmosDB Endpoint>;AccountKey=<CosmosDB Key>;Database=<CosmosDB Database>;" /t.Collection:BulkZips /t.IdField:_id /t.CollectionThroughput:2500

#Import documents from a MongoDB collection which match the query and exclude the loc field
dt.exe /s:MongoDB /s.ConnectionString:mongodb://<dbuser>:<dbpassword>@<host>:<port>/<database> /s.Collection:zips /s.Query:{pop:{$gt:50000}} /s.Projection:{loc:0}
/t:DocumentDBBulk /t.ConnectionString:"AccountEndpoint=<CosmosDB Endpoint>;AccountKey=<CosmosDB Key>;Database=<CosmosDB Database>;" /t.Collection:BulkZipsTransform
/t.IdField:_id/t.CollectionThroughput:2500
```

Import MongoDB export files

> **IMPORTANT**
>
> If you're importing to an Azure Cosmos DB account with support for MongoDB, follow these instructions.

The MongoDB export JSON file source importer option allows you to import one or more JSON files produced from the mongoexport utility.

When adding folders that have MongoDB export JSON files for import, you have the option of recursively searching for files in subfolders.

Here is a command-line sample to import from MongoDB export JSON files:

```
dt.exe /s:MongoDBExport /s.Files:D:\mongoemployees.json /t:DocumentDBBulk /t.ConnectionString:"AccountEndpoint=<CosmosDB Endpoint>;AccountKey=<CosmosDB Key>;Database=
<CosmosDB Database>;" /t.Collection:employees /t.IdField:_id /t.Dates:Epoch /t.CollectionThroughput:2500
```

Import from SQL Server

The SQL source importer option allows you to import from an individual SQL Server database and optionally filter the records to be imported using a query. In addition, you can modify the document structure by specifying a nesting separator (more on that in a moment).

The format of the connection string is the standard SQL connection string format.

> **NOTE**
> Use the Verify command to ensure that the SQL Server instance specified in the connection string field can be accessed.

The nesting separator property is used to create hierarchical relationships (sub-documents) during import. Consider the following SQL query:

```
select CAST(BusinessEntityID AS varchar) as Id, Name, AddressType as [Address.AddressType], AddressLine1 as [Address.AddressLine1], City as [Address.Location.City],
StateProvinceName as [Address.Location.StateProvinceName], PostalCode as [Address.PostalCode], CountryRegionName as [Address.CountryRegionName] from
Sales.vStoreWithAddresses WHERE AddressType='Main Office'
```

Which returns the following (partial) results:

	Id	Name	Address.AddressType	Address.AddressLine1	Address.Location.City	Address.Location.StateProvinceName	Address.PostalCode	Address.CountryRegionName
1	956	Finer Sales and Service	Main Office	#500-75 O'Connor Street	Ottawa	Ontario	K4B 1S2	Canada
2	780	Finer Riding Supplies	Main Office	#9900 2700 Production Way	Burnaby	British Columbia	V5A 4X1	Canada
3	1012	Stylish Department Stores	Main Office	1 Corporate Center Drive	Miami	Florida	33127	United States
4	482	Favorite Toy Distributor	Main Office	1, place de la République	Paris	Seine (Paris)	75017	France
5	1338	Sports Sales and Rental	Main Office	100 Fifth Drive	Millington	Tennessee	38054	United States
6	1424	Closeout Boutique	Main Office	1050 Oak Street	Seattle	Washington	98104	United States
7	1274	Self-Contained Cycle Parts Company	Main Office	12, rue des Grands Champs	Verrieres Le Buisson	Essonne	91370	France
8	1958	Ultimate Bicycle Company	Main Office	12, rue Lafayette	Morangis	Essonne	91420	France
9	1110	Local Sales and Rental	Main Office	1200 First Ave.	Joliet	Illinois	60433	United States
10	1262	Roadway Bicycle Supply	Main Office	121, rue de Varenne	Courbevoie	Hauts de Seine	92400	France

Note the aliases such as Address.AddressType and Address.Location.StateProvinceName. By specifying a nesting separator of '.', the import tool creates Address and Address.Location subdocuments during the import. Here is an example of a resulting document in Azure Cosmos DB:

{ "id": "956", "Name": "Finer Sales and Service", "Address": { "AddressType": "Main Office", "AddressLine1": "#500-75 O'Connor Street", "Location": { "City": "Ottawa", "StateProvinceName": "Ontario" }, "PostalCode": "K4B 1S2", "CountryRegionName": "Canada" } }

Here are some command-line samples to import from SQL Server:

```
#Import records from SQL which match a query
dt.exe /s:SQL /s.ConnectionString:"Data Source=<server>;Initial Catalog=AdventureWorks;User Id=advworks;Password=<password>;" /s.Query:"select CAST(BusinessEntityID AS
varchar) as Id, * from Sales.vStoreWithAddresses WHERE AddressType='Main Office'" /t:DocumentDBBulk /t.ConnectionString:" AccountEndpoint=<CosmosDB Endpoint>;AccountKey=
<CosmosDB Key>;Database=<CosmosDB Database>;" /t.Collection:Stores /t.IdField:Id /t.CollectionThroughput:2500

#Import records from sql which match a query and create hierarchical relationships
dt.exe /s:SQL /s.ConnectionString:"Data Source=<server>;Initial Catalog=AdventureWorks;User Id=advworks;Password=<password>;" /s.Query:"select CAST(BusinessEntityID AS
varchar) as Id, Name, AddressType as [Address.AddressType], AddressLine1 as [Address.AddressLine1], City as [Address.Location.City], StateProvinceName as
[Address.Location.StateProvinceName], PostalCode as [Address.PostalCode], CountryRegionName as [Address.CountryRegionName] from Sales.vStoreWithAddresses WHERE
AddressType='Main Office'" /s.NestingSeparator:. /t:DocumentDBBulk /t.ConnectionString:" AccountEndpoint=<CosmosDB Endpoint>;AccountKey=<CosmosDB Key>;Database=<CosmosDB
Database>;" /t.Collection:StoresSub /t.IdField:Id /t.CollectionThroughput:2500
```

Import CSV files and convert CSV to JSON

The CSV file source importer option enables you to import one or more CSV files. When adding folders that have CSV files for import, you have the option of recursively searching for files in subfolders.

Similar to the SQL source, the nesting separator property may be used to create hierarchical relationships (sub-documents) during import. Consider the following CSV header row and data rows:

```
DomainInfo.Domain_Name,DomainInfo.Domain_Name_Address,Federal Agency,RedirectInfo.Redirecting,RedirectInfo.Redirect_Destination
ACUS.GOV,http://www.ACUS.GOV,Administrative Conference of the United States,0,
ACHP.GOV,http://www.ACHP.GOV,Advisory Council on Historic Preservation,0,
PRESERVEAMERICA.GOV,http://www.PRESERVEAMERICA.GOV,Advisory Council on Historic Preservation,0,
ADF.GOV,http://www.ADF.GOV,African Development Foundation,0,
```

Note the aliases such as DomainInfo.Domain_Name and RedirectInfo.Redirecting. By specifying a nesting separator of '.', the import tool will create DomainInfo and RedirectInfo subdocuments during the import. Here is an example of a resulting document in Azure Cosmos DB:

{ "DomainInfo": { "Domain_Name": "ACUS.GOV", "Domain_Name_Address": "https://www.ACUS.GOV" }, "Federal Agency": "Administrative Conference of the United States", "RedirectInfo": { "Redirecting": "0", "Redirect_Destination": "" }, "id": "9cc565c5-ebcd-1c03-ebd3-cc3e2ecd814d" }

The import tool tries to infer type information for unquoted values in CSV files (quoted values are always treated as strings). Types are identified in the following order: number, datetime, boolean.

There are two other things to note about CSV import:

1. By default, unquoted values are always trimmed for tabs and spaces, while quoted values are preserved as-is. This behavior can be overridden with the Trim quoted values checkbox or the /s.TrimQuoted command-line option.
2. By default, an unquoted null is treated as a null value. This behavior can be overridden (that is, treat an unquoted null as a "null" string) with the Treat unquoted NULL as string checkbox or the /s.NoUnquotedNulls command-line option.

Here is a command-line sample for CSV import:

```
dt.exe /s:CsvFile /s.Files:.\Employees.csv /t:DocumentDBBulk /t.ConnectionString:"AccountEndpoint=<CosmosDB Endpoint>;AccountKey=<CosmosDB Key>;Database=<CosmosDB
Database>;" /t.Collection:Employees /t.IdField:EntityID /t.CollectionThroughput:2500
```

Import from Azure Table storage

The Azure Table storage source importer option allows you to import from an individual Azure Table storage table. Optionally, you can filter the table entities to be imported.

You may output data that was imported from Azure Table Storage to Azure Cosmos DB tables and entities for use with the Table API. Imported data can also be output to collections and documents for use with the SQL API. However, Table API is only available as a target in the command-line utility. You can't export to Table API by using the Data Migration tool user interface. For more information, see Import data for use with the Azure Cosmos DB Table API.

The format of the Azure Table storage connection string is:

```
DefaultEndpointsProtocol=<protocol>;AccountName=<Account Name>;AccountKey=<Account Key>;
```

> **NOTE**
> Use the Verify command to ensure that the Azure Table storage instance specified in the connection string field can be accessed.

Enter the name of the Azure table from to import from. You may optionally specify a filter.

The Azure Table storage source importer option has the following additional options:

1. Include Internal Fields
   a. All - Include all internal fields (PartitionKey, RowKey, and Timestamp)
   b. None - Exclude all internal fields
   c. RowKey - Only include the RowKey field
2. Select Columns
   a. Azure Table storage filters don't support projections. If you want to only import specific Azure Table entity properties, add them to the Select Columns list. All other entity properties are ignored.

Here is a command-line sample to import from Azure Table storage:

```
dt.exe /s:AzureTable /s.ConnectionString:"DefaultEndpointsProtocol=https;AccountName=<Account Name>;AccountKey=<Account Key>" /s.Table:metrics /s.InternalFields:All
/s.Filter:"PartitionKey eq 'Partition1' and RowKey gt '00001'" /s.Projection:ObjectCount;ObjectSize  /t:DocumentDBBulk /t.ConnectionString:" AccountEndpoint=<CosmosDB
Endpoint>;AccountKey=<CosmosDB Key>;Database=<CosmosDB Database>;" /t.Collection:metrics /t.CollectionThroughput:2500
```

Import from Amazon DynamoDB

The Amazon DynamoDB source importer option allows you to import from a single Amazon DynamoDB table. It can optionally filter the entities to be imported. Several templates are provided so that setting up an import is as easy as possible.

The format of the Amazon DynamoDB connection string is:

```
ServiceURL=<Service Address>;AccessKey=<Access Key>;SecretKey=<Secret Key>;
```

> **NOTE**
> Use the Verify command to ensure that the Amazon DynamoDB instance specified in the connection string field can be accessed.

Here is a command-line sample to import from Amazon DynamoDB:

```
dt.exe /s:DynamoDB /s.ConnectionString:ServiceURL=https://dynamodb.us-east-1.amazonaws.com;AccessKey=<accessKey>;SecretKey=<secretKey> /s.Request:"{   """TableName""":
"""ProductCatalog""" }" /t:DocumentDBBulk /t.ConnectionString:"AccountEndpoint=<Azure Cosmos DB Endpoint>;AccountKey=<Azure Cosmos DB Key>;Database=<Azure Cosmos
database>;" /t.Collection:catalogCollection /t.CollectionThroughput:2500
```

Import from Azure Blob storage

The JSON file, MongoDB export file, and CSV file source importer options allow you to import one or more files from Azure Blob storage. After specifying a Blob container URL and Account Key, provide a regular expression to select the file(s) to import.

Here is command-line sample to import JSON files from Azure Blob storage:

```
dt.exe /s:JsonFile /s.Files:"blobs://<account key>@account.blob.core.windows.net:443/importcontainer/.*" /t:DocumentDBBulk /t.ConnectionString:"AccountEndpoint=<CosmosDB
Endpoint>;AccountKey=<CosmosDB Key>;Database=<CosmosDB Database>;" /t.Collection:doctest
```

Import from a SQL API collection

The Azure Cosmos DB source importer option allows you to import data from one or more Azure Cosmos containers and optionally filter documents using a query.

The format of the Azure Cosmos DB connection string is:

```
AccountEndpoint=<CosmosDB Endpoint>;AccountKey=<CosmosDB Key>;Database=<CosmosDB Database>;
```

You can retrieve the Azure Cosmos DB account connection string from the Keys page of the Azure portal, as described in How to manage an Azure Cosmos DB account. However, the name of the database needs to be appended to the connection string in the following format:

```
Database=<CosmosDB Database>;
```

> **NOTE**
> Use the Verify command to ensure that the Azure Cosmos DB instance specified in the connection string field can be accessed.

To import from a single Azure Cosmos container, enter the name of the collection to import data from. To import from more than one Azure Cosmos container, provide a regular expression to match one or more collection names (for example, collection01 | collection02 | collection03). You may optionally specify, or provide a file for, a query to both filter and shape the data that you're importing.

> **NOTE**
> Since the collection field accepts regular expressions, if you're importing from a single collection whose name has regular expression characters, then those characters must be escaped accordingly.

The Azure Cosmos DB source importer option has the following advanced options:

1. Include Internal Fields: Specifies whether or not to include Azure Cosmos DB document system properties in the export (for example, _rid, _ts).
2. Number of Retries on Failure: Specifies the number of times to retry the connection to Azure Cosmos DB in case of transient failures (for example, network connectivity interruption).
3. Retry Interval: Specifies how long to wait between retrying the connection to Azure Cosmos DB in case of transient failures (for example, network connectivity interruption).
4. Connection Mode: Specifies the connection mode to use with Azure Cosmos DB. The available choices are DirectTcp, DirectHttps, and Gateway. The direct connection modes are faster, while the gateway mode is more firewall friendly as it only uses port 443.

TIP

The import tool defaults to connection mode DirectTcp. If you experience firewall issues, switch to connection mode Gateway, as it only requires port 443.

Here are some command-line samples to import from Azure Cosmos DB:

```
#Migrate data from one Azure Cosmos container to another Azure Cosmos containers
dt.exe /s:DocumentDB /s.ConnectionString:"AccountEndpoint=<CosmosDB Endpoint>;AccountKey=<CosmosDB Key>;Database=<CosmosDB Database>;" /s.Collection:TEColl
/t:DocumentDBBulk /t.ConnectionString:" AccountEndpoint=<CosmosDB Endpoint>;AccountKey=<CosmosDB Key>;Database=<CosmosDB Database>;" /t.Collection:TESessions
/t.CollectionThroughput:2500

#Migrate data from more than one Azure Cosmos container to a single Azure Cosmos container
dt.exe /s:DocumentDB /s.ConnectionString:"AccountEndpoint=<CosmosDB Endpoint>;AccountKey=<CosmosDB Key>;Database=<CosmosDB Database>;"
/s.Collection:comp1|comp2|comp3|comp4 /t:DocumentDBBulk /t.ConnectionString:"AccountEndpoint=<CosmosDB Endpoint>;AccountKey=<CosmosDB Key>;Database=<CosmosDB Database>;"
/t.Collection:singleCollection /t.CollectionThroughput:2500

#Export an Azure Cosmos container to a JSON file
dt.exe /s:DocumentDB /s.ConnectionString:"AccountEndpoint=<CosmosDB Endpoint>;AccountKey=<CosmosDB Key>;Database=<CosmosDB Database>;" /s.Collection:StoresSub
/t:JsonFile /t.File:StoresExport.json /t.Overwrite
```

TIP

The Azure Cosmos DB Data Import Tool also supports import of data from the Azure Cosmos DB Emulator. When importing data from a local emulator, set the endpoint to `https://localhost:<port>`.

Import from HBase

The HBase source importer option allows you to import data from an HBase table and optionally filter the data. Several templates are provided so that setting up an import is as easy as possible.

The format of the HBase Stargate connection string is:

```
ServiceURL=<server-address>;Username=<username>;Password=<password>
```

> **NOTE**
> Use the Verify command to ensure that the HBase instance specified in the connection string field can be accessed.

Here is a command-line sample to import from HBase:

```
dt.exe /s:HBase /s.ConnectionString:ServiceURL=<server-address>;Username=<username>;Password=<password> /s.Table:Contacts /t:DocumentDBBulk
/t.ConnectionString:"AccountEndpoint=<CosmosDB Endpoint>;AccountKey=<CosmosDB Key>;Database=<CosmosDB Database>;" /t.Collection:hbaseimport
```

Import to the SQL API (Bulk Import)

The Azure Cosmos DB Bulk importer allows you to import from any of the available source options, using an Azure Cosmos DB stored procedure for efficiency. The tool supports import to one single-partitioned Azure Cosmos container. It also supports sharded import whereby data is partitioned across more than one single-partitioned Azure Cosmos container. For more information about partitioning data, see Partitioning and scaling in Azure Cosmos DB. The tool creates, executes, and then deletes the stored procedure from the target collection(s).

The format of the Azure Cosmos DB connection string is:

```
AccountEndpoint=<CosmosDB Endpoint>;AccountKey=<CosmosDB Key>;Database=<CosmosDB Database>;
```

The Azure Cosmos DB account connection string can be retrieved from the Keys page of the Azure portal, as described in How to manage an Azure Cosmos DB account, however the name of the database needs to be appended to the connection string in the following format:

```
Database=<CosmosDB Database>;
```

> **NOTE**
> Use the Verify command to ensure that the Azure Cosmos DB instance specified in the connection string field can be accessed.

To import to a single collection, enter the name of the collection to import data from and click the Add button. To import to more than one collection, either enter each collection name individually or use the following syntax to specify more than one collection: *collection_prefix*[start index - end index]. When specifying more than one collection using the aforementioned syntax, keep the following guidelines in mind:

1. Only integer range name patterns are supported. For example, specifying collection[0-3] creates the following collections: collection0, collection1, collection2, collection3.
2. You can use an abbreviated syntax: collection[3] creates the same set of collections mentioned in step 1.
3. More than one substitution can be provided. For example, collection[0-1] [0-9] generates 20 collection names with leading zeros (collection01, ..02, ..03).

Once the collection name(s) have been specified, choose the desired throughput of the collection(s) (400 RUs to 10,000 RUs). For best import performance, choose a higher throughput. For more information about performance levels, see Performance levels in Azure Cosmos DB.

> **NOTE**
> The performance throughput setting only applies to collection creation. If the specified collection already exists, its throughput won't be modified.

When you import to more than one collection, the import tool supports hash-based sharding. In this scenario, specify the document property you wish to use as the Partition Key. (If Partition Key is left blank, documents are sharded randomly across the target collections.)

You may optionally specify which field in the import source should be used as the Azure Cosmos DB document ID property during the import. If documents don't have this property, then the import tool generates a GUID as the ID property value.

There are a number of advanced options available during import. First, while the tool includes a default bulk import stored procedure (BulkInsert.js), you may choose to specify your own import stored procedure:

Additionally, when importing date types (for example, from SQL Server or MongoDB), you can choose between three import options:

- String: Persist as a string value
- Epoch: Persist as an Epoch number value
- Both: Persist both string and Epoch number values. This option creates a subdocument, for example: "date_joined": { "Value": "2013-10-21T21:17:25.2410000Z", "Epoch": 1382390245 }

The Azure Cosmos DB Bulk importer has the following additional advanced options:

1. Batch Size: The tool defaults to a batch size of 50. If the documents to be imported are large, consider lowering the batch size. Conversely, if the documents to be imported are small, consider raising the batch size.
2. Max Script Size (bytes): The tool defaults to a max script size of 512 KB.
3. Disable Automatic Id Generation: If every document to be imported has an ID field, then selecting this option can increase performance. Documents missing a unique ID

field aren't imported.

4. Update Existing Documents: The tool defaults to not replacing existing documents with ID conflicts. Selecting this option allows overwriting existing documents with matching IDs. This feature is useful for scheduled data migrations that update existing documents.
5. Number of Retries on Failure: Specifies how often to retry the connection to Azure Cosmos DB during transient failures (for example, network connectivity interruption).
6. Retry Interval: Specifies how long to wait between retrying the connection to Azure Cosmos DB in case of transient failures (for example, network connectivity interruption).
7. Connection Mode: Specifies the connection mode to use with Azure Cosmos DB. The available choices are DirectTcp, DirectHttps, and Gateway. The direct connection modes are faster, while the gateway mode is more firewall friendly as it only uses port 443.

TIP

The import tool defaults to connection mode DirectTcp. If you experience firewall issues, switch to connection mode Gateway, as it only requires port 443.

Import to the SQL API (Sequential Record Import)

The Azure Cosmos DB sequential record importer allows you to import from an available source option on a record-by-record basis. You might choose this option if you're importing to an existing collection that has reached its quota of stored procedures. The tool supports import to a single (both single-partition and multi-partition) Azure Cosmos container. It also supports sharded import whereby data is partitioned across more than one single-partition or multi-partition Azure Cosmos container. For more information about partitioning data, see Partitioning and scaling in Azure Cosmos DB.

The format of the Azure Cosmos DB connection string is:

```
AccountEndpoint=<CosmosDB Endpoint>;AccountKey=<CosmosDB Key>;Database=<CosmosDB Database>;
```

You can retrieve the connection string for the Azure Cosmos DB account from the Keys page of the Azure portal, as described in How to manage an Azure Cosmos DB account. However, the name of the database needs to be appended to the connection string in the following format:

```
Database=<Azure Cosmos database>;
```

> **NOTE**
> Use the Verify command to ensure that the Azure Cosmos DB instance specified in the connection string field can be accessed.

To import to a single collection, enter the name of the collection to import data into, and then click the Add button. To import to more than one collection, enter each collection name individually. You may also use the following syntax to specify more than one collection: *collection_prefix*[start index - end index]. When specifying more than one collection via the aforementioned syntax, keep the following guidelines in mind:

1. Only integer range name patterns are supported. For example, specifying collection[0-3] creates the following collections: collection0, collection1, collection2, collection3.
2. You can use an abbreviated syntax: collection[3] creates the same set of collections mentioned in step 1.
3. More than one substitution can be provided. For example, collection[0-1] [0-9] creates 20 collection names with leading zeros (collection01, ..02, ..03).

Once the collection name(s) have been specified, choose the desired throughput of the collection(s) (400 RUs to 250,000 RUs). For best import performance, choose a higher throughput. For more information about performance levels, see Performance levels in Azure Cosmos DB. Any import to collections with throughput >10,000 RUs require a partition key. If you choose to have more than 250,000 RUs, you need to file a request in the portal to have your account increased.

> **NOTE**
> The throughput setting only applies to collection or database creation. If the specified collection already exists, its throughput won't be modified.

When importing to more than one collection, the import tool supports hash-based sharding. In this scenario, specify the document property you wish to use as the Partition Key. (If Partition Key is left blank, documents are sharded randomly across the target collections.)

You may optionally specify which field in the import source should be used as the Azure Cosmos DB document ID property during the import. (If documents don't have this property, then the import tool generates a GUID as the ID property value.)

There are a number of advanced options available during import. First, when importing date types (for example, from SQL Server or MongoDB), you can choose between three import options:

- String: Persist as a string value
- Epoch: Persist as an Epoch number value
- Both: Persist both string and Epoch number values. This option creates a subdocument, for example: "date_joined": { "Value": "2013-10-21T21:17:25.2410000Z", "Epoch": 1382390245 }

The Azure Cosmos DB - Sequential record importer has the following additional advanced options:

1. Number of Parallel Requests: The tool defaults to two parallel requests. If the documents to be imported are small, consider raising the number of parallel requests. If this number is raised too much, the import may experience rate limiting.
2. Disable Automatic Id Generation: If every document to be imported has an ID field, then selecting this option can increase performance. Documents missing a unique ID field aren't imported.
3. Update Existing Documents: The tool defaults to not replacing existing documents with ID conflicts. Selecting this option allows overwriting existing documents with matching IDs. This feature is useful for scheduled data migrations that update existing documents.
4. Number of Retries on Failure: Specifies how often to retry the connection to Azure Cosmos DB during transient failures (for example, network connectivity interruption).
5. Retry Interval: Specifies how long to wait between retrying the connection to Azure Cosmos DB during transient failures (for example, network connectivity interruption).
6. Connection Mode: Specifies the connection mode to use with Azure Cosmos DB. The available choices are DirectTcp, DirectHttps, and Gateway. The direct connection modes are faster, while the gateway mode is more firewall friendly as it only uses port 443.

TIP

The import tool defaults to connection mode DirectTcp. If you experience firewall issues, switch to connection mode Gateway, as it only requires port 443.

Specify an indexing policy

When you allow the migration tool to create Azure Cosmos DB SQL API collections during import, you can specify the indexing policy of the collections. In the advanced options section of the Azure Cosmos DB Bulk import and Azure Cosmos DB Sequential record options, navigate to the Indexing Policy section.

Using the Indexing Policy advanced option, you can select an indexing policy file, manually enter an indexing policy, or select from a set of default templates (by right-clicking in the indexing policy textbox).

The policy templates the tool provides are:

- Default. This policy is best when you perform equality queries against strings. It also works if you use ORDER BY, range, and equality queries for numbers. This policy has a lower index storage overhead than Range.
- Range. This policy is best when you use ORDER BY, range, and equality queries on both numbers and strings. This policy has a higher index storage overhead than Default or Hash.

> **NOTE**
> If you don't specify an indexing policy, then the default policy is applied. For more information about indexing policies, see Azure Cosmos DB indexing policies.

Export to JSON file

The Azure Cosmos DB JSON exporter allows you to export any of the available source options to a JSON file that has an array of JSON documents. The tool handles the export for you. Alternatively, you can choose to view the resulting migration command and run the command yourself. The resulting JSON file may be stored locally or in Azure Blob storage.

You may optionally choose to prettify the resulting JSON. This action will increase the size of the resulting document while making the contents more human readable.

- Standard JSON export

```
[{"id":"Sample","Title":"About Paris","Language":{"Name":"English"},"Author":{"Name":"Don","Location":{"City":"Paris","Country":"France"}},"Content":"Don's
document in Azure Cosmos DB is a valid JSON document as defined by the JSON spec.","PageViews":10000,"Topics":[{"Title":"History of Paris"},{"Title":"Places to
see in Paris"}]}]
```

- Prettified JSON export

```
[
 {
"id": "Sample",
"Title": "About Paris",
"Language": {
  "Name": "English"
},
"Author": {
  "Name": "Don",
  "Location": {
    "City": "Paris",
    "Country": "France"
  }
},
"Content": "Don's document in Azure Cosmos DB is a valid JSON document as defined by the JSON spec.",
"PageViews": 10000,
"Topics": [
  {
    "Title": "History of Paris"
  },
  {
    "Title": "Places to see in Paris"
  }
]
}]
```

Here is a command-line sample to export the JSON file to Azure Blob storage:

```
dt.exe /ErrorDetails:All /s:DocumentDB /s.ConnectionString:"AccountEndpoint=<CosmosDB Endpoint>;AccountKey=<CosmosDB Key>;Database=<CosmosDB database_name>"
/s.Collection:<CosmosDB collection_name>
/t:JsonFile /t.File:"blobs://<Storage account key>@<Storage account name>.blob.core.windows.net:443/<Container_name>/<Blob_name>"
/t.Overwrite
```

Advanced configuration

In the Advanced configuration screen, specify the location of the log file to which you would like any errors written. The following rules apply to this page:

1. If a file name isn't provided, then all errors are returned on the Results page.
2. If a file name is provided without a directory, then the file is created (or overwritten) in the current environment directory.
3. If you select an existing file, then the file is overwritten, there's no append option.
4. Then, choose whether to log all, critical, or no error messages. Finally, decide how frequently the on-screen transfer message is updated with its progress.

Confirm import settings and view command line

1. After you specify the source information, target information, and advanced configuration, review the migration summary and view or copy the resulting migration command if you want. (Copying the command is useful to automate import operations.)

2. Once you're satisfied with your source and target options, click **Import**. The elapsed time, transferred count, and failure information (if you didn't provide a file name in the Advanced configuration) update as the import is in process. Once complete, you can export the results (for example, to deal with any import failures).

3. You may also start a new import by either resetting all values or keeping the existing settings. (For example, you may choose to keep connection string information, source and target choice, and more.)

Next steps

In this tutorial, you've done the following tasks:

- Installed the Data Migration tool
- Imported data from different data sources
- Exported from Azure Cosmos DB to JSON

You can now proceed to the next tutorial and learn how to query data using Azure Cosmos DB.

How to query data?

Bulk import data to Azure Cosmos DB SQL API account by using the .NET SDK

12/4/2020 • 6 minutes to read • Edit Online

APPLIES TO: SQL API

This tutorial shows how to build a .NET console application that optimizes provisioned throughput (RU/s) required to import data to Azure Cosmos DB. In this article, you will read data from a sample data source and import it into an Azure Cosmos container. This tutorial uses Version 3.0+ of the Azure Cosmos DB .NET SDK, which can be targeted to .NET Framework or .NET Core.

This tutorial covers:

- Creating an Azure Cosmos account
- Configuring your project
- Connecting to an Azure Cosmos account with bulk support enabled
- Perform a data import through concurrent create operations

Prerequisites

Before following the instructions in this article, make sure that you have the following resources:

- An active Azure account. If you don't have an Azure subscription, create a free account before you begin.

  You can Try Azure Cosmos DB for free without an Azure subscription, free of charge and commitments, or create an Azure Cosmos DB free tier account, with the first 400 RU/s and 5 GB of storage for free. You can also use the Azure Cosmos DB Emulator with a URI of `https://localhost:8081`. For the key to use with the emulator, see Authenticating requests.

- NET Core 3 SDK. You can verify which version is available in your environment by running `dotnet --version`.

Step 1: Create an Azure Cosmos DB account

Create an Azure Cosmos DB SQL API account from the Azure portal or you can create the account by using the Azure Cosmos DB Emulator.

Step 2: Set up your .NET project

Open the Windows command prompt or a Terminal window from your local computer. You will run all the commands in the next sections from the command prompt or terminal. Run the following dotnet new command to create a new app with the name *bulk-import-demo*. The `--langVersion` parameter sets the *LangVersion* property in the created project file.

```
dotnet new console –langVersion:8 -n bulk-import-demo
```

Change your directory to the newly created app folder. You can build the application with:

```
cd bulk-import-demo
dotnet build
```

The expected output from the build should look something like this:

```
Restore completed in 100.37 ms for C:\Users\user1\Downloads\CosmosDB_Samples\bulk-import-demo\bulk-import-demo.csproj.
  bulk -> C:\Users\user1\Downloads\CosmosDB_Samples\bulk-import-demo \bin\Debug\netcoreapp2.2\bulk-import-demo.dll

Build succeeded.
    0 Warning(s)
    0 Error(s)

Time Elapsed 00:00:34.17
```

Step 3: Add the Azure Cosmos DB package

While still in the application directory, install the Azure Cosmos DB client library for .NET Core by using the dotnet add package command.

```
dotnet add package Microsoft.Azure.Cosmos
```

Step 4: Get your Azure Cosmos account credentials

The sample application needs to authenticate to your Azure Cosmos account. To authenticate, you should pass the Azure Cosmos account credentials to the application. Get your Azure Cosmos account credentials by following these steps:

1. Sign in to the Azure portal.
2. Navigate to your Azure Cosmos account.
3. Open the **Keys** pane and copy the **URI** and **PRIMARY KEY** of your account.

If you are using the Azure Cosmos DB Emulator, obtain the emulator credentials from this article.

Step 5: Initialize the CosmosClient object with bulk execution support

Open the generated `Program.cs` file in a code editor. You will create a new instance of CosmosClient with bulk execution enabled and use it to do operations against Azure Cosmos DB.

Let's start by overwriting the default `Main` method and defining the global variables. These global variables will include the endpoint and authorization keys, the name of the database, container that you will create, and the number of items that you will be inserting in bulk. Make sure to replace the endpointURL and authorization key values according to your environment.

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.IO;
using System.Text.Json;
using System.Threading.Tasks;
using Microsoft.Azure.Cosmos;

public class Program
{
     private const string EndpointUrl = "https://<your-account>.documents.azure.com:443/";
     private const string AuthorizationKey = "<your-account-key>";
     private const string DatabaseName = "bulk-tutorial";
     private const string ContainerName = "items";
     private const int ItemsToInsert = 300000;

     static async Task Main(string[] args)
     {

     }
}
```

Inside the `Main` method, add the following code to initialize the CosmosClient object:

```
CosmosClient cosmosClient = new CosmosClient(EndpointUrl, AuthorizationKey, new CosmosClientOptions() {
AllowBulkExecution = true });
```

After the bulk execution is enabled, the CosmosClient internally groups concurrent operations into single service calls. This way it optimizes the throughput utilization by distributing service calls across partitions, and finally assigning individual results to the original callers.

You can then create a container to store all our items. Define `/pk` as the partition key, 50000 RU/s as provisioned throughput, and a custom indexing policy that will exclude all fields to optimize the write throughput. Add the following code after the CosmosClient initialization statement:

```
Database database = await cosmosClient.CreateDatabaseIfNotExistsAsync(Program.DatabaseName);

await database.DefineContainer(Program.ContainerName, "/pk")
        .WithIndexingPolicy()
            .WithIndexingMode(IndexingMode.Consistent)
            .WithIncludedPaths()
                .Attach()
            .WithExcludedPaths()
                .Path("/*")
                .Attach()
        .Attach()
    .CreateAsync(50000);
```

Step 6: Populate a list of concurrent tasks

To take advantage of the bulk execution support, create a list of asynchronous tasks based on the source of data and the operations you want to perform, and use `Task.WhenAll` to execute them concurrently. Let's start by using "Bogus" data to generate a list of items from our data model. In a real-world application, the items would come from your desired data source.

First, add the Bogus package to the solution by using the dotnet add package command.

```
dotnet add package Bogus
```

Define the definition of the items that you want to save. You need to define the `Item` class within the `Program.cs` file:

```
public class Item
{
    public string id {get;set;}
    public string pk {get;set;}

    public string username{get;set;}
}
```

Next, create a helper function inside the `Program` class. This helper function will get the number of items you defined to insert and generates random data:

```
private static IReadOnlyCollection<Item> GetItemsToInsert()
{
    return new Bogus.Faker<Item>()
    .StrictMode(true)
    //Generate item
    .RuleFor(o => o.id, f => Guid.NewGuid().ToString()) //id
    .RuleFor(o => o.username, f => f.Internet.UserName())
    .RuleFor(o => o.pk, (f, o) => o.id) //partitionkey
    .Generate(ItemsToInsert);
}
```

Read the items and serialize them into stream instances by using the `System.Text.Json` class. Because of the nature of the autogenerated data, you are serializing the data as streams. You can also use the item instance directly, but by converting them to streams, you can leverage the performance of stream APIs in the CosmosClient. Typically you can use the data directly as long as you know the partition key.

To convert the data to stream instances, within the `Main` method, add the following code right after creating the container:

```
Dictionary<PartitionKey, Stream> itemsToInsert = new Dictionary<PartitionKey, Stream>(ItemsToInsert);
foreach (Item item in Program.GetItemsToInsert())
{
    MemoryStream stream = new MemoryStream();
    await JsonSerializer.SerializeAsync(stream, item);
    itemsToInsert.Add(new PartitionKey(item.pk), stream);
}
```

Next use the data streams to create concurrent tasks and populate the task list to insert the items into the container. To perform this operation, add the following code to the `Program` class:

```
Container container = database.GetContainer(ContainerName);
List<Task> tasks = new List<Task>(ItemsToInsert);
foreach (KeyValuePair<PartitionKey, Stream> item in itemsToInsert)
{
    tasks.Add(container.CreateItemStreamAsync(item.Value, item.Key)
        .ContinueWith((Task<ResponseMessage> task) =>
        {
            using (ResponseMessage response = task.Result)
            {
                if (!response.IsSuccessStatusCode)
                {
                    Console.WriteLine($"Received {response.StatusCode} ({response.ErrorMessage}).");
                }
            }
        }));
}

// Wait until all are done
await Task.WhenAll(tasks);
```

All these concurrent point operations will be executed together (that is in bulk) as described in the introduction section.

Step 7: Run the sample

In order to run the sample, you can do it simply by the `dotnet` command:

```
dotnet run
```

Get the complete sample

If you didn't have time to complete the steps in this tutorial, or just want to download the code samples, you can get it from GitHub.

After cloning the project, make sure to update the desired credentials inside Program.cs.

The sample can be run by changing to the repository directory and using `dotnet`:

```
cd cosmos-dotnet-bulk-import-throughput-optimizer
dotnet run
```

Next steps

In this tutorial, you've done the following steps:

- Creating an Azure Cosmos account
- Configuring your project
- Connecting to an Azure Cosmos account with bulk support enabled
- Perform a data import through concurrent create operations

You can now proceed to the next tutorial:

Query Azure Cosmos DB by using the SQL API

Tutorial: Query Azure Cosmos DB by using the SQL API

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: SQL API

The Azure Cosmos DB SQL API supports querying documents using SQL. This article provides a sample document and two sample SQL queries and results.

This article covers the following tasks:

- Querying data with SQL

Sample document

The SQL queries in this article use the following sample document.

```
{
  "id": "WakefieldFamily",
  "parents": [
      { "familyName": "Wakefield", "givenName": "Robin" },
      { "familyName": "Miller", "givenName": "Ben" }
  ],
  "children": [
      {
        "familyName": "Merriam",
        "givenName": "Jesse",
        "gender": "female", "grade": 1,
        "pets": [
            { "givenName": "Goofy" },
            { "givenName": "Shadow" }
        ]
      },
      {
        "familyName": "Miller",
         "givenName": "Lisa",
         "gender": "female",
         "grade": 8 }
  ],
  "address": { "state": "NY", "county": "Manhattan", "city": "NY" },
  "creationDate": 1431620462,
  "isRegistered": false
}
```

Where can I run SQL queries?

You can run queries using the Data Explorer in the Azure portal, via the REST API and SDKs, and even the Query playground, which runs queries on an existing set of sample data.

For more information about SQL queries, see:

- SQL query and SQL syntax

Prerequisites

This tutorial assumes you have an Azure Cosmos DB account and collection. Don't have any of those resources? Complete the 5-minute quickstart.

Example query 1

Given the sample family document above, following SQL query returns the documents where the ID field matches `WakefieldFamily`. Since it's a `SELECT *` statement, the output of the query is the complete JSON document:

Query

```
SELECT *
FROM Families f
WHERE f.id = "WakefieldFamily"
```

Results

```
{
  "id": "WakefieldFamily",
  "parents": [
      { "familyName": "Wakefield", "givenName": "Robin" },
      { "familyName": "Miller", "givenName": "Ben" }
  ],
  "children": [
      {
        "familyName": "Merriam",
        "givenName": "Jesse",
        "gender": "female", "grade": 1,
        "pets": [
            { "givenName": "Goofy" },
            { "givenName": "Shadow" }
        ]
      },
      {
        "familyName": "Miller",
         "givenName": "Lisa",
         "gender": "female",
         "grade": 8 }
  ],
  "address": { "state": "NY", "county": "Manhattan", "city": "NY" },
  "creationDate": 1431620462,
  "isRegistered": false
}
```

Example query 2

The next query returns all the given names of children in the family whose ID matches `WakefieldFamily`.

Query

```
SELECT c.givenName
FROM Families f
JOIN c IN f.children
WHERE f.id = 'WakefieldFamily'
```

Results

```
[
    {
        "givenName": "Jesse"
    },
    {
        "givenName": "Lisa"
    }
]
```

Next steps

In this tutorial, you've done the following tasks:

- Learned how to query using SQL

You can now proceed to the next tutorial to learn how to distribute your data globally.

Distribute your data globally

Tutorial: Create a notebook in Azure Cosmos DB to analyze and visualize the data

12/4/2020 • 6 minutes to read • Edit Online

APPLIES TO: SQL API

This article describes how to use built-in Jupyter notebooks to import sample retail data to Azure Cosmos DB. You will see how to use the SQL and Azure Cosmos DB magic commands to run queries, analyze the data, and visualize the results.

Prerequisites

- Enable notebooks support while creating the Azure Cosmos account

Create the resources and import data

In this section, you will create the Azure Cosmos database, container, and import the retail data to the container.

1. Navigate to your Azure Cosmos account and open the **Data Explorer.**
2. Go to the **Notebooks** tab, select ... next to **My Notebooks** and create a **New Notebook**. Select **Python 3** as the default Kernel.

3. After a new notebook is created, you can rename it to something like **VisualizeRetailData.ipynb.**
4. Next you will create a database named "RetailDemo" and a container named "WebsiteData" to store the retail data. You can use /CartID as the partition key. Copy and paste the following code to a new cell in your notebook and run it:

```
import azure.cosmos
from azure.cosmos.partition_key import PartitionKey

database = cosmos_client.create_database_if_not_exists('RetailDemo')
print('Database RetailDemo created')

container = database.create_container_if_not_exists(id='WebsiteData',
partition_key=PartitionKey(path='/CartID'))
print('Container WebsiteData created')
```

To run a cell, select `Shift + Enter` Or select the cell and choose **Run Active Cell** option at the data explorer navigation bar.

The database and container are created in your current Azure Cosmos account. The container is provisioned with 400 RU/s. You will see the following output after the database and container is created.

```
Database RetailDemo created
Container WebsiteData created
```

You can also refresh the **Data** tab and see the newly created resources:

5. Next you will import the sample retail data into Azure Cosmos container. Here is the format of an item from the retail data:

```
{
   "CartID":5399,
   "Action":"Viewed",
   "Item":"Cosmos T-shirt",
   "Price":350,
   "UserName":"Demo.User10",
   "Country":"Iceland",
   "EventDate":"2015-06-25T00:00:00",
   "Year":2015,"Latitude":-66.8673,
   "Longitude":-29.8214,
   "Address":"852 Modesto Loop, Port Ola, Iceland",
   "id":"00ffd39c-7e98-4451-9b91-b2bcf2f9a32d"
}
```

For the tutorial purpose, the sample retail data is stored in the Azure blob storage. You can import it to the Azure Cosmos container by pasting the following code into a new cell. You can confirm that the data is successfully imported by running a query to select the number of items.

```
# Read data from storage
import urllib.request, json

with
urllib.request.urlopen("https://cosmosnotebooksdata.blob.core.windows.net/notebookdata/websiteData.json"
) as url:
   data = json.loads(url.read().decode())

print("Importing data. This will take a few minutes...\n")

for event in data:
 try:
    container.upsert_item(body=event)
 except errors.CosmosHttpResponseError as e:
    raise

## Run a query against the container to see number of documents
query = 'SELECT VALUE COUNT(1) FROM c'
result = list(container.query_items(query, enable_cross_partition_query=True))

print('Container with id \'{0}\' contains \'{1}\' items'.format(container.id, result[0]))
```

When you run the previous query, it returns the following output:

```
Importing data. This will take a few minutes...

Container with id 'WebsiteData' contains '2654' items
```

Get your data into a DataFrame

Before running queries to analyze the data, you can read the data from container to a Pandas DataFrame for analysis. Use the following sql magic command to read the data into a DataFrame:

```
%%sql --database {database_id} --container {container_id} --output outputDataframeVar
{Query text}
```

To learn more, see the built-in notebook commands and features in Azure Cosmos DB article. You will run the query- `SELECT c.Action, c.Price as ItemRevenue, c.Country, c.Item FROM c`. The results will be saved into a Pandas DataFrame named df_cosmos. Paste the following command in a new notebook cell and run it:

```
%%sql --database RetailDemo --container WebsiteData --output df_cosmos
SELECT c.Action, c.Price as ItemRevenue, c.Country, c.Item FROM c
```

In a new notebook cell, run the following code to read the first 10 items from the output:

```
# See a sample of the result
df_cosmos.head(10)
```

	Action	ItemRevenue	Country	Item
0	Viewed	9.00	Tunisia	Black Tee
1	Viewed	19.99	Antigua and Barbuda	Flannel Shirt
2	Added	3.75	Guinea-Bissau	Socks
3	Viewed	3.75	Guinea-Bissau	Socks
4	Viewed	55.00	Czech Republic	Rainjacket
5	Viewed	350.00	Iceland	Cosmos T-shirt
6	Added	19.99	Syrian Arab Republic	Button-Up Shirt
7	Viewed	19.99	Syrian Arab Republic	Button-Up Shirt
8	Viewed	33.00	Tuvalu	Red Top
9	Viewed	14.00	Cape Verde	Flip Flop Shoes

Run queries and analyze your data

In this section, you will run some queries on the data retrieved.

- **Query1**: Run a Group by query on the DataFrame to get the sum of total sales revenue for each country/region and display 5 items from the results. In a new notebook cell, run the following code:

  ```
  df_revenue = df_cosmos.groupby("Country").sum().reset_index()
  display(df_revenue.head(5))
  ```

	Country	ItemRevenue
0	Afghanistan	785.44
1	Albania	605.80
2	Algeria	1058.98
3	American Samoa	229.00
4	Andorra	247.49

- **Query2**: To get a list of top five purchased items, open a new notebook cell and run the following code:

  ```
  import pandas as pd

  ## What are the top 5 purchased items?
  pd.DataFrame(df_cosmos[df_cosmos['Action']=='Purchased'].groupby('Item').size().sort_values(ascending=Fa
  lse).head(5), columns=['Count'])
  ```

[21]:

Item	Count
Puffy Jacket	25
Athletic Shoes	20
Athletic Shorts	19
Crewneck Sweater	14
Light Jeans	13

Visualize your data

1. Now that we have our data on revenue from the Azure Cosmos container, you can visualize your data with a visualization library of your choice. In this tutorial, we will use Bokeh library. Open a new notebook cell and run the following code to install the Bokeh library. After all the requirements are satisfied, the library will be installed.

```
import sys
!{sys.executable} -m pip install bokeh --user
```

2. Next prepare to plot the data on a map. Join the data in Azure Cosmos DB with country/region information located in Azure Blob storage and convert the result to GeoJSON format. Copy the following code to a new notebook cell and run it.

```
import urllib.request, json
import geopandas as gpd

# Load country/region information for mapping
countries =
gpd.read_file("https://cosmosnotebooksdata.blob.core.windows.net/notebookdata/countries.json")

# Merge the countries/regions dataframe with our data in Azure Cosmos DB, joining on country/region code
df_merged = countries.merge(df_revenue, left_on = 'admin', right_on = 'Country', how='left')

# Convert to GeoJSON so bokeh can plot it
merged_json = json.loads(df_merged.to_json())
json_data = json.dumps(merged_json)
```

3. Visualize the sales revenue of different countries/regions on a world map by running the following code in a new notebook cell:

```
from bokeh.io import output_notebook, show
from bokeh.plotting import figure
from bokeh.models import GeoJSONDataSource, LinearColorMapper, ColorBar
from bokeh.palettes import brewer

#Input GeoJSON source that contains features for plotting.
geosource = GeoJSONDataSource(geojson = json_data)

#Choose our choropleth color palette: https://bokeh.pydata.org/en/latest/docs/reference/palettes.html
palette = brewer['YlGn'][8]

#Reverse color order so that dark green is highest revenue
palette = palette[::-1]

#Instantiate LinearColorMapper that linearly maps numbers in a range, into a sequence of colors.
color_mapper = LinearColorMapper(palette = palette, low = 0, high = 1000)

#Define custom tick labels for color bar.
tick_labels = {'0': '$0', '250': '$250', '500':'$500', '750':'$750', '1000':'$1000', '1250':'$1250',
'1500':'$1500','1750':'$1750', '2000': '>$2000'}

#Create color bar.
color_bar = ColorBar(color_mapper=color_mapper, label_standoff=8,width = 500, height = 20,
border_line_color=None,location = (0,0), orientation = 'horizontal', major_label_overrides =
tick_labels)

#Create figure object.
p = figure(title = 'Sales revenue by country', plot_height = 600 , plot_width = 1150, toolbar_location =
None)
p.xgrid.grid_line_color = None
p.ygrid.grid_line_color = None

#Add patch renderer to figure.
p.patches('xs','ys', source = geosource,fill_color = {'field' :'ItemRevenue', 'transform' :
color_mapper},
           line_color = 'black', line_width = 0.25, fill_alpha = 1)

#Specify figure layout.
p.add_layout(color_bar, 'below')

#Display figure inline in Jupyter Notebook.
output_notebook()

#Display figure.
show(p)
```

The output displays the world map with different colors. The colors darker to lighter represent the countries/regions with highest revenue to lowest revenue.

4. Let's see another case of data visualization. The WebsiteData container has record of users who viewed an item, added to their cart, and purchased the item. Let's plot the conversion rate of items purchased. Run the following code in a new cell to visualize the conversion rate for each item:

```
from bokeh.io import show, output_notebook
from bokeh.plotting import figure
from bokeh.palettes import Spectral3
from bokeh.transform import factor_cmap
from bokeh.models import ColumnDataSource, FactorRange

# Get the top 10 items as an array
top_10_items =
df_cosmos[df_cosmos['Action']=='Purchased'].groupby('Item').size().sort_values(ascending=False)
[:10].index.values.tolist()

# Filter our data to only these 10 items
df_top10 = df_cosmos[df_cosmos['Item'].isin(top_10_items)]

# Group by Item and Action, sorting by event count
df_top10_sorted = df_top10.groupby(['Item', 'Action']).count().rename(columns={'Country':'ResultCount'},
inplace=False).reset_index().sort_values(['Item', 'ResultCount'], ascending = False).set_index(['Item',
'Action'])

# Get sorted X-axis values - this way, we can display the funnel of view -> add -> purchase
x_axis_values = df_top10_sorted.index.values.tolist()

group = df_top10_sorted.groupby(['Item', 'Action'])

# Specifiy colors for X axis
index_cmap = factor_cmap('Item_Action', palette=Spectral3, factors=sorted(df_top10.Action.unique()),
start=1, end=2)

# Create the plot

p = figure(plot_width=1200, plot_height=500, title="Conversion rate of items from View -> Add to cart ->
Purchase", x_range=FactorRange(*x_axis_values), toolbar_location=None, tooltips=[("Number of events",
"@ResultCount_max"), ("Item, Action", "@Item_Action")])

p.vbar(x='Item_Action', top='ItemRevenue_max', width=1, source=group,
        line_color="white", fill_color=index_cmap, )

#Configure how the plot looks
p.y_range.start = 0
p.x_range.range_padding = 0.05
p.xgrid.grid_line_color = None
p.xaxis.major_label_orientation = 1.2
p.outline_line_color = "black"
p.xaxis.axis_label = "Item"
p.yaxis.axis_label = "Count"

#Display figure inline in Jupyter Notebook.
output_notebook()

#Display figure.
show(p)
```

Next steps

- To learn more about Python notebook commands, see how to use built-in notebook commands and features in Azure Cosmos DB article.

Tutorial: Set up Azure Cosmos DB global distribution using the SQL API

12/4/2020 • 7 minutes to read • Edit Online

APPLIES TO: SQL API

In this article, we show how to use the Azure portal to set up Azure Cosmos DB global distribution and then connect using the SQL API.

This article covers the following tasks:

- Configure global distribution using the Azure portal
- Configure global distribution using the SQL APIs

Add global database regions using the Azure portal

Azure Cosmos DB is available in all Azure regions worldwide. After selecting the default consistency level for your database account, you can associate one or more regions (depending on your choice of default consistency level and global distribution needs).

1. In the Azure portal, in the left bar, click **Azure Cosmos DB**.
2. In the **Azure Cosmos DB** page, select the database account to modify.
3. In the account page, click **Replicate data globally** from the menu.
4. In the **Replicate data globally** page, select the regions to add or remove by clicking regions in the map, and then click **Save**. There is a cost to adding regions, see the pricing page or the Distribute data globally with Azure Cosmos DB article for more information.

Once you add a second region, the **Manual Failover** option is enabled on the **Replicate data globally** page in the portal. You can use this option to test the failover process or change the primary write region. Once you add a third region, the **Failover Priorities** option is enabled on the same page so that you can change the failover order for reads.

Selecting global database regions

There are two common scenarios for configuring two or more regions:

1. Delivering low-latency access to data to end users no matter where they are located around the globe
2. Adding regional resiliency for business continuity and disaster recovery (BCDR)

For delivering low-latency to end users, it is recommended that you deploy both the application and Azure Cosmos DB in the regions that correspond to where the application's users are located.

For BCDR, it is recommended to add regions based on the region pairs described in the Business continuity and disaster recovery (BCDR): Azure Paired Regions article.

Connecting to a preferred region using the SQL API

In order to take advantage of global distribution, client applications can specify the ordered preference list of regions to be used to perform document operations. Based on the Azure Cosmos DB account configuration, current regional availability and the preference list specified, the most optimal endpoint will be chosen by the SQL SDK to perform write and read operations.

This preference list is specified when initializing a connection using the SQL SDKs. The SDKs accept an optional parameter `PreferredLocations` that is an ordered list of Azure regions.

The SDK will automatically send all writes to the current write region. All reads will be sent to the first available region in the preferred locations list. If the request fails, the client will fail down the list to the next region.

The SDK will only attempt to read from the regions specified in preferred locations. So, for example, if the Azure Cosmos account is available in four regions, but the client only specifies two read(non-write) regions within the `PreferredLocations`, then no reads will be served out of the read region that is not specified in `PreferredLocations`. If the read regions specified in the `PreferredLocations` list are not available, reads will be served out of write region.

The application can verify the current write endpoint and read endpoint chosen by the SDK by checking two properties, `WriteEndpoint` and `ReadEndpoint`, available in SDK version 1.8 and above. If the `PreferredLocations` property is not set, all requests will be served from the current write region.

If you don't specify the preferred locations but used the `setCurrentLocation` method, the SDK automatically populates the preferred locations based on the current region that the client is running in. The SDK orders the regions based on the proximity of a region to the current region.

.NET SDK

The SDK can be used without any code changes. In this case, the SDK automatically directs both reads and writes to the current write region.

In version 1.8 and later of the .NET SDK, the ConnectionPolicy parameter for the DocumentClient constructor has a property called Microsoft.Azure.Documents.ConnectionPolicy.PreferredLocations. This property is of type Collection `<string>` and should contain a list of region names. The string values are formatted per the region name column on the Azure Regions page, with no spaces before or after the first and last character respectively.

The current write and read endpoints are available in DocumentClient.WriteEndpoint and DocumentClient.ReadEndpoint respectively.

> **NOTE**
> The URLs for the endpoints should not be considered as long-lived constants. The service may update these at any point. The SDK handles this change automatically.

- .NET SDK V2
- .NET SDK V3

If you are using the .NET V2 SDK, use the `PreferredLocations` property to set the preferred region.

```
// Getting endpoints from application settings or other configuration location
Uri accountEndPoint = new Uri(Properties.Settings.Default.GlobalDatabaseUri);
string accountKey = Properties.Settings.Default.GlobalDatabaseKey;

ConnectionPolicy connectionPolicy = new ConnectionPolicy();

//Setting read region selection preference
connectionPolicy.PreferredLocations.Add(LocationNames.WestUS); // first preference
connectionPolicy.PreferredLocations.Add(LocationNames.EastUS); // second preference
connectionPolicy.PreferredLocations.Add(LocationNames.NorthEurope); // third preference

// initialize connection
DocumentClient docClient = new DocumentClient(
    accountEndPoint,
    accountKey,
    connectionPolicy);

// connect to DocDB
await docClient.OpenAsync().ConfigureAwait(false);
```

Alternatively, you can use the `SetCurrentLocation` property and let the SDK choose the preferred location based on proximity.

```
// Getting endpoints from application settings or other configuration location
Uri accountEndPoint = new Uri(Properties.Settings.Default.GlobalDatabaseUri);
string accountKey = Properties.Settings.Default.GlobalDatabaseKey;

ConnectionPolicy connectionPolicy = new ConnectionPolicy();

connectionPolicy.SetCurrentLocation("West US 2"); /

// initialize connection
DocumentClient docClient = new DocumentClient(
    accountEndPoint,
    accountKey,
    connectionPolicy);

// connect to DocDB
await docClient.OpenAsync().ConfigureAwait(false);
```

Node.js/JavaScript

> **NOTE**
>
> The URLs for the endpoints should not be considered as long-lived constants. The service may update these at any point. The SDK will handle this change automatically.

Below is a code example for Node.js/Javascript.

```
// Setting read region selection preference, in the following order -
// 1 - West US
// 2 - East US
// 3 - North Europe
const preferredLocations = ['West US', 'East US', 'North Europe'];

// initialize the connection
const client = new CosmosClient{ endpoint, key, connectionPolicy: { preferredLocations } });
```

Python SDK

The following code shows how to set preferred locations by using the Python SDK:

```
connectionPolicy = documents.ConnectionPolicy()
connectionPolicy.PreferredLocations = ['West US', 'East US', 'North Europe']
client = cosmos_client.CosmosClient(ENDPOINT, {'masterKey': MASTER_KEY}, connectionPolicy)
```

Java V4 SDK

The following code shows how to set preferred locations by using the Java SDK:

- Async
- Sync

Java SDK V4 (Maven com.azure::azure-cosmos) Async API

```
ArrayList<String> preferredRegions = new ArrayList<String>();
preferredRegions.add("East US");
preferredRegions.add( "West US");
preferredRegions.add("Canada Central");

CosmosAsyncClient client =
        new CosmosClientBuilder()
                .endpoint(HOST)
                .key(MASTER_KEY)
                .preferredRegions(preferredRegions)
                .contentResponseOnWriteEnabled(true)
                .buildAsyncClient();
```

REST

Once a database account has been made available in multiple regions, clients can query its availability by performing a GET request on this URI `https://{databaseaccount}.documents.azure.com/`

The service will return a list of regions and their corresponding Azure Cosmos DB endpoint URIs for the replicas. The current write region will be indicated in the response. The client can then select the appropriate endpoint for all further REST API requests as follows.

Example response

```
{
    "_dbs": "//dbs/",
    "media": "//media/",
    "writableLocations": [
        {
            "Name": "West US",
            "DatabaseAccountEndpoint": "https://globaldbexample-westus.documents.azure.com:443/"
        }
    ],
    "readableLocations": [
        {
            "Name": "East US",
            "DatabaseAccountEndpoint": "https://globaldbexample-eastus.documents.azure.com:443/"
        }
    ],
    "MaxMediaStorageUsageInMB": 2048,
    "MediaStorageUsageInMB": 0,
    "ConsistencyPolicy": {
        "defaultConsistencyLevel": "Session",
        "maxStalenessPrefix": 100,
        "maxIntervalInSeconds": 5
    },
    "addresses": "//addresses/",
    "id": "globaldbexample",
    "_rid": "globaldbexample.documents.azure.com",
    "_self": "",
    "_ts": 0,
    "_etag": null
}
```


- All PUT, POST and DELETE requests must go to the indicated write URI
- All GETs and other read-only requests (for example queries) may go to any endpoint of the client's choice

Write requests to read-only regions will fail with HTTP error code 403 ("Forbidden").

If the write region changes after the client's initial discovery phase, subsequent writes to the previous write region will fail with HTTP error code 403 ("Forbidden"). The client should then GET the list of regions again to get the updated write region.

That's it, that completes this tutorial. You can learn how to manage the consistency of your globally replicated account by reading Consistency levels in Azure Cosmos DB. And for more information about how global database replication works in Azure Cosmos DB, see Distribute data globally with Azure Cosmos DB.

Next steps

In this tutorial, you've done the following:

- Configure global distribution using the Azure portal
- Configure global distribution using the SQL APIs

You can now proceed to the next tutorial to learn how to develop locally using the Azure Cosmos DB local emulator.

Develop locally with the emulator

Azure Cosmos DB: .NET examples for the SQL API

11/2/2020 • 5 minutes to read • Edit Online

APPLIES TO: SQL API

The azure-cosmos-dotnet-v2 GitHub repository includes the latest .NET sample solutions to perform CRUD and other common operations on Azure Cosmos DB resources. This article provides:

- Links to the tasks in each of the example C# project files.
- Links to the related API reference content.

For .NET SDK Version 3.0 (Preview) code samples, see the latest samples in the azure-cosmos-dotnet-v3 GitHub repository.

Prerequisites

Visual Studio 2019 with the Azure development workflow installed

- You can download and use the **free** Visual Studio 2019 Community Edition. Make sure that you enable **Azure development** during the Visual Studio setup.

The Microsoft.Azure.DocumentDB NuGet package

An Azure subscription or free Cosmos DB trial account

- If you don't have an Azure subscription, create a free account before you begin.
- You can activate Visual Studio subscriber benefits: Your Visual Studio subscription gives you credits every month, which you can use for paid Azure services.
- You can Try Azure Cosmos DB for free without an Azure subscription, free of charge and commitments, or create an Azure Cosmos DB free tier account, with the first 400 RU/s and 5 GB of storage for free. You can also use the Azure Cosmos DB Emulator with a URI of `https://localhost:8081`. For the key to use with the emulator, see Authenticating requests.

> **NOTE**
> The samples are self-contained, and set up and clean up after themselves with multiple calls to CreateDocumentCollectionAsync(). Each occurrence bills your subscription for one hour of usage in your collection's performance tier.

Database examples

The RunDatabaseDemo method of the sample *DatabaseManagement* project shows how to do the following tasks. To learn about Azure Cosmos databases before you run the following samples, see Work with databases, containers, and items.

TASK	API REFERENCE
Create a database	DocumentClient.CreateDatabaseIfNotExistsAsync
Read a database by ID	DocumentClient.ReadDatabaseAsync

TASK	API REFERENCE
Read all the databases	DocumentClient.ReadDatabaseFeedAsync
Delete a database	DocumentClient.DeleteDatabaseAsync

Collection examples

The RunCollectionDemo method of the sample *CollectionManagement* project shows how to do the following tasks. To learn about Azure Cosmos collections before you run the following samples, see Work with databases, containers, and items.

TASK	API REFERENCE
Create a collection	DocumentClient.CreateDocumentCollectionIfNotExistsAsync
Change configured performance of a collection	DocumentClient.ReplaceOfferAsync
Get a collection by ID	DocumentClient.ReadDocumentCollectionAsync
Read all the collections in a database	DocumentClient.ReadDocumentCollectionFeedAsync
Delete a collection	DocumentClient.DeleteDocumentCollectionAsync

Document examples

The RunDocumentsDemo method of the sample *DocumentManagement* project shows how to do the following tasks. To learn about Azure Cosmos documents before you run the following samples, see Work with databases, containers, and items.

TASK	API REFERENCE
Create a document	DocumentClient.CreateDocumentAsync
Read a document by ID	DocumentClient.ReadDocumentAsync
Read all the documents in a collection	DocumentClient.ReadDocumentFeedAsync
Query for documents	DocumentClient.CreateDocumentQuery
Replace a document	DocumentClient.ReplaceDocumentAsync
Upsert a document	DocumentClient.UpsertDocumentAsync
Delete a document	DocumentClient.DeleteDocumentAsync
Working with .NET dynamic objects	DocumentClient.CreateDocumentAsync DocumentClient.ReadDocumentAsync DocumentClient.ReplaceDocumentAsync
Replace document with conditional ETag check	DocumentClient.AccessCondition Documents.Client.AccessConditionType

TASK	API REFERENCE
Read document only if document has changed	DocumentClient.AccessCondition Documents.Client.AccessConditionType

Indexing examples

The RunIndexDemo method of the sample *IndexManagement* project shows how to do the following tasks. To learn about indexing in Azure Cosmos DB before you run the following samples, see index policies, index types, and index paths.

TASK	API REFERENCE
Exclude a document from the index	IndexingDirective.Exclude
Use Lazy indexing	IndexingPolicy.IndexingMode
Exclude specified document paths from the index	IndexingPolicy.ExcludedPaths
Force a range scan operation on a hash indexed path	FeedOptions.EnableScanInQuery
Use range indexes on strings	IndexingPolicy.IncludedPaths RangeIndex
Perform an index transform	ReplaceDocumentCollectionAsync

Geospatial examples

The RunDemoAsync method of the sample *Geospatial* project shows how to do the following tasks. To learn about GeoJSON and geospatial data before you run the following samples, see Use geospatial and GeoJSON location data.

TASK	API REFERENCE
Enable geospatial indexing on a new collection	IndexingPolicy IndexKind.Spatial DataType.Point
Insert documents with GeoJSON points	DocumentClient.CreateDocumentAsync DataType.Point
Find points within a specified distance	ST_DISTANCE GeometryOperationExtensions.Distance
Find points within a polygon	ST_WITHIN GeometryOperationExtensions.Within Polygon
Enable geospatial indexing on an existing collection	DocumentClient.ReplaceDocumentCollectionAsync DocumentCollection.IndexingPolicy

TASK	API REFERENCE
Validate point and polygon data	ST_ISVALID ST_ISVALIDDETAILED GeometryOperationExtensions.IsValid GeometryOperationExtensions.IsValidDetailed

Query examples

The RunDemoAsync method of the sample *Queries* project shows how to do the following tasks using the SQL query grammar, the LINQ provider with query, and Lambda. To learn about the SQL query reference in Azure Cosmos DB before you run the following samples, see SQL query examples for Azure Cosmos DB.

TASK	API REFERENCE
Query for all documents	DocumentQueryable.CreateDocumentQuery
Query for equality using ==	DocumentQueryable.CreateDocumentQuery
Query for inequality using != and NOT	DocumentQueryable.CreateDocumentQuery
Query using range operators like >, <, >=, <=	DocumentQueryable.CreateDocumentQuery
Query using range operators against strings	DocumentQueryable.CreateDocumentQuery
Query with ORDER BY	DocumentQueryable.CreateDocumentQuery
Query with aggregate functions	DocumentQueryable.CreateDocumentQuery
Work with subdocuments	DocumentQueryable.CreateDocumentQuery
Query with intra-document Joins	DocumentQueryable.CreateDocumentQuery
Query with string, math, and array operators	DocumentQueryable.CreateDocumentQuery
Query with parameterized SQL using SqlQuerySpec	DocumentQueryable.CreateDocumentQuery SqlQuerySpec
Query with explicit paging	DocumentQueryable.CreateDocumentQuery
Query partitioned collections in parallel	DocumentQueryable.CreateDocumentQuery
Query with ORDER BY for partitioned collections	DocumentQueryable.CreateDocumentQuery

Change feed examples

The RunDemoAsync method of the sample *ChangeFeed* project shows how to do the following tasks. To learn about change feed in Azure Cosmos DB before you run the following samples, see Read Azure Cosmos DB change feed and Change feed processor.

TASK	API REFERENCE
Read change feed	DocumentClient.CreateDocumentChangeFeedQuery
Read partition key ranges	DocumentClient.ReadPartitionKeyRangeFeedAsync

The change feed processor sample, ChangeFeedMigrationTool, shows how to use the change feed processor library to replicate data to another Cosmos container.

Server-side programming examples

The RunDemoAsync method of the sample *ServerSideScripts* project shows how to do the following tasks. To learn about server-side programming in Azure Cosmos DB before you run the following samples, see Stored procedures, triggers, and user-defined functions.

TASK	API REFERENCE
Create a stored procedure	DocumentClient.CreateStoredProcedureAsync
Execute a stored procedure	DocumentClient.ExecuteStoredProcedureAsync
Read a document feed for a stored procedure	DocumentClient.ReadDocumentFeedAsync
Create a stored procedure with ORDER BY	DocumentClient.CreateStoredProcedureAsync
Create a pre-trigger	DocumentClient.CreateTriggerAsync
Create a post-trigger	DocumentClient.CreateTriggerAsync
Create a user-defined function (UDF)	DocumentClient.CreateUserDefinedFunctionAsync

User management examples

The RunDemoAsync method of the sample *UserManagement* project shows how to do the following tasks:

TASK	API REFERENCE
Create a user	DocumentClient.CreateUserAsync
Set permissions on a collection or document	DocumentClient.CreatePermissionAsync
Get a list of a user's permissions	DocumentClient.ReadUserAsync DocumentClient.ReadPermissionFeedAsync

Azure Cosmos DB.NET V3 SDK (Microsoft.Azure.Cosmos) examples for the SQL API

11/2/2020 • 3 minutes to read • Edit Online

APPLIES TO: SQL API

The azure-cosmos-dotnet-v3 GitHub repository includes the latest .NET sample solutions to perform CRUD and other common operations on Azure Cosmos DB resources. If you're familiar with the previous version of the .NET SDK, you may be used to the terms collection and document. Because Azure Cosmos DB supports multiple API models, version 3.0 of the .NET SDK uses the generic terms "container" and "item". A container can be a collection, graph, or table. An item can be a document, edge/vertex, or row, and is the content inside a container. This article provides:

- Links to the tasks in each of the example C# project files.
- Links to the related API reference content.

Prerequisites

Visual Studio 2019 with the Azure development workflow installed

- You can download and use the **free** Visual Studio 2019 Community Edition. Make sure that you enable **Azure development** during the Visual Studio setup.

  The Microsoft.Azure.cosmos NuGet package

An Azure subscription or free Cosmos DB trial account

- If you don't have an Azure subscription, create a free account before you begin.
- You can activate Visual Studio subscriber benefits: Your Visual Studio subscription gives you credits every month, which you can use for paid Azure services.
- You can Try Azure Cosmos DB for free without an Azure subscription, free of charge and commitments, or create an Azure Cosmos DB free tier account, with the first 400 RU/s and 5 GB of storage for free. You can also use the Azure Cosmos DB Emulator with a URI of `https://localhost:8081`. For the key to use with the emulator, see Authenticating requests.

> **NOTE**
>
> The samples are self-contained, and set up and clean up after themselves. Each occurrence bills your subscription for one hour of usage in your container's performance tier.

Database examples

The RunDatabaseDemo method of the sample *DatabaseManagement* project shows how to do the following tasks. To learn about Azure Cosmos databases before you run the following samples, see Work with databases, containers, and items.

TASK	API REFERENCE
Create a database	CosmosClient.CreateDatabaseIfNotExistsAsync
Read a database by ID	Database.ReadAsync
Read all the databases for an account	CosmosClient.GetDatabaseQueryIterator
Delete a database	Database.DeleteAsync

Container examples

The RunContainerDemo method of the sample *ContainerManagement* project shows how to do the following tasks. To learn about Azure Cosmos containers before you run the following samples, see Work with databases, containers, and items.

TASK	API REFERENCE
Create a container	Database.CreateContainerIfNotExistsAsync
Create a container with custom index policy	Database.CreateContainerIfNotExistsAsync
Change configured performance of a container	Container.ReplaceThroughputAsync
Get a container by ID	Container.ReadContainerAsync
Read all the containers in a database	Database.GetContainerQueryIterator
Delete a container	Container.DeleteContainerAsync

Item examples

The RunItemsDemo method of the sample *ItemManagement* project shows how to do the following tasks. To learn about Azure Cosmos items before you run the following samples, see Work with databases, containers, and items.

TASK	API REFERENCE
Create an item	Container.CreateItemAsync
Read an item by ID	container.ReadItemAsync
Query for items	container.GetItemQueryIterator
Replace an item	container.ReplaceItemAsync
Upsert an item	container.UpsertItemAsync
Delete an item	container.DeleteItemAsync
Replace an item with conditional ETag check	RequestOptions.IfMatchEtag

Indexing examples

The RunIndexDemo method of the sample *IndexManagement* project shows how to do the following tasks. To learn about indexing in Azure Cosmos DB before you run the following samples, see index policies, index types, and index paths.

TASK	API REFERENCE
Exclude an item from the index	IndexingDirective.Exclude
Use Lazy indexing	IndexingPolicy.IndexingMode
Exclude specified item paths from the index	IndexingPolicy.ExcludedPaths

Query examples

The RunDemoAsync method of the sample *Queries* project shows how to do the following tasks using the SQL query grammar, the LINQ provider with query, and Lambda. To learn about the SQL query reference in Azure Cosmos DB before you run the following samples, see SQL query examples for Azure Cosmos DB.

TASK	API REFERENCE
Query items from single partition	container.GetItemQueryIterator
Query items from multiple partitions	container.GetItemQueryIterator
Query using a SQL statement	container.GetItemQueryIterator

Change feed examples

The RunBasicChangeFeed method of the sample *ChangeFeed* project shows how to do the following tasks. To learn about change feed in Azure Cosmos DB before you run the following samples, see Read Azure Cosmos DB change feed and Change feed processor.

TASK	API REFERENCE
Basic change feed functionality	Container.GetChangeFeedProcessorBuilder
Read change feed from a specific time	Container.GetChangeFeedProcessorBuilder
Read change feed from the beginning	ChangeFeedProcessorBuilder.WithStartTime(DateTime)
MIgrate from change feed processor to change feed in V3 SDK	Container.GetChangeFeedProcessorBuilder

Server-side programming examples

The RunDemoAsync method of the sample *ServerSideScripts* project shows how to do the following tasks. To learn about server-side programming in Azure Cosmos DB before you run the following samples, see Stored procedures, triggers, and user-defined functions.

TASK	API REFERENCE
Create a stored procedure	Scripts.CreateStoredProcedureAsync
Execute a stored procedure	Scripts.ExecuteStoredProcedureAsync
Delete a stored procedure	Scripts.DeleteStoredProcedureAsync

Azure Cosmos DB SQL API: Java SDK v4 examples

12/4/2020 • 6 minutes to read • Edit Online

APPLIES TO: SQL API

> **IMPORTANT**
>
> To learn more about Java SDK v4, please see the Azure Cosmos DB Java SDK v4 Release notes, Maven repository, Azure Cosmos DB Java SDK v4 performance tips, and Azure Cosmos DB Java SDK v4 troubleshooting guide for more information. If you are currently using an older version than v4, see the Migrate to Azure Cosmos DB Java SDK v4 guide for help upgrading to v4.

> **IMPORTANT**
>
> If you don't have an Azure subscription, create a free account before you begin.
>
> - You can activate Visual Studio subscriber benefits: Your Visual Studio subscription gives you credits every month that you can use for paid Azure services.
>
> You can Try Azure Cosmos DB for free without an Azure subscription, free of charge and commitments, or create an Azure Cosmos DB free tier account, with the first 400 RU/s and 5 GB of storage for free. You can also use the Azure Cosmos DB Emulator with a URI of `https://localhost:8081`. For the key to use with the emulator, see Authenticating requests.

The latest sample applications that perform CRUD operations and other common operations on Azure Cosmos DB resources are included in the azure-cosmos-java-sql-api-samples GitHub repository. This article provides:

- Links to the tasks in each of the example Java project files.
- Links to the related API reference content.

Prerequisites

You need the following to run this sample application:

- Java Development Kit 8
- Azure Cosmos DB Java SDK v4

You can optionally use Maven to get the latest Azure Cosmos DB Java SDK v4 binaries for use in your project. Maven automatically adds any necessary dependencies. Otherwise, you can directly download the dependencies listed in the pom.xml file and add them to your build path.

```
<dependency>
 <groupId>com.azure</groupId>
 <artifactId>azure-cosmos</artifactId>
 <version>LATEST</version>
</dependency>
```

Running the sample applications

Clone the sample repo:

```
$ git clone https://github.com/Azure-Samples/azure-cosmos-java-sql-api-samples.git

$ cd azure-cosmos-java-sql-api-samples
```

You can run the samples using either an IDE (Eclipse, IntelliJ, or VSCODE) or from the command line using Maven.

These environment variables must be set

```
ACCOUNT_HOST=your account hostname;ACCOUNT_KEY=your account primary key
```

in order to give the samples read/write access to your account.

To run a sample, specify its Main Class

```
com.azure.cosmos.examples.sample.synchronicity.MainClass
```

where *sample.synchronicity.MainClass* can be

- crudquickstart.sync.SampleCRUDQuickstart
- crudquickstart.async.SampleCRUDQuickstartAsync
- indexmanagement.sync.SampleIndexManagement
- indexmanagement.async.SampleIndexManagementAsync
- storedprocedure.sync.SampleStoredProcedure
- storedprocedure.async.SampleStoredProcedureAsync
- changefeed.SampleChangeFeedProcessor *(Changefeed has only an async sample, no sync sample.)* ...etc...

> **NOTE**
> Each sample is self-contained; it sets itself up and cleans up after itself. The samples issue multiple calls to create a `CosmosContainer`. Each time this is done, your subscription is billed for 1 hour of usage for the performance tier of the collection created.

Database examples

The Database CRUD Samples file shows how to perform the following tasks. To learn about the Azure Cosmos databases before running the following samples, see Working with databases, containers, and items conceptual article.

TASK	API REFERENCE
Create a database	CosmosClient.createDatabaseIfNotExists
Read a database by ID	CosmosClient.getDatabase
Read all the databases	CosmosClient.readAllDatabases
Delete a database	CosmosDatabase.delete

Collection examples

The Collection CRUD Samples file shows how to perform the following tasks. To learn about the Azure Cosmos

collections before running the following samples, see Working with databases, containers, and items conceptual article.

TASK	API REFERENCE
Create a collection	CosmosDatabase.createContainerIfNotExists
Change configured performance of a collection	CosmosContainer.replaceProvisionedThroughput
Get a collection by ID	CosmosDatabase.getContainer
Read all the collections in a database	CosmosDatabase.readAllContainers
Delete a collection	CosmosContainer.delete

Autoscale collection examples

To learn more about autoscale before running these samples, take a look at these instructions for enabling autoscale in your account and in your databases and containers.

The autoscale Database CRUD Samples file shows how to perform the following tasks.

TASK	API REFERENCE
Create a database with specified autoscale max throughput	CosmosClient.createDatabase ThroughputProperties.createAutoscaledThroughput

The autoscale Collection CRUD Samples file shows how to perform the following tasks.

TASK	API REFERENCE
Create a collection with specified autoscale max throughput	CosmosDatabase.createContainerIfNotExists
Change configured autoscale max throughput of a collection	CosmosContainer.replaceThroughput
Read autoscale throughput configuration of a collection	CosmosContainer.readThroughput

Analytical storage collection examples

The Analytical storage Collection CRUD Samples file shows how to perform the following tasks. To learn about the Azure Cosmos collections before running the following samples, read about Azure Cosmos DB Synapse and Analytical Store.

TASK	API REFERENCE
Create a collection	CosmosDatabase.createContainerIfNotExists

Document examples

The Document CRUD Samples file shows how to perform the following tasks. To learn about the Azure Cosmos documents before running the following samples, see Working with databases, containers, and items conceptual article.

TASK	API REFERENCE
Create a document	CosmosContainer.createItem
Read a document by ID	CosmosContainer.readItem
Query for documents	CosmosContainer.queryItems
Replace a document	CosmosContainer.replaceItem
Upsert a document	CosmosContainer.upsertItem
Delete a document	CosmosContainer.deleteItem
Replace a document with conditional ETag check	AccessCondition.setType AccessCondition.setCondition
Read document only if document has changed	AccessCondition.setType AccessCondition.setCondition

Indexing examples

The Collection CRUD Samples file shows how to perform the following tasks. To learn about indexing in Azure Cosmos DB before running the following samples, see indexing policies, indexing types, and indexing paths conceptual articles.

TASK	API REFERENCE
Exclude a document from the index	ExcludedIndex IndexingPolicy
Use Lazy Indexing	IndexingPolicy.IndexingMode
Include specified documents paths in the index	IndexingPolicy.IncludedPaths
Exclude specified documents paths from the index	IndexingPolicy.ExcludedPaths
Create a composite index	IndexingPolicy.setCompositeIndexes CompositePath
Force a range scan operation on a hash indexed path	FeedOptions.EnableScanInQuery
Use range indexes on Strings	IndexingPolicy.IncludedPaths RangeIndex
Perform an index transform	-
Create a geospatial index	IndexingPolicy.setSpatialIndexes SpatialSpec SpatialType

For more information about indexing, see Azure Cosmos DB indexing policies.

Query examples

The Query Samples file shows how to do the following tasks using the SQL query grammar. To learn about the SQL query reference in Azure Cosmos DB before you run the following samples, see SQL query examples for Azure Cosmos DB.

TASK	API REFERENCE
Query for all documents	CosmosContainer.queryItems
Query for equality using ==	CosmosContainer.queryItems
Query for inequality using != and NOT	CosmosContainer.queryItems
Query using range operators like >, <, >=, <=	CosmosContainer.queryItems
Query using range operators against strings	CosmosContainer.queryItems
Query with ORDER BY	CosmosContainer.queryItems
Query with DISTINCT	CosmosContainer.queryItems
Query with aggregate functions	CosmosContainer.queryItems
Work with subdocuments	CosmosContainer.queryItems
Query with intra-document Joins	CosmosContainer.queryItems
Query with string, math, and array operators	CosmosContainer.queryItems
Query with parameterized SQL using SqlQuerySpec	CosmosContainer.queryItems
Query with explicit paging	CosmosContainer.queryItems
Query partitioned collections in parallel	CosmosContainer.queryItems
Query with ORDER BY for partitioned collections	CosmosContainer.queryItems

Change feed examples

The Change Feed Processor Sample file shows how to do the following tasks. To learn about change feed in Azure Cosmos DB before you run the following samples, see Read Azure Cosmos DB change feed and Change feed processor.

TASK	API REFERENCE
Basic change feed functionality	ChangeFeedProcessor.changeFeedProcessorBuilder
Read change feed from a specific time	ChangeFeedProcessor.changeFeedProcessorBuilder
Read change feed from the beginning	-

Server-side programming examples

The Stored Procedure Sample file shows how to do the following tasks. To learn about server-side programming in Azure Cosmos DB before you run the following samples, see Stored procedures, triggers, and user-defined functions.

TASK	API REFERENCE
Create a stored procedure	CosmosScripts.createStoredProcedure
Execute a stored procedure	CosmosStoredProcedure.execute
Delete a stored procedure	CosmosStoredProcedure.delete

User management examples

The User Management Sample file shows how to do the following tasks:

TASK	API REFERENCE
Create a user	-
Set permissions on a collection or document	-
Get a list of a user's permissions	-

Azure Cosmos DB SQL API: Spring Data Azure Cosmos DB v3 examples

11/2/2020 • 3 minutes to read • Edit Online

APPLIES TO: SQL API

> **IMPORTANT**
>
> These release notes are for version 3 of Spring Data Azure Cosmos DB. You can find release notes for version 2 here.
>
> Spring Data Azure Cosmos DB supports only the SQL API.
>
> See these articles for information about Spring Data on other Azure Cosmos DB APIs:
>
> - Spring Data for Apache Cassandra with Azure Cosmos DB
> - Spring Data MongoDB with Azure Cosmos DB
> - Spring Data Gremlin with Azure Cosmos DB

> **IMPORTANT**
>
> If you don't have an Azure subscription, create a free account before you begin.
>
> - You can activate Visual Studio subscriber benefits: Your Visual Studio subscription gives you credits every month that you can use for paid Azure services.
>
> You can Try Azure Cosmos DB for free without an Azure subscription, free of charge and commitments, or create an Azure Cosmos DB free tier account, with the first 400 RU/s and 5 GB of storage for free. You can also use the Azure Cosmos DB Emulator with a URI of `https://localhost:8081`. For the key to use with the emulator, see Authenticating requests.

The latest sample applications that perform CRUD operations and other common operations on Azure Cosmos DB resources are included in the azure-spring-data-cosmos-java-sql-api-samples GitHub repository. This article provides:

- Links to the tasks in each of the example Spring Data Azure Cosmos DB project files.
- Links to the related API reference content.

Prerequisites

You need the following to run this sample application:

- Java Development Kit 8
- Spring Data Azure Cosmos DB v3

You can optionally use Maven to get the latest Spring Data Azure Cosmos DB v3 binaries for use in your project. Maven automatically adds any necessary dependencies. Otherwise, you can directly download the dependencies listed in the **pom.xml** file and add them to your build path.

```
<dependency>
 <groupId>com.azure</groupId>
 <artifactId>azure-spring-data-cosmos</artifactId>
 <version>LATEST</version>
</dependency>
```

Running the sample applications

Clone the sample repo:

```
$ git clone https://github.com/Azure-Samples/azure-spring-data-cosmos-java-sql-api-samples

$ cd azure-spring-data-cosmos-java-sql-api-samples
```

You can run the samples using either an IDE (Eclipse, IntelliJ, or VSCODE) or from the command line using Maven.

In **application.properties** these environment variables must be set

```
cosmos.uri=${ACCOUNT_HOST}
cosmos.key=${ACCOUNT_KEY}
cosmos.secondaryKey=${SECONDARY_ACCOUNT_KEY}

dynamic.collection.name=spel-property-collection
# Populate query metrics
cosmos.queryMetricsEnabled=true
```

in order to give the samples read/write access to your account, databases and containers.

Your IDE may provide the ability to execute the Spring Data sample code. Otherwise you may use the following terminal command to execute the sample:

```
mvn spring-boot:run
```

Document CRUD examples

The samples file shows how to perform the following tasks. To learn about Azure Cosmos documents before running the following samples, see Working with databases, containers, and items conceptual article.

TASK	API REFERENCE
Create a document	CosmosRepository.save
Read a document by ID	CosmosRepository.derivedQueryMethod
Delete all documents	CosmosRepository.deleteAll

Derived query method examples

The samples file shows how to perform the following tasks. To learn about Azure Cosmos DB queries before running the following samples, you may find it helpful to read Baeldung's Derived Query Methods in Spring article.

| Query for documents | CosmosRepository.derivedQueryMethod |

Custom query examples

The samples file shows how to do the following tasks using the SQL query grammar. To learn about the SQL query reference in Azure Cosmos DB before you run the following samples, see SQL query examples for Azure Cosmos DB.

TASK	API REFERENCE
Query for all documents	@Query annotation
Query for equality using ==	@Query annotation
Query for inequality using != and NOT	@Query annotation
Query using range operators like >, <, >=, <=	@Query annotation
Query using range operators against strings	@Query annotation
Query with ORDER BY	@Query annotation
Query with DISTINCT	@Query annotation
Query with aggregate functions	@Query annotation
Work with subdocuments	@Query annotation
Query with intra-document Joins	@Query annotation
Query with string, math, and array operators	@Query annotation

Node.js examples to manage data in Azure Cosmos DB

11/2/2020 • 2 minutes to read • Edit Online

APPLIES TO: SQL API

Sample solutions that perform CRUD operations and other common operations on Azure Cosmos DB resources are included in the azure-cosmos-js GitHub repository. This article provides:

- Links to the tasks in each of the Node.js example project files.
- Links to the related API reference content.

Prerequisites

If you don't have an Azure subscription, create a free account before you begin.

- You can activate Visual Studio subscriber benefits: Your Visual Studio subscription gives you credits every month that you can use for paid Azure services.

You can Try Azure Cosmos DB for free without an Azure subscription, free of charge and commitments, or create an Azure Cosmos DB free tier account, with the first 400 RU/s and 5 GB of storage for free. You can also use the Azure Cosmos DB Emulator with a URI of `https://localhost:8081`. For the key to use with the emulator, see Authenticating requests.

You also need the JavaScript SDK.

> **NOTE**
> Each sample is self-contained, it sets itself up and cleans up after itself. As such, the samples issue multiple calls to Containers.create. Each time this is done your subscription will be billed for 1 hour of usage per the performance tier of the container being created.

Database examples

The DatabaseManagement file shows how to perform the CRUD operations on the database. To learn about the Azure Cosmos databases before running the following samples, see Working with databases, containers, and items conceptual article.

TASK	API REFERENCE
Create a database if it does not exist	Databases.createIfNotExists
List databases for an account	Databases.readAll
Read a database by ID	Database.read
Delete a database	Database.delete

Container examples

The ContainerManagement file shows how to perform the CRUD operations on the container. To learn about the Azure Cosmos collections before running the following samples, see Working with databases, containers, and items conceptual article.

TASK	API REFERENCE
Create a container if it does not exist	Containers.createIfNotExists
List containers for an account	Containers.readAll
Read a container definition	Container.read
Delete a container	Container.delete

Item examples

The ItemManagement file shows how to perform the CRUD operations on the item. To learn about the Azure Cosmos documents before running the following samples, see Working with databases, containers, and items conceptual article.

TASK	API REFERENCE
Create items	Items.create
Read all items in a container	Items.readAll
Read an item by ID	Item.read
Read item only if item has changed	Item.read RequestOptions.accessCondition
Query for documents	Items.query
Replace an item	Item.replace
Replace item with conditional ETag check	Item.replace RequestOptions.accessCondition
Delete an item	Item.delete

Indexing examples

The IndexManagement file shows how to manage indexing. To learn about indexing in Azure Cosmos DB before running the following samples, see indexing policies, indexing types, and indexing paths conceptual articles.

TASK	API REFERENCE
Manually index a specific item	RequestOptions.indexingDirective: 'include'
Manually exclude a specific item from the index	RequestOptions.indexingDirective: 'exclude'
Exclude a path from the index	IndexingPolicy.ExcludedPath

TASK	API REFERENCE
Create a range index on a string path	IndexKind.Range, IndexingPolicy, Items.query
Create a container with default indexPolicy, then update this online	Containers.create

Server-side programming examples

The index.ts file of the ServerSideScripts project shows how to perform the following tasks. To learn about Server-side programming in Azure Cosmos DB before running the following samples, see Stored procedures, triggers, and user-defined functions conceptual article.

TASK	API REFERENCE
Create a stored procedure	StoredProcedures.create
Execute a stored procedure	StoredProcedure.execute

For more information about server-side programming, see Azure Cosmos DB server-side programming: Stored procedures, database triggers, and UDFs.

Azure Cosmos DB Python examples

11/2/2020 • 2 minutes to read • Edit Online

APPLIES TO: SQL API

Sample solutions that do CRUD operations and other common operations on Azure Cosmos DB resources are included in the azure-documentdb-python GitHub repository. This article provides:

- Links to the tasks in each of the Python example project files.
- Links to the related API reference content.

Prerequisites

- A Cosmos DB Account. You options are:
  - Within an Azure active subscription:
    - Create an Azure free Account or use your existing subscription
    - Visual Studio Monthly Credits
    - Azure Cosmos DB Free Tier
  - Without an Azure active subscription:
    - Try Azure Cosmos DB for free, a tests environment that lasts for 30 days.
    - Azure Cosmos DB Emulator
- Python 2.7 or 3.5.3+, with the `python` executable in your `PATH`.
- Visual Studio Code.
- The Python extension for Visual Studio Code.
- Git.
- Azure Cosmos DB SQL API SDK for Python

Database examples

The database_management.py Python sample shows how to do the following tasks. To learn about the Azure Cosmos databases before running the following samples, see Working with databases, containers, and items conceptual article.

TASK	API REFERENCE
Create a database	CosmosClient.create_database
Read a database by ID	CosmosClient.get_database_client
Query the databases	CosmosClient.query_databases
List databases for an account	CosmosClient.list_databases
Delete a database	CosmosClient.delete_database

Container examples

The container_management.py Python sample shows how to do the following tasks. To learn about the Azure

Cosmos collections before running the following samples, see Working with databases, containers, and items conceptual article.

TASK	API REFERENCE
Query for a container	database.query_containers
Create a container	database.create_container
List all the containers in a database	database.list_containers
Get a container by its ID	database.get_container_client
Manage container's provisioned throughput	container.read_offer, container.replace_throughput
Delete a container	database.delete_container

Item examples

The item_management.py Python sample shows how to do the following tasks. To learn about the Azure Cosmos documents before running the following samples, see Working with databases, containers, and items conceptual article.

TASK	API REFERENCE
Create items in a container	container.create_item
Read an item by its ID	container.read_item
Read all the items in a container	container.read_all_items
Query an item by its ID	container.query_items
Replace an item	container.replace_items
Upsert an item	container.upsert_item
Delete an item	container.delete_item
Get the change feed of items in a container	container.query_items_change_feed

Indexing examples

The index_management.py Python sample shows how to do the following tasks. To learn about indexing in Azure Cosmos DB before running the following samples, see indexing policies, indexing types, and indexing paths conceptual articles.

TASK	API REFERENCE
Exclude a specific item from indexing	documents.IndexingDirective.Exclude
Use manual indexing with specific items indexed	documents.IndexingDirective.Include

TASK	API REFERENCE
Exclude paths from indexing	Define paths to exclude in `IndexingPolicy` property
Use range indexes on strings	Define indexing policy with range indexes on string data type. `'kind': documents.IndexKind.Range`, `'dataType': documents.DataType.String`
Perform an index transformation	database.replace_container (use the updated indexing policy)
Use scans when only hash index exists on the path	set the `enable_scan_in_query=True` and `enable_cross_partition_query=True` when querying the items

Azure PowerShell samples for Azure Cosmos DB Core (SQL) API

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: SQL API

The following table includes links to commonly used Azure PowerShell scripts for Azure Cosmos DB. Use the links on the right to navigate to API specific samples. Common samples are the same across all APIs. Reference pages for all Azure Cosmos DB PowerShell cmdlets are available in the Azure PowerShell Reference. Please check for updates to `Az.CosmosDB` regularly. You can also fork these PowerShell samples for Cosmos DB from our GitHub repository, Cosmos DB PowerShell Samples on GitHub.

For PowerShell cmdlets for other APIs see PowerShell Samples for Cassandra, PowerShell Samples for MongoDB API, PowerShell Samples for Gremlin, PowerShell Samples for Table

Common Samples

TASK	DESCRIPTION
Update an account	Update a Cosmos DB account's default consistency level.
Update an account's regions	Update a Cosmos DB account's regions.
Change failover priority or trigger failover	Change the regional failover priority of an Azure Cosmos account or trigger a manual failover.
Account keys or connection strings	Get primary and secondary keys, connection strings or regenerate an account key of an Azure Cosmos DB account.
Create a Cosmos Account with IP Firewall	Create an Azure Cosmos DB account with IP Firewall enabled.

Core (SQL) API Samples

TASK	DESCRIPTION
Create an account, database and container	Create an Azure Cosmos DB account, database and container.
Create an account, database and container with autoscale	Create an Azure Cosmos DB account, database and container with autoscale.
Create a container with a large partition key	Create a container with a large partition key.
Create a container with no index policy	Create an Azure Cosmos container with index policy turned off.
List or get databases or containers	List or get database or containers.

TASK	DESCRIPTION
Throughput operations	Throughput operations for a database or container including get, update and migrate between autoscale and standard throughput.
Lock resources from deletion	Prevent resources from being deleted with resource locks.

Azure CLI samples for Azure Cosmos DB Core (SQL) API

11/2/2020 • 2 minutes to read • Edit Online

APPLIES TO: SQL API

The following table includes links to sample Azure CLI scripts for Azure Cosmos DB. Use the links on the right to navigate to API specific samples. Common samples are the same across all APIs. Reference pages for all Azure Cosmos DB CLI commands are available in the Azure CLI Reference. Azure Cosmos DB CLI script samples can also be found in the Azure Cosmos DB CLI GitHub Repository.

These samples require Azure CLI version 2.12.1 or later. Run `az --version` to find the version. If you need to install or upgrade, see Install Azure CLI

For Azure CLI samples for other APIs see CLI Samples for Cassandra, CLI Samples for MongoDB API, CLI Samples for Gremlin, CLI Samples for Table

Common Samples

These samples apply to all Azure Cosmos DB APIs

TASK	DESCRIPTION
Add or failover regions	Add a region, change failover priority, trigger a manual failover.
Account keys and connection strings	List account keys, read-only keys, regenerate keys and list connection strings.
Secure with IP firewall	Create a Cosmos account with IP firewall configured.
Secure new account with service endpoints	Create a Cosmos account and secure with service-endpoints.
Secure existing account with service endpoints	Update a Cosmos account to secure with service-endpoints when the subnet is eventually configured.

Core (SQL) API Samples

TASK	DESCRIPTION
Create an Azure Cosmos account, database and container	Creates an Azure Cosmos DB account, database, and container for Core (SQL) API.
Create an Azure Cosmos account, database and container with autoscale	Creates an Azure Cosmos DB account, database, and container with autoscale for Core (SQL) API.
Throughput operations	Read, update and migrate between autoscale and standard throughput on a database and container.

TASK	DESCRIPTION
Lock resources from deletion	Prevent resources from being deleted with resource locks.

Azure Resource Manager templates for Azure Cosmos DB

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: SQL API

This article only shows Azure Resource Manager template examples for Core (SQL) API accounts. You can also find template examples for Cassandra, Gremlin, MongoDB, and Table APIs.

Core (SQL) API

TEMPLATE	DESCRIPTION
Create an Azure Cosmos account, database, container with autoscale throughput	This template creates a Core (SQL) API account in two regions, a database and container with autoscale throughput.
Create an Azure Cosmos account, database, container with analytical store	This template creates a Core (SQL) API account in one region with a container configured with Analytical TTL enabled and option to use manual or autoscale throughput.
Create an Azure Cosmos account, database, container with standard (manual) throughput	This template creates a Core (SQL) API account in two regions, a database and container with standard throughput.
Create an Azure Cosmos account, database and container with a stored procedure, trigger and UDF	This template creates a Core (SQL) API account in two regions with a stored procedure, trigger and UDF for a container.
Create a private endpoint for an existing Azure Cosmos account	This template creates a private endpoint for an existing Azure Cosmos Core (SQL) API account in an existing virtual network.
Create a free-tier Azure Cosmos account	This template creates an Azure Cosmos DB Core (SQL) API account on free-tier.

See Azure Resource Manager reference for Azure Cosmos DB page for the reference documentation.

Data modeling in Azure Cosmos DB

12/4/2020 • 14 minutes to read • Edit Online

APPLIES TO: SQL API

While schema-free databases, like Azure Cosmos DB, make it super easy to store and query unstructured and semi-structured data, you should spend some time thinking about your data model to get the most of the service in terms of performance and scalability and lowest cost.

How is data going to be stored? How is your application going to retrieve and query data? Is your application read-heavy, or write-heavy?

After reading this article, you will be able to answer the following questions:

- What is data modeling and why should I care?
- How is modeling data in Azure Cosmos DB different to a relational database?
- How do I express data relationships in a non-relational database?
- When do I embed data and when do I link to data?

Embedding data

When you start modeling data in Azure Cosmos DB try to treat your entities as **self-contained items** represented as JSON documents.

For comparison, let's first see how we might model data in a relational database. The following example shows how a person might be stored in a relational database.

When working with relational databases, the strategy is to normalize all your data. Normalizing your data typically involves taking an entity, such as a person, and breaking it down into discrete components. In the example above, a person can have multiple contact detail records, as well as multiple address records. Contact details can be further broken down by further extracting common fields like a type. The same applies to address, each record can be of type *Home* or *Business*.

The guiding premise when normalizing data is to **avoid storing redundant data** on each record and rather refer to data. In this example, to read a person, with all their contact details and addresses, you need to use JOINS to effectively compose back (or denormalize) your data at run time.

```
SELECT p.FirstName, p.LastName, a.City, cd.Detail
FROM Person p
JOIN ContactDetail cd ON cd.PersonId = p.Id
JOIN ContactDetailType cdt ON cdt.Id = cd.TypeId
JOIN Address a ON a.PersonId = p.Id
```

Updating a single person with their contact details and addresses requires write operations across many individual tables.

Now let's take a look at how we would model the same data as a self-contained entity in Azure Cosmos DB.

```
{
    "id": "1",
    "firstName": "Thomas",
    "lastName": "Andersen",
    "addresses": [
        {
            "line1": "100 Some Street",
            "line2": "Unit 1",
            "city": "Seattle",
            "state": "WA",
            "zip": 98012
        }
    ],
    "contactDetails": [
        {"email": "thomas@andersen.com"},
        {"phone": "+1 555 555-5555", "extension": 5555}
    ]
}
```

Using the approach above we have **denormalized** the person record, by **embedding** all the information related to this person, such as their contact details and addresses, into a *single JSON* document. In addition, because we're not confined to a fixed schema we have the flexibility to do things like having contact details of different shapes entirely.

Retrieving a complete person record from the database is now a **single read operation** against a single container and for a single item. Updating a person record, with their contact details and addresses, is also a **single write operation** against a single item.

By denormalizing data, your application may need to issue fewer queries and updates to complete common operations.

When to embed

In general, use embedded data models when:

- There are **contained** relationships between entities.
- There are **one-to-few** relationships between entities.
- There is embedded data that **changes infrequently**.
- There is embedded data that will not grow **without bound**.
- There is embedded data that is **queried frequently together**.

> **NOTE**
>
> Typically denormalized data models provide better **read** performance.

When not to embed

While the rule of thumb in Azure Cosmos DB is to denormalize everything and embed all data into a single item, this can lead to some situations that should be avoided.

Take this JSON snippet.

```
{
    "id": "1",
    "name": "What's new in the coolest Cloud",
    "summary": "A blog post by someone real famous",
    "comments": [
        {"id": 1, "author": "anon", "comment": "something useful, I'm sure"},
        {"id": 2, "author": "bob", "comment": "wisdom from the interwebs"},
        …
        {"id": 100001, "author": "jane", "comment": "and on we go ..."},
        …
        {"id": 1000000001, "author": "angry", "comment": "blah angry blah angry"},
        …
        {"id": ∞ + 1, "author": "bored", "comment": "oh man, will this ever end?"},
    ]
}
```

This might be what a post entity with embedded comments would look like if we were modeling a typical blog, or CMS, system. The problem with this example is that the comments array is **unbounded**, meaning that there is no (practical) limit to the number of comments any single post can have. This may become a problem as the size of the item could grow infinitely large.

As the size of the item grows the ability to transmit the data over the wire as well as reading and updating the item, at scale, will be impacted.

In this case, it would be better to consider the following data model.

```
Post item:
{
    "id": "1",
    "name": "What's new in the coolest Cloud",
    "summary": "A blog post by someone real famous",
    "recentComments": [
        {"id": 1, "author": "anon", "comment": "something useful, I'm sure"},
        {"id": 2, "author": "bob", "comment": "wisdom from the interwebs"},
        {"id": 3, "author": "jane", "comment": "....."}
    ]
}

Comment items:
{
    "postId": "1"
    "comments": [
        {"id": 4, "author": "anon", "comment": "more goodness"},
        {"id": 5, "author": "bob", "comment": "tails from the field"},
        ...
        {"id": 99, "author": "angry", "comment": "blah angry blah angry"}
    ]
},
{
    "postId": "1"
    "comments": [
        {"id": 100, "author": "anon", "comment": "yet more"},
        ...
        {"id": 199, "author": "bored", "comment": "will this ever end?"}
    ]
}
```

This model has the three most recent comments embedded in the post container, which is an array with a fixed set of attributes. The other comments are grouped in to batches of 100 comments and stored as separate items. The size of the batch was chosen as 100 because our fictitious application allows the user to load 100 comments at a time.

Another case where embedding data is not a good idea is when the embedded data is used often across items and will change frequently.

Take this JSON snippet.

```
{
    "id": "1",
    "firstName": "Thomas",
    "lastName": "Andersen",
    "holdings": [
        {
            "numberHeld": 100,
            "stock": { "symbol": "zaza", "open": 1, "high": 2, "low": 0.5 }
        },
        {
            "numberHeld": 50,
            "stock": { "symbol": "xcxc", "open": 89, "high": 93.24, "low": 88.87 }
        }
    ]
}
```

This could represent a person's stock portfolio. We have chosen to embed the stock information into each portfolio document. In an environment where related data is changing frequently, like a stock trading application, embedding data that changes frequently is going to mean that you are constantly updating each portfolio document every time a stock is traded.

Stock *zaza* may be traded many hundreds of times in a single day and thousands of users could have *zaza* on their portfolio. With a data model like the above we would have to update many thousands of portfolio documents many times every day leading to a system that won't scale well.

Referencing data

Embedding data works nicely for many cases but there are scenarios when denormalizing your data will cause more problems than it is worth. So what do we do now?

Relational databases are not the only place where you can create relationships between entities. In a document database, you can have information in one document that relates to data in other documents. We do not recommend building systems that would be better suited to a relational database in Azure Cosmos DB, or any other document database, but simple relationships are fine and can be useful.

In the JSON below we chose to use the example of a stock portfolio from earlier but this time we refer to the stock item on the portfolio instead of embedding it. This way, when the stock item changes frequently throughout the day the only document that needs to be updated is the single stock document.

```
Person document:
{
    "id": "1",
    "firstName": "Thomas",
    "lastName": "Andersen",
    "holdings": [
        { "numberHeld":  100, "stockId": 1},
        { "numberHeld":  50, "stockId": 2}
    ]
}

Stock documents:
{
    "id": "1",
    "symbol": "zaza",
    "open": 1,
    "high": 2,
    "low": 0.5,
    "vol": 11970000,
    "mkt-cap": 42000000,
    "pe": 5.89
},
{
    "id": "2",
    "symbol": "xcxc",
    "open": 89,
    "high": 93.24,
    "low": 88.87,
    "vol": 2970200,
    "mkt-cap": 1005000,
    "pe": 75.82
}
```

An immediate downside to this approach though is if your application is required to show information about each stock that is held when displaying a person's portfolio; in this case you would need to make multiple trips to the database to load the information for each stock document. Here we've made a decision to improve the efficiency of write operations, which happen frequently throughout the day, but in turn compromised on the read operations that potentially have less impact on the performance of this particular system.

> **NOTE**
>
> Normalized data models **can require more round trips** to the server.

What about foreign keys?

Because there is currently no concept of a constraint, foreign-key or otherwise, any inter-document relationships that you have in documents are effectively "weak links" and will not be verified by the database itself. If you want to ensure that the data a document is referring to actually exists, then you need to do this in your application, or through the use of server-side triggers or stored procedures on Azure Cosmos DB.

When to reference

In general, use normalized data models when:

- Representing **one-to-many** relationships.
- Representing **many-to-many** relationships.
- Related data **changes frequently**.
- Referenced data could be **unbounded**.

> **NOTE**
> Typically normalizing provides better **write** performance.

Where do I put the relationship?

The growth of the relationship will help determine in which document to store the reference.

If we look at the JSON below that models publishers and books.

```
Publisher document:
{
    "id": "mspress",
    "name": "Microsoft Press",
    "books": [ 1, 2, 3, ..., 100, ..., 1000]
}

Book documents:
{"id": "1", "name": "Azure Cosmos DB 101" }
{"id": "2", "name": "Azure Cosmos DB for RDBMS Users" }
{"id": "3", "name": "Taking over the world one JSON doc at a time" }
...
{"id": "100", "name": "Learn about Azure Cosmos DB" }
...
{"id": "1000", "name": "Deep Dive into Azure Cosmos DB" }
```

If the number of the books per publisher is small with limited growth, then storing the book reference inside the publisher document may be useful. However, if the number of books per publisher is unbounded, then this data model would lead to mutable, growing arrays, as in the example publisher document above.

Switching things around a bit would result in a model that still represents the same data but now avoids these large mutable collections.

```
Publisher document:
{
    "id": "mspress",
    "name": "Microsoft Press"
}

Book documents:
{"id": "1","name": "Azure Cosmos DB 101", "pub-id": "mspress"}
{"id": "2","name": "Azure Cosmos DB for RDBMS Users", "pub-id": "mspress"}
{"id": "3","name": "Taking over the world one JSON doc at a time"}
...
{"id": "100","name": "Learn about Azure Cosmos DB", "pub-id": "mspress"}
...
{"id": "1000","name": "Deep Dive into Azure Cosmos DB", "pub-id": "mspress"}
```

In the above example, we have dropped the unbounded collection on the publisher document. Instead we just have a reference to the publisher on each book document.

How do I model many:many relationships?

In a relational database *many:many* relationships are often modeled with join tables, which just join records from other tables together.

You might be tempted to replicate the same thing using documents and produce a data model that looks similar to the following.

```
Author documents:
{"id": "a1", "name": "Thomas Andersen" }
{"id": "a2", "name": "William Wakefield" }

Book documents:
{"id": "b1", "name": "Azure Cosmos DB 101" }
{"id": "b2", "name": "Azure Cosmos DB for RDBMS Users" }
{"id": "b3", "name": "Taking over the world one JSON doc at a time" }
{"id": "b4", "name": "Learn about Azure Cosmos DB" }
{"id": "b5", "name": "Deep Dive into Azure Cosmos DB" }

Joining documents:
{"authorId": "a1", "bookId": "b1" }
{"authorId": "a2", "bookId": "b1" }
{"authorId": "a1", "bookId": "b2" }
{"authorId": "a1", "bookId": "b3" }
```

This would work. However, loading either an author with their books, or loading a book with its author, would always require at least two additional queries against the database. One query to the joining document and then another query to fetch the actual document being joined.

If all this join table is doing is gluing together two pieces of data, then why not drop it completely? Consider the following.

```
Author documents:
{"id": "a1", "name": "Thomas Andersen", "books": ["b1", "b2", "b3"]}
{"id": "a2", "name": "William Wakefield", "books": ["b1", "b4"]}

Book documents:
{"id": "b1", "name": "Azure Cosmos DB 101", "authors": ["a1", "a2"]}
{"id": "b2", "name": "Azure Cosmos DB for RDBMS Users", "authors": ["a1"]}
{"id": "b3", "name": "Learn about Azure Cosmos DB", "authors": ["a1"]}
{"id": "b4", "name": "Deep Dive into Azure Cosmos DB", "authors": ["a2"]}
```

Now, if I had an author, I immediately know which books they have written, and conversely if I had a book document loaded I would know the IDs of the author(s). This saves that intermediary query against the join table reducing the number of server round trips your application has to make.

Hybrid data models

We've now looked embedding (or denormalizing) and referencing (or normalizing) data, each have their upsides and each have compromises as we have seen.

It doesn't always have to be either or, don't be scared to mix things up a little.

Based on your application's specific usage patterns and workloads there may be cases where mixing embedded and referenced data makes sense and could lead to simpler application logic with fewer server round trips while still maintaining a good level of performance.

Consider the following JSON.

```
Author documents:
{
    "id": "a1",
    "firstName": "Thomas",
    "lastName": "Andersen",
    "countOfBooks": 3,
    "books": ["b1", "b2", "b3"],
    "images": [
        {"thumbnail": "https://....png"}
        {"profile": "https://....png"}
        {"large": "https://....png"}
    ]
},
{
    "id": "a2",
    "firstName": "William",
    "lastName": "Wakefield",
    "countOfBooks": 1,
    "books": ["b1"],
    "images": [
        {"thumbnail": "https://....png"}
    ]
}

Book documents:
{
    "id": "b1",
    "name": "Azure Cosmos DB 101",
    "authors": [
        {"id": "a1", "name": "Thomas Andersen", "thumbnailUrl": "https://....png"},
        {"id": "a2", "name": "William Wakefield", "thumbnailUrl": "https://....png"}
    ]
},
{
    "id": "b2",
    "name": "Azure Cosmos DB for RDBMS Users",
    "authors": [
        {"id": "a1", "name": "Thomas Andersen", "thumbnailUrl": "https://....png"},
    ]
}
```

Here we've (mostly) followed the embedded model, where data from other entities are embedded in the top-level document, but other data is referenced.

If you look at the book document, we can see a few interesting fields when we look at the array of authors. There is an `id` field that is the field we use to refer back to an author document, standard practice in a normalized model, but then we also have `name` and `thumbnailUrl`. We could have stuck with `id` and left the application to get any additional information it needed from the respective author document using the "link", but because our application displays the author's name and a thumbnail picture with every book displayed we can save a round trip to the server per book in a list by denormalizing **some** data from the author.

Sure, if the author's name changed or they wanted to update their photo we'd have to go and update every book they ever published but for our application, based on the assumption that authors don't change their names often, this is an acceptable design decision.

In the example, there are **pre-calculated aggregates** values to save expensive processing on a read operation. In the example, some of the data embedded in the author document is data that is calculated at run-time. Every time a new book is published, a book document is created **and** the countOfBooks field is set to a calculated value based on the number of book documents that exist for a particular author. This optimization would be good in read heavy systems where we can afford to do computations on writes in order to optimize reads.

The ability to have a model with pre-calculated fields is made possible because Azure Cosmos DB supports **multi-document transactions**. Many NoSQL stores cannot do transactions across documents and therefore advocate design decisions, such as "always embed everything", due to this limitation. With Azure Cosmos DB, you can use server-side triggers, or stored procedures, that insert books and update authors all within an ACID transaction. Now you don't **have** to embed everything into one document just to be sure that your data remains consistent.

Distinguishing between different document types

In some scenarios, you may want to mix different document types in the same collection; this is usually the case when you want multiple, related documents to sit in the same partition. For example, you could put both books and book reviews in the same collection and partition it by `bookId`. In such situation, you usually want to add to your documents with a field that identifies their type in order to differentiate them.

```
Book documents:
{
    "id": "b1",
    "name": "Azure Cosmos DB 101",
    "bookId": "b1",
    "type": "book"
}

Review documents:
{
    "id": "r1",
    "content": "This book is awesome",
    "bookId": "b1",
    "type": "review"
},
{
    "id": "r2",
    "content": "Best book ever!",
    "bookId": "b1",
    "type": "review"
}
```

Next steps

The biggest takeaways from this article are to understand that data modeling in a schema-free world is as important as ever.

Just as there is no single way to represent a piece of data on a screen, there is no single way to model your data. You need to understand your application and how it will produce, consume, and process the data. Then, by applying some of the guidelines presented here you can set about creating a model that addresses the immediate needs of your application. When your applications need to change, you can leverage the flexibility of a schema-free database to embrace that change and evolve your data model easily.

To learn more about Azure Cosmos DB, refer to the service's documentation page.

To understand how to shard your data across multiple partitions, refer to Partitioning Data in Azure Cosmos DB.

To learn how to model and partition data on Azure Cosmos DB using a real-world example, refer to Data Modeling and Partitioning - a Real-World Example.

How to model and partition data on Azure Cosmos DB using a real-world example

12/4/2020 • 17 minutes to read • Edit Online

APPLIES TO: SQL API

This article builds on several Azure Cosmos DB concepts like data modeling, partitioning, and provisioned throughput to demonstrate how to tackle a real-world data design exercise.

If you usually work with relational databases, you have probably built habits and intuitions on how to design a data model. Because of the specific constraints, but also the unique strengths of Azure Cosmos DB, most of these best practices don't translate well and may drag you into suboptimal solutions. The goal of this article is to guide you through the complete process of modeling a real-world use-case on Azure Cosmos DB, from item modeling to entity colocation and container partitioning.

Download or view a community-generated source code that illustrates the concepts from this article. This code sample was contributed by a community contributor and Azure Cosmos DB team doesn't support its maintenance.

The scenario

For this exercise, we are going to consider the domain of a blogging platform where *users* can create *posts*. Users can also *like* and add *comments* to those posts.

> **TIP**
> We have highlighted some words in *italic*, these words identify the kind of "things" our model will have to manipulate.

Adding more requirements to our specification:

- A front page displays a feed of recently created posts,
- We can fetch all posts for a user, all comments for a post and all likes for a post,
- Posts are returned with the username of their authors and a count of how many comments and likes they have,
- Comments and likes are also returned with the username of the users who have created them,
- When displayed as lists, posts only have to present a truncated summary of their content.

Identify the main access patterns

To start, we give some structure to our initial specification by identifying our solution's access patterns. When designing a data model for Azure Cosmos DB, it's important to understand which requests our model will have to serve to make sure that the model will serve those requests efficiently.

To make the overall process easier to follow, we categorize those different requests as either commands or queries, borrowing some vocabulary from CQRS where commands are write requests (that is, intents to update the system) and queries are read-only requests.

Here is the list of requests that our platform will have to expose:

- **[C1]** Create/edit a user
- **[Q1]** Retrieve a user
- **[C2]** Create/edit a post

- **[Q2]** Retrieve a post
- **[Q3]** List a user's posts in short form
- **[C3]** Create a comment
- **[Q4]** List a post's comments
- **[C4]** Like a post
- **[Q5]** List a post's likes
- **[Q6]** List the *x* most recent posts created in short form (feed)

At this stage, we haven't thought about the details of what each entity (user, post etc.) will contain. This step is usually among the first ones to be tackled when designing against a relational store, because we have to figure out how those entities will translate in terms of tables, columns, foreign keys etc. It is much less of a concern with a document database that doesn't enforce any schema at write.

The main reason why it is important to identify our access patterns from the beginning, is because this list of requests is going to be our test suite. Every time we iterate over our data model, we will go through each of the requests and check its performance and scalability.

V1: A first version

We start with two containers: `users` and `posts`.

Users container

This container only stores user items:

```
{
    "id": "<user-id>",
    "username": "<username>"
}
```

We partition this container by `id`, which means that each logical partition within that container will only contain one item.

Posts container

This container hosts posts, comments, and likes:

```
{
    "id": "<post-id>",
    "type": "post",
    "postId": "<post-id>",
    "userId": "<post-author-id>",
    "title": "<post-title>",
    "content": "<post-content>",
    "creationDate": "<post-creation-date>"
}

{
    "id": "<comment-id>",
    "type": "comment",
    "postId": "<post-id>",
    "userId": "<comment-author-id>",
    "content": "<comment-content>",
    "creationDate": "<comment-creation-date>"
}

{
    "id": "<like-id>",
    "type": "like",
    "postId": "<post-id>",
    "userId": "<liker-id>",
    "creationDate": "<like-creation-date>"
}
```

We partition this container by `postId`, which means that each logical partition within that container will contain one post, all the comments for that post and all the likes for that post.

Note that we have introduced a `type` property in the items stored in this container to distinguish between the three types of entities that this container hosts.

Also, we have chosen to reference related data instead of embedding it (check this section for details about these concepts) because:

- there's no upper limit to how many posts a user can create,
- posts can be arbitrarily long,
- there's no upper limit to how many comments and likes a post can have,
- we want to be able to add a comment or a like to a post without having to update the post itself.

How well does our model perform?

It's now time to assess the performance and scalability of our first version. For each of the requests previously identified, we measure its latency and how many request units it consumes. This measurement is done against a dummy data set containing 100,000 users with 5 to 50 posts per user, and up to 25 comments and 100 likes per post.

[C1] Create/edit a user

This request is straightforward to implement as we just create or update an item in the `users` container. The requests will nicely spread across all partitions thanks to the `id` partition key.

LATENCY	RU CHARGE	PERFORMANCE
7 ms	5.71 RU	✅

[Q1] Retrieve a user

Retrieving a user is done by reading the corresponding item from the `users` container.

LATENCY	RU CHARGE	PERFORMANCE
2 ms	1 RU	✅

[C2] Create/edit a post

Similarly to **[C1]**, we just have to write to the `posts` container.

LATENCY	RU CHARGE	PERFORMANCE
9 ms	8.76 RU	✅

[Q2] Retrieve a post

We start by retrieving the corresponding document from the `posts` container. But that's not enough, as per our specification we also have to aggregate the username of the post's author and the counts of how many comments and how many likes this post has, which requires 3 additional SQL queries to be issued.

Each of the additional queries filters on the partition key of its respective container, which is exactly what we want to maximize performance and scalability. But we eventually have to perform four operations to return a single post, so we'll improve that in a next iteration.

LATENCY	RU CHARGE	PERFORMANCE
9 ms	19.54 RU	⚠

[Q3] List a user's posts in short form

First, we have to retrieve the desired posts with a SQL query that fetches the posts corresponding to that particular user. But we also have to issue additional queries to aggregate the author's username and the counts of comments and likes.

This implementation presents many drawbacks:

- the queries aggregating the counts of comments and likes have to be issued for each post returned by the first query,
- the main query does not filter on the partition key of the `posts` container, leading to a fan-out and a partition scan across the container.

LATENCY	RU CHARGE	PERFORMANCE
130 ms	619.41 RU	⚠

[C3] Create a comment

A comment is created by writing the corresponding item in the `posts` container.

LATENCY	RU CHARGE	PERFORMANCE
7 ms	8.57 RU	☑

[Q4] List a post's comments

We start with a query that fetches all the comments for that post and once again, we also need to aggregate usernames separately for each comment.

Although the main query does filter on the container's partition key, aggregating the usernames separately penalizes the overall performance. We'll improve that later on.

LATENCY	RU CHARGE	PERFORMANCE
23 ms	27.72 RU	⚠

[C4] Like a post

Just like **[C3]**, we create the corresponding item in the `posts` container.

LATENCY	RU CHARGE	PERFORMANCE
6 ms	7.05 RU	☑

[Q5] List a post's likes

Just like **[Q4]**, we query the likes for that post, then aggregate their usernames.

LATENCY	RU CHARGE	PERFORMANCE
59 ms	58.92 RU	⚠

[Q6] List the x most recent posts created in short form (feed)

We fetch the most recent posts by querying the `posts` container sorted by descending creation date, then aggregate usernames and counts of comments and likes for each of the posts.

Once again, our initial query doesn't filter on the partition key of the `posts` container, which triggers a costly fan-out. This one is even worse as we target a much larger result set and sort the results with an `ORDER BY` clause, which makes it more expensive in terms of request units.

LATENCY	RU CHARGE	PERFORMANCE
306 ms	2063.54 RU	⚠

Reflecting on the performance of V1

Looking at the performance issues we faced in the previous section, we can identify two main classes of problems:

- some requests require multiple queries to be issued in order to gather all the data we need to return,
- some queries don't filter on the partition key of the containers they target, leading to a fan-out that impedes our scalability.

Let's resolve each of those problems, starting with the first one.

V2: Introducing denormalization to optimize read queries

The reason why we have to issue additional requests in some cases is because the results of the initial request doesn't contain all the data we need to return. When working with a non-relational data store like Azure Cosmos DB, this kind of issue is commonly solved by denormalizing data across our data set.

In our example, we modify post items to add the username of the post's author, the count of comments and the count of likes:

```
{
    "id": "<post-id>",
    "type": "post",
    "postId": "<post-id>",
    "userId": "<post-author-id>",
    "userUsername": "<post-author-username>",
    "title": "<post-title>",
    "content": "<post-content>",
    "commentCount": <count-of-comments>,
    "likeCount": <count-of-likes>,
    "creationDate": "<post-creation-date>"
}
```

We also modify comment and like items to add the username of the user who has created them:

```
{
    "id": "<comment-id>",
    "type": "comment",
    "postId": "<post-id>",
    "userId": "<comment-author-id>",
    "userUsername": "<comment-author-username>",
    "content": "<comment-content>",
    "creationDate": "<comment-creation-date>"
}

{
    "id": "<like-id>",
    "type": "like",
    "postId": "<post-id>",
    "userId": "<liker-id>",
    "userUsername": "<liker-username>",
    "creationDate": "<like-creation-date>"
}
```

Denormalizing comment and like counts

What we want to achieve is that every time we add a comment or a like, we also increment the `commentCount` or the `likeCount` in the corresponding post. As our `posts` container is partitioned by `postId`, the new item (comment or like) and its corresponding post sit in the same logical partition. As a result, we can use a stored procedure to perform that operation.

Now when creating a comment ([C3]), instead of just adding a new item in the `posts` container we call the following stored procedure on that container:

```
function createComment(postId, comment) {
  var collection = getContext().getCollection();

  collection.readDocument(
    `${collection.getAltLink()}/docs/${postId}`,
    function (err, post) {
      if (err) throw err;

      post.commentCount++;
      collection.replaceDocument(
        post._self,
        post,
        function (err) {
          if (err) throw err;

          comment.postId = postId;
          collection.createDocument(
            collection.getSelfLink(),
            comment
          );
        }
      );
    })
}
```

This stored procedure takes the ID of the post and the body of the new comment as parameters, then:

- retrieves the post
- increments the `commentCount`
- replaces the post
- adds the new comment

As stored procedures are executed as atomic transactions, the value of `commentCount` and the actual number of comments will always stay in sync.

We obviously call a similar stored procedure when adding new likes to increment the `likeCount`.

Denormalizing usernames

Usernames require a different approach as users not only sit in different partitions, but in a different container. When we have to denormalize data across partitions and containers, we can use the source container's change feed.

In our example, we use the change feed of the `users` container to react whenever users update their usernames. When that happens, we propagate the change by calling another stored procedure on the `posts` container:

```
function updateUsernames(userId, username) {
  var collection = getContext().getCollection();

  collection.queryDocuments(
    collection.getSelfLink(),
    `SELECT * FROM p WHERE p.userId = '${userId}'`,
    function (err, results) {
      if (err) throw err;

      for (var i in results) {
        var doc = results[i];
        doc.userUsername = username;

        collection.upsertDocument(
          collection.getSelfLink(),
          doc);
      }
    });
}
```

This stored procedure takes the ID of the user and the user's new username as parameters, then:

- fetches all items matching the `userId` (which can be posts, comments, or likes)
- for each of those items
  - replaces the `userUsername`
  - replaces the item

> **IMPORTANT**
> This operation is costly because it requires this stored procedure to be executed on every partition of the `posts` container. We assume that most users choose a suitable username during sign-up and won't ever change it, so this update will run very rarely.

What are the performance gains of V2?

[Q2] Retrieve a post

Now that our denormalization is in place, we only have to fetch a single item to handle that request.

LATENCY	RU CHARGE	PERFORMANCE
2 ms	1 RU	☑

[Q4] List a post's comments

Here again, we can spare the extra requests that fetched the usernames and end up with a single query that filters on the partition key.

LATENCY	RU CHARGE	PERFORMANCE
4 ms	7.72 RU	☑

[Q5] List a post's likes

Exact same situation when listing the likes.

LATENCY	RU CHARGE	PERFORMANCE
4 ms	8.92 RU	☑

V3: Making sure all requests are scalable

Looking at our overall performance improvements, there are still two requests that we haven't fully optimized: **[Q3]** and **[Q6]**. They are the requests involving queries that don't filter on the partition key of the containers they target.

[Q3] List a user's posts in short form

This request already benefits from the improvements introduced in V2, which spares additional queries.

But the remaining query is still not filtering on the partition key of the `posts` container.

The way to think about this situation is actually simple:

1. This request *has* to filter on the `userId` because we want to fetch all posts for a particular user
2. It doesn't perform well because it is executed against the `posts` container, which is not partitioned by `userId`
3. Stating the obvious, we would solve our performance problem by executing this request against a container that *is* partitioned by `userId`
4. It turns out that we already have such a container: the `users` container!

So we introduce a second level of denormalization by duplicating entire posts to the `users` container. By doing that, we effectively get a copy of our posts, only partitioned along a different dimensions, making them way more efficient to retrieve by their `userId`.

The `users` container now contains 2 kinds of items:

```
{
    "id": "<user-id>",
    "type": "user",
    "userId": "<user-id>",
    "username": "<username>"
}

{
    "id": "<post-id>",
    "type": "post",
    "postId": "<post-id>",
    "userId": "<post-author-id>",
    "userUsername": "<post-author-username>",
    "title": "<post-title>",
    "content": "<post-content>",
    "commentCount": <count-of-comments>,
    "likeCount": <count-of-likes>,
    "creationDate": "<post-creation-date>"
}
```

Note that:

- we have introduced a `type` field in the user item to distinguish users from posts,
- we have also added a `userId` field in the user item, which is redundant with the `id` field but is required as the `users` container is now partitioned by `userId` (and not `id` as previously)

To achieve that denormalization, we once again use the change feed. This time, we react on the change feed of the `posts` container to dispatch any new or updated post to the `users` container. And because listing posts does not require to return their full content, we can truncate them in the process.

We can now route our query to the `users` container, filtering on the container's partition key.

LATENCY	RU CHARGE	PERFORMANCE
4 ms	6.46 RU	☑

[Q6] List the x most recent posts created in short form (feed)

We have to deal with a similar situation here: even after sparing the additional queries left unnecessary by the denormalization introduced in V2, the remaining query does not filter on the container's partition key:

Following the same approach, maximizing this request's performance and scalability requires that it only hits one partition. This is conceivable because we only have to return a limited number of items; in order to populate our blogging platform's home page, we just need to get the 100 most recent posts, without the need to paginate through the entire data set.

So to optimize this last request, we introduce a third container to our design, entirely dedicated to serving this

request. We denormalize our posts to that new `feed` container:

```
{
    "id": "<post-id>",
    "type": "post",
    "postId": "<post-id>",
    "userId": "<post-author-id>",
    "userUsername": "<post-author-username>",
    "title": "<post-title>",
    "content": "<post-content>",
    "commentCount": <count-of-comments>,
    "likeCount": <count-of-likes>,
    "creationDate": "<post-creation-date>"
}
```

This container is partitioned by `type`, which will always be `post` in our items. Doing that ensures that all the items in this container will sit in the same partition.

To achieve the denormalization, we just have to hook on the change feed pipeline we have previously introduced to dispatch the posts to that new container. One important thing to bear in mind is that we need to make sure that we only store the 100 most recent posts; otherwise, the content of the container may grow beyond the maximum size of a partition. This is done by calling a post-trigger every time a document is added in the container:

Here's the body of the post-trigger that truncates the collection:

```
function truncateFeed() {
  const maxDocs = 100;
  var context = getContext();
  var collection = context.getCollection();

  collection.queryDocuments(
    collection.getSelfLink(),
    "SELECT VALUE COUNT(1) FROM f",
    function (err, results) {
      if (err) throw err;

      processCountResults(results);
    });

  function processCountResults(results) {
    // + 1 because the query didn't count the newly inserted doc
    if ((results[0] + 1) > maxDocs) {
      var docsToRemove = results[0] + 1 - maxDocs;
      collection.queryDocuments(
        collection.getSelfLink(),
        `SELECT TOP ${docsToRemove} * FROM f ORDER BY f.creationDate`,
        function (err, results) {
          if (err) throw err;

          processDocsToRemove(results, 0);
        });
    }
  }

  function processDocsToRemove(results, index) {
    var doc = results[index];
    if (doc) {
      collection.deleteDocument(
        doc._self,
        function (err) {
          if (err) throw err;

          processDocsToRemove(results, index + 1);
        });
    }
  }
}
```

The final step is to reroute our query to our new `feed` container:

LATENCY	RU CHARGE	PERFORMANCE
9 ms	16.97 RU	☑

Conclusion

Let's have a look at the overall performance and scalability improvements we have introduced over the different

versions of our design.

	V1	V2	V3
[C1]	7 ms / 5.71 RU	7 ms / 5.71 RU	7 ms / 5.71 RU
[Q1]	2 ms / 1 RU	2 ms / 1 RU	2 ms / 1 RU
[C2]	9 ms / 8.76 RU	9 ms / 8.76 RU	9 ms / 8.76 RU
[Q2]	9 ms / 19.54 RU	2 ms / 1 RU	2 ms / 1 RU
[Q3]	130 ms / 619.41 RU	28 ms / 201.54 RU	4 ms / 6.46 RU
[C3]	7 ms / 8.57 RU	7 ms / 15.27 RU	7 ms / 15.27 RU
[Q4]	23 ms / 27.72 RU	4 ms / 7.72 RU	4 ms / 7.72 RU
[C4]	6 ms / 7.05 RU	7 ms / 14.67 RU	7 ms / 14.67 RU
[Q5]	59 ms / 58.92 RU	4 ms / 8.92 RU	4 ms / 8.92 RU
[Q6]	306 ms / 2063.54 RU	83 ms / 532.33 RU	9 ms / 16.97 RU

We have optimized a read-heavy scenario

You may have noticed that we have concentrated our efforts towards improving the performance of read requests (queries) at the expense of write requests (commands). In many cases, write operations now trigger subsequent denormalization through change feeds, which makes them more computationally expensive and longer to materialize.

This is justified by the fact that a blogging platform (like most social apps) is read-heavy, which means that the amount of read requests it has to serve is usually orders of magnitude higher than the amount of write requests. So it makes sense to make write requests more expensive to execute in order to let read requests be cheaper and better performing.

If we look at the most extreme optimization we have done, **[Q6]** went from 2000+ RUs to just 17 RUs; we have achieved that by denormalizing posts at a cost of around 10 RUs per item. As we would serve a lot more feed requests than creation or updates of posts, the cost of this denormalization is negligible considering the overall savings.

Denormalization can be applied incrementally

The scalability improvements we've explored in this article involve denormalization and duplication of data across the data set. It should be noted that these optimizations don't have to be put in place at day 1. Queries that filter on partition keys perform better at scale, but cross-partition queries can be totally acceptable if they are called rarely or against a limited data set. If you're just building a prototype, or launching a product with a small and controlled user base, you can probably spare those improvements for later; what's important then is to monitor your model's performance so you can decide if and when it's time to bring them in.

The change feed that we use to distribute updates to other containers store all those updates persistently. This makes it possible to request all updates since the creation of the container and bootstrap denormalized views as a one-time catch-up operation even if your system already has a lot of data.

Next steps

After this introduction to practical data modeling and partitioning, you may want to check the following articles to review the concepts we have covered:

- Work with databases, containers, and items
- Partitioning in Azure Cosmos DB
- Change feed in Azure Cosmos DB

Create a synthetic partition key

12/4/2020 • 3 minutes to read • Edit Online

APPLIES TO: SQL API

It's the best practice to have a partition key with many distinct values, such as hundreds or thousands. The goal is to distribute your data and workload evenly across the items associated with these partition key values. If such a property doesn't exist in your data, you can construct a *synthetic partition key*. This document describes several basic techniques for generating a synthetic partition key for your Cosmos container.

Concatenate multiple properties of an item

You can form a partition key by concatenating multiple property values into a single artificial `partitionKey` property. These keys are referred to as synthetic keys. For example, consider the following example document:

```
{
"deviceId": "abc-123",
"date": 2018
}
```

For the previous document, one option is to set /deviceId or /date as the partition key. Use this option, if you want to partition your container based on either device ID or date. Another option is to concatenate these two values into a synthetic `partitionKey` property that's used as the partition key.

```
{
"deviceId": "abc-123",
"date": 2018,
"partitionKey": "abc-123-2018"
}
```

In real-time scenarios, you can have thousands of items in a database. Instead of adding the synthetic key manually, define client-side logic to concatenate values and insert the synthetic key into the items in your Cosmos containers.

Use a partition key with a random suffix

Another possible strategy to distribute the workload more evenly is to append a random number at the end of the partition key value. When you distribute items in this way, you can perform parallel write operations across partitions.

An example is if a partition key represents a date. You might choose a random number between 1 and 400 and concatenate it as a suffix to the date. This method results in partition key values like `2018-08-09.1`, `2018-08-09.2`, and so on, through `2018-08-09.400`. Because you randomize the partition key, the write operations on the container on each day are spread evenly across multiple partitions. This method results in better parallelism and overall higher throughput.

Use a partition key with pre-calculated suffixes

The random suffix strategy can greatly improve write throughput, but it's difficult to read a specific item. You don't know the suffix value that was used when you wrote the item. To make it easier to read individual items, use the pre-calculated suffixes strategy. Instead of using a random number to distribute the items among the partitions,

use a number that is calculated based on something that you want to query.

Consider the previous example, where a container uses a date as the partition key. Now suppose that each item has a `Vehicle-Identification-Number` (`VIN`) attribute that we want to access. Further, suppose that you often run queries to find items by the `VIN`, in addition to date. Before your application writes the item to the container, it can calculate a hash suffix based on the VIN and append it to the partition key date. The calculation might generate a number between 1 and 400 that is evenly distributed. This result is similar to the results produced by the random suffix strategy method. The partition key value is then the date concatenated with the calculated result.

With this strategy, the writes are evenly spread across the partition key values, and across the partitions. You can easily read a particular item and date, because you can calculate the partition key value for a specific `Vehicle-Identification-Number`. The benefit of this method is that you can avoid creating a single hot partition key, i.e., a partition key that takes all the workload.

Next steps

You can learn more about the partitioning concept in the following articles:

- Learn more about logical partitions.
- Learn more about how to provision throughput on Azure Cosmos containers and databases.
- Learn how to provision throughput on an Azure Cosmos container.
- Learn how to provision throughput on an Azure Cosmos database.

Getting started with SQL queries

12/4/2020 • 4 minutes to read • Edit Online

APPLIES TO: SQL API

In Azure Cosmos DB SQL API accounts, there are two ways to read data:

Point reads - You can do a key/value lookup on a single *item ID* and partition key. The *item ID* and partition key combination is the key and the item itself is the value. For a 1 KB document, point reads typically cost 1 request unit with a latency under 10 ms. Point reads return a single item.

Here are some examples of how to do **Point reads** with each SDK:

- .NET SDK
- Java SDK
- Node.js SDK
- Python SDK

SQL queries - You can query data by writing queries using the Structured Query Language (SQL) as a JSON query language. Queries always cost at least 2.3 request units and, in general, will have a higher and more variable latency than point reads. Queries can return many items.

Most read-heavy workloads on Azure Cosmos DB use a combination of both point reads and SQL queries. If you just need to read a single item, point reads are cheaper and faster than queries. Point reads don't need to use the query engine to access data and can read the data directly. Of course, it's not possible for all workloads to exclusively read data using point reads, so support of SQL as a query language and schema-agnostic indexing provide a more flexible way to access your data.

Here are some examples of how to do **SQL queries** with each SDK:

- .NET SDK
- Java SDK
- Node.js SDK
- Python SDK

The remainder of this doc shows how to get started writing SQL queries in Azure Cosmos DB. SQL queries can be run through either the SDK or Azure portal.

Upload sample data

In your SQL API Cosmos DB account, open the Data Explorer to create a container called `Families`. After the it is created, use the data structures browser, to find and open it. In your `Families` container, you will see the `Items` option right below the name of the container. Open this option and you'll see a button, in the menu bar in center of the screen, to create a 'New Item'. You will use this feature to create the JSON items below.

Create JSON items

The following 2 JSON items are documents about the Andersen and Wakefield families. They include parents, children and their pets, address, and registration information.

The first item has strings, numbers, Booleans, arrays, and nested properties:

```
{
  "id": "AndersenFamily",
  "lastName": "Andersen",
  "parents": [
     { "firstName": "Thomas" },
     { "firstName": "Mary Kay"}
  ],
  "children": [
     {
         "firstName": "Henriette Thaulow",
         "gender": "female",
         "grade": 5,
         "pets": [{ "givenName": "Fluffy" }]
     }
  ],
  "address": { "state": "WA", "county": "King", "city": "Seattle" },
  "creationDate": 1431620472,
  "isRegistered": true
}
```

The second item uses `givenName` and `familyName` instead of `firstName` and `lastName`:

```
{
  "id": "WakefieldFamily",
  "parents": [
      { "familyName": "Wakefield", "givenName": "Robin" },
      { "familyName": "Miller", "givenName": "Ben" }
  ],
  "children": [
      {
        "familyName": "Merriam",
        "givenName": "Jesse",
        "gender": "female",
        "grade": 1,
        "pets": [
            { "givenName": "Goofy" },
            { "givenName": "Shadow" }
        ]
      },
      {
        "familyName": "Miller",
         "givenName": "Lisa",
         "gender": "female",
         "grade": 8 }
  ],
  "address": { "state": "NY", "county": "Manhattan", "city": "NY" },
  "creationDate": 1431620462,
  "isRegistered": false
}
```

Query the JSON items

Try a few queries against the JSON data to understand some of the key aspects of Azure Cosmos DB's SQL query language.

The following query returns the items where the `id` field matches `AndersenFamily`. Since it's a `SELECT *` query, the output of the query is the complete JSON item. For more information about SELECT syntax, see SELECT statement.

```
    SELECT *
    FROM Families f
    WHERE f.id = "AndersenFamily"
```

The query results are:

```
[{
    "id": "AndersenFamily",
    "lastName": "Andersen",
    "parents": [
       { "firstName": "Thomas" },
       { "firstName": "Mary Kay"}
    ],
    "children": [
       {
           "firstName": "Henriette Thaulow", "gender": "female", "grade": 5,
           "pets": [{ "givenName": "Fluffy" }]
       }
    ],
    "address": { "state": "WA", "county": "King", "city": "Seattle" },
    "creationDate": 1431620472,
    "isRegistered": true
}]
```

The following query reformats the JSON output into a different shape. The query projects a new JSON `Family` object with two selected fields, `Name` and `City`, when the address city is the same as the state. "NY, NY" matches this case.

```
SELECT {"Name":f.id, "City":f.address.city} AS Family
FROM Families f
WHERE f.address.city = f.address.state
```

The query results are:

```
[{
    "Family": {
        "Name": "WakefieldFamily",
        "City": "NY"
    }
}]
```

The following query returns all the given names of children in the family whose `id` matches `WakefieldFamily`, ordered by city.

```
SELECT c.givenName
FROM Families f
JOIN c IN f.children
WHERE f.id = 'WakefieldFamily'
ORDER BY f.address.city ASC
```

The results are:

```
[
  { "givenName": "Jesse" },
  { "givenName": "Lisa"}
]
```

Remarks

The preceding examples show several aspects of the Cosmos DB query language:

- Since SQL API works on JSON values, it deals with tree-shaped entities instead of rows and columns. You can refer to the tree nodes at any arbitrary depth, like `Node1.Node2.Node3…..Nodem`, similar to the two-part reference of `<table>.<column>` in ANSI SQL.
- Because the query language works with schemaless data, the type system must be bound dynamically. The same expression could yield different types on different items. The result of a query is a valid JSON value, but isn't guaranteed to be of a fixed schema.
- Azure Cosmos DB supports strict JSON items only. The type system and expressions are restricted to deal only with JSON types. For more information, see the JSON specification.
- A Cosmos container is a schema-free collection of JSON items. The relations within and across container items are implicitly captured by containment, not by primary key and foreign key relations. This feature is important for the intra-item joins discussed later in this article.

Next steps

- Introduction to Azure Cosmos DB
- Azure Cosmos DB .NET samples
- SELECT clause

SELECT clause in Azure Cosmos DB

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: SQL API

Every query consists of a `SELECT` clause and optional FROM and WHERE clauses, per ANSI SQL standards. Typically, the source in the `FROM` clause is enumerated, and the `WHERE` clause applies a filter on the source to retrieve a subset of JSON items. The `SELECT` clause then projects the requested JSON values in the select list.

Syntax

```
SELECT <select_specification>

<select_specification> ::=
      '*'
      | [DISTINCT] <object_property_list>
      | [DISTINCT] VALUE <scalar_expression> [[ AS ] value_alias]

<object_property_list> ::=
{ <scalar_expression> [ [ AS ] property_alias ] } [ ,...n ]
```

Arguments

- `<select_specification>`

  Properties or value to be selected for the result set.

- `'*'`

  Specifies that the value should be retrieved without making any changes. Specifically if the processed value is an object, all properties will be retrieved.

- `<object_property_list>`

  Specifies the list of properties to be retrieved. Each returned value will be an object with the properties specified.

- `VALUE`

  Specifies that the JSON value should be retrieved instead of the complete JSON object. This, unlike `<property_list>` does not wrap the projected value in an object.

- `DISTINCT`

  Specifies that duplicates of projected properties should be removed.

- `<scalar_expression>`

  Expression representing the value to be computed. See Scalar expressions section for details.

Remarks

The `SELECT *` syntax is only valid if FROM clause has declared exactly one alias. `SELECT *` provides an identity projection, which can be useful if no projection is needed. SELECT * is only valid if FROM clause is specified and

introduced only a single input source.

Both `SELECT <select_list>` and `SELECT *` are "syntactic sugar" and can be alternatively expressed by using simple SELECT statements as shown below.

1. `SELECT * FROM ... AS from_alias ...`

   is equivalent to:

   `SELECT from_alias FROM ... AS from_alias ...`

2. `SELECT <expr1> AS p1, <expr2> AS p2,..., <exprN> AS pN [other clauses...]`

   is equivalent to:

   `SELECT VALUE { p1: <expr1>, p2: <expr2>, ..., pN: <exprN> }[other clauses...]`

Examples

The following SELECT query example returns `address` from `Families` whose `id` matches `AndersenFamily`:

```
SELECT f.address
FROM Families f
WHERE f.id = "AndersenFamily"
```

The results are:

```
[{
  "address": {
    "state": "WA",
    "county": "King",
    "city": "Seattle"
  }
}]
```

Next steps

- Getting started
- Azure Cosmos DB .NET samples
- WHERE clause

FROM clause in Azure Cosmos DB

12/4/2020 • 3 minutes to read • Edit Online

APPLIES TO: SQL API

The FROM (`FROM <from_specification>`) clause is optional, unless the source is filtered or projected later in the query. A query like `SELECT * FROM Families` enumerates over the entire `Families` container. You can also use the special identifier ROOT for the container instead of using the container name.

The `FROM` clause enforces the following rules per query:

- The container can be aliased, such as `SELECT f.id FROM Families AS f` or simply `SELECT f.id FROM Families f`. Here `f` is the alias for `Families`. AS is an optional keyword to alias the identifier.
- Once aliased, the original source name cannot be bound. For example, `SELECT Families.id FROM Families f` is syntactically invalid because the identifier `Families` has been aliased and can't be resolved anymore.
- All referenced properties must be fully qualified, to avoid any ambiguous bindings in the absence of strict schema adherence. For example, `SELECT id FROM Families f` is syntactically invalid because the property `id` isn't bound.

Syntax

```
FROM <from_specification>

<from_specification> ::=
        <from_source> {[ JOIN <from_source>][,...n]}

<from_source> ::=
          <container_expression> [[AS] input_alias]
        | input_alias IN <container_expression>

<container_expression> ::=
        ROOT
     | container_name
     | input_alias
     | <container_expression> '.' property_name
     | <container_expression> '[' "property_name" | array_index ']'
```

Arguments

- `<from_source>`

  Specifies a data source, with or without an alias. If alias is not specified, it will be inferred from the `<container_expression>` using following rules:

- If the expression is a container_name, then container_name will be used as an alias.
- If the expression is `<container_expression>`, then property_name, then property_name will be used as an alias. If the expression is a container_name, then container_name will be used as an alias.
- AS `input_alias`

  Specifies that the `input_alias` is a set of values returned by the underlying container expression.

- `input_alias` IN

  Specifies that the `input_alias` should represent the set of values obtained by iterating over all array elements of each array returned by the underlying container expression. Any value returned by underlying container expression that is not an array is ignored.

- `<container_expression>`

  Specifies the container expression to be used to retrieve the documents.

- `ROOT`

  Specifies that document should be retrieved from the default, currently connected container.

- `container_name`

  Specifies that document should be retrieved from the provided container. The name of the container must match the name of the container currently connected to.

- `input_alias`

  Specifies that document should be retrieved from the other source defined by the provided alias.

- `<container_expression> '.' property_name`

  Specifies that document should be retrieved by accessing the `property_name` property.

- `<container_expression> '[' "property_name" | array_index ']'`

  Specifies that document should be retrieved by accessing the `property_name` property or array_index array element for all documents retrieved by specified container expression.

Remarks

All aliases provided or inferred in the `<from_source>(` s) must be unique. The Syntax `<container_expression>.` property_name is the same as `<container_expression>' ['"property_name"']'`. However, the latter syntax can be used if a property name contains a non-identifier character.

Handling missing properties, missing array elements, and undefined values

If a container expression accesses properties or array elements and that value does not exist, that value will be ignored and not processed further.

Container expression context scoping

A container expression may be container-scoped or document-scoped:

- An expression is container-scoped, if the underlying source of the container expression is either ROOT or `container_name`. Such an expression represents a set of documents retrieved from the container directly, and is not dependent on the processing of other container expressions.

- An expression is document-scoped, if the underlying source of the container expression is `input_alias` introduced earlier in the query. Such an expression represents a set of documents obtained by evaluating the container expression in the scope of each document belonging to the set associated with the aliased container. The resulting set will be a union of sets obtained by evaluating the container expression for each of the documents in the underlying set.

Examples

Get subitems by using the FROM clause

The FROM clause can reduce the source to a smaller subset. To enumerate only a subtree in each item, the subroot

can become the source, as shown in the following example:

```
SELECT *
FROM Families.children
```

The results are:

```
[
  [
    {
        "firstName": "Henriette Thaulow",
        "gender": "female",
        "grade": 5,
        "pets": [
          {
              "givenName": "Fluffy"
          }
        ]
    }
  ],
  [
   {
        "familyName": "Merriam",
        "givenName": "Jesse",
        "gender": "female",
        "grade": 1
    },
    {
        "familyName": "Miller",
        "givenName": "Lisa",
        "gender": "female",
        "grade": 8
    }
  ]
]
```

The preceding query used an array as the source, but you can also use an object as the source. The query considers any valid, defined JSON value in the source for inclusion in the result. The following example would exclude `Families` that don't have an `address.state` value.

```
SELECT *
FROM Families.address.state
```

The results are:

```
[
  "WA",
  "NY"
]
```

Next steps

- Getting started
- SELECT clause
- WHERE clause

WHERE clause in Azure Cosmos DB

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: SQL API

The optional WHERE clause (`WHERE <filter_condition>`) specifies condition(s) that the source JSON items must satisfy for the query to include them in results. A JSON item must evaluate the specified conditions to `true` to be considered for the result. The index layer uses the WHERE clause to determine the smallest subset of source items that can be part of the result.

Syntax

```
WHERE <filter_condition>
<filter_condition> ::= <scalar_expression>
```

Arguments

- `<filter_condition>`

  Specifies the condition to be met for the documents to be returned.

- `<scalar_expression>`

  Expression representing the value to be computed. See Scalar expressions for details.

Remarks

In order for the document to be returned an expression specified as filter condition must evaluate to true. Only Boolean value `true` will satisfy the condition, any other value: undefined, null, false, Number, Array, or Object will not satisfy the condition.

If you include your partition key in the `WHERE` clause as part of an equality filter, your query will automatically filter to only the relevant partitions.

Examples

The following query requests items that contain an `id` property whose value is `AndersenFamily`. It excludes any item that does not have an `id` property or whose value doesn't match `AndersenFamily`.

```
SELECT f.address
FROM Families f
WHERE f.id = "AndersenFamily"
```

The results are:

```
[{
  "address": {
    "state": "WA",
    "county": "King",
    "city": "Seattle"
  }
}]
```

Scalar expressions in the WHERE clause

The previous example showed a simple equality query. The SQL API also supports various scalar expressions. The most commonly used are binary and unary expressions. Property references from the source JSON object are also valid expressions.

You can use the following supported binary operators:

OPERATOR TYPE	VALUES
Arithmetic	+,-,*,/,%
Bitwise	|, &, ^, <<, >>, >>> (zero-fill right shift)
Logical	AND, OR, NOT
Comparison	=, !=, <, >, <=, >=, <>
String	|| (concatenate)

The following queries use binary operators:

```
SELECT *
FROM Families.children[0] c
WHERE c.grade % 2 = 1     -- matching grades == 5, 1

SELECT *
FROM Families.children[0] c
WHERE c.grade ^ 4 = 1    -- matching grades == 5

SELECT *
FROM Families.children[0] c
WHERE c.grade >= 5    -- matching grades == 5
```

You can also use the unary operators +,-, ~, and NOT in queries, as shown in the following examples:

```
SELECT *
FROM Families.children[0] c
WHERE NOT(c.grade = 5)  -- matching grades == 1

SELECT *
FROM Families.children[0] c
WHERE (-c.grade = -5)  -- matching grades == 5
```

You can also use property references in queries. For example, `SELECT * FROM Families f WHERE f.isRegistered` returns the JSON item containing the property `isRegistered` with value equal to `true`. Any other value, such as `false`, `null`, `Undefined`, `<number>`, `<string>`, `<object>`, or `<array>`, excludes the item from the result. Additionally, you can use the `IS_DEFINED` type checking function to query based on the presence or absence of a given JSON property. For instance, `SELECT * FROM Families f WHERE NOT IS_DEFINED(f.isRegistered)` returns any

JSON item that does not have a value for `isRegistered`.

Next steps

- Getting started
- IN keyword
- FROM clause

ORDER BY clause in Azure Cosmos DB

12/4/2020 • 5 minutes to read • Edit Online

APPLIES TO: SQL API

The optional `ORDER BY` clause specifies the sorting order for results returned by the query.

Syntax

```
ORDER BY <sort_specification>
<sort_specification> ::= <sort_expression> [, <sort_expression>]
<sort_expression> ::= {<scalar_expression> [ASC | DESC]} [ ,...n ]
```

Arguments

- `<sort_specification>`

  Specifies a property or expression on which to sort the query result set. A sort column can be specified as a name or property alias.

  Multiple properties can be specified. Property names must be unique. The sequence of the sort properties in the `ORDER BY` clause defines the organization of the sorted result set. That is, the result set is sorted by the first property and then that ordered list is sorted by the second property, and so on.

  The property names referenced in the `ORDER BY` clause must correspond to either a property in the select list or to a property defined in the collection specified in the `FROM` clause without any ambiguities.

- `<sort_expression>`

  Specifies one or more properties or expressions on which to sort the query result set.

- `<scalar_expression>`

  See the Scalar expressions section for details.

- `ASC | DESC`

  Specifies that the values in the specified column should be sorted in ascending or descending order. `ASC` sorts from the lowest value to highest value. `DESC` sorts from highest value to lowest value. `ASC` is the default sort order. Null values are treated as the lowest possible values.

Remarks

The `ORDER BY` clause requires that the indexing policy include an index for the fields being sorted. The Azure Cosmos DB query runtime supports sorting against a property name and not against computed properties. Azure Cosmos DB supports multiple `ORDER BY` properties. In order to run a query with multiple ORDER BY properties, you should define a composite index on the fields being sorted.

> **NOTE**
> If the properties being sorted might be undefined for some documents and you want to retrieve them in an ORDER BY query, you must explicitly include this path in the index. The default indexing policy won't allow for the retrieval of the documents where the sort property is undefined. Review example queries on documents with some missing fields.

Examples

For example, here's a query that retrieves families in ascending order of the resident city's name:

```
SELECT f.id, f.address.city
FROM Families f
ORDER BY f.address.city
```

The results are:

```
[
  {
    "id": "WakefieldFamily",
    "city": "NY"
  },
  {
    "id": "AndersenFamily",
    "city": "Seattle"
  }
]
```

The following query retrieves family `id`s in order of their item creation date. Item `creationDate` is a number representing the *epoch time*, or elapsed time since Jan. 1, 1970 in seconds.

```
SELECT f.id, f.creationDate
FROM Families f
ORDER BY f.creationDate DESC
```

The results are:

```
[
  {
    "id": "WakefieldFamily",
    "creationDate": 1431620462
  },
  {
    "id": "AndersenFamily",
    "creationDate": 1431620472
  }
]
```

Additionally, you can order by multiple properties. A query that orders by multiple properties requires a composite index. Consider the following query:

```
SELECT f.id, f.creationDate
FROM Families f
ORDER BY f.address.city ASC, f.creationDate DESC
```

This query retrieves the family `id` in ascending order of the city name. If multiple items have the same city name,

the query will order by the `creationDate` in descending order.

Documents with missing fields

Queries with `ORDER BY` that are run against containers with the default indexing policy will not return documents where the sort property is undefined. If you would like to include documents where the sort property is undefined, you should explicitly include this property in the indexing policy.

For example, here's a container with an indexing policy that does not explicitly include any paths besides `"/*"`:

```
{
    "indexingMode": "consistent",
    "automatic": true,
    "includedPaths": [
        {
            "path": "/*"
        }
    ],
    "excludedPaths": []
}
```

If you run a query that includes `lastName` in the `Order By` clause, the results will only include documents that have a `lastName` property defined. We have not defined an explicit included path for `lastName` so any documents without a `lastName` will not appear in the query results.

Here is a query that sorts by `lastName` on two documents, one of which does not have a `lastName` defined:

```
SELECT f.id, f.lastName
FROM Families f
ORDER BY f.lastName
```

The results only include the document that has a defined `lastName`:

```
[
    {
        "id": "AndersenFamily",
        "lastName": "Andersen"
    }
]
```

If we update the container's indexing policy to explicitly include a path for `lastName`, we will include documents with an undefined sort property in the query results. You must explicitly define the path to lead to this scalar value (and not beyond it). You should use the `?` character in your path definition in the indexing policy to ensure that you explicitly index the property `lastName` and no additional nested paths beyond it. If your `Order By` query uses a composite index, the results will always include documents with an undefined sort property in the query results.

Here is a sample indexing policy which allows you to have documents with an undefined `lastName` appear in the query results:

```
{
    "indexingMode": "consistent",
    "automatic": true,
    "includedPaths": [
        {
            "path": "/lastName/?"
        },
        {
            "path": "/*"
        }
    ],
    "excludedPaths": []
}
```

If you run the same query again, documents that are missing `lastName` appear first in the query results:

```
    SELECT f.id, f.lastName
    FROM Families f
    ORDER BY f.lastName
```

The results are:

```
[
    {
        "id": "WakefieldFamily"
    },
    {
        "id": "AndersenFamily",
        "lastName": "Andersen"
    }
]
```

If you modify the sort order to `DESC`, documents that are missing `lastName` appear last in the query results:

```
    SELECT f.id, f.lastName
    FROM Families f
    ORDER BY f.lastName DESC
```

The results are:

```
[
    {
        "id": "AndersenFamily",
        "lastName": "Andersen"
    },
    {
        "id": "WakefieldFamily"
    }
]
```

NOTE

Only the .NET SDK version 3.4.0 or later supports ORDER BY with mixed types. Therefore, if you want to sort by a combination of undefined and defined values, you should use this version (or later).

You can't control the order that different types appear in the results. In the above example, we showed how undefined values were sorted before string values. If instead, for example, you wanted more control over the sort

order of undefined values, you could assign any undefined properties a string value of "aaaaaaaaa" or "zzzzzzzz" to ensure they were either first or last.

Next steps

- Getting started
- Indexing policies in Azure Cosmos DB
- OFFSET LIMIT clause

GROUP BY clause in Azure Cosmos DB

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: SQL API

The GROUP BY clause divides the query's results according to the values of one or more specified properties.

Syntax

```
<group_by_clause> ::= GROUP BY <scalar_expression_list>

<scalar_expression_list> ::=
          <scalar_expression>
        | <scalar_expression_list>, <scalar_expression>
```

Arguments

- `<scalar_expression_list>`

  Specifies the expressions that will be used to divide query results.

- `<scalar_expression>`

  Any scalar expression is allowed except for scalar subqueries and scalar aggregates. Each scalar expression must contain at least one property reference. There is no limit to the number of individual expressions or the cardinality of each expression.

Remarks

When a query uses a GROUP BY clause, the SELECT clause can only contain the subset of properties and system functions included in the GROUP BY clause. One exception is aggregate functions, which can appear in the SELECT clause without being included in the GROUP BY clause. You can also always include literal values in the SELECT clause.

The GROUP BY clause must be after the SELECT, FROM, and WHERE clause and before the OFFSET LIMIT clause. You currently cannot use GROUP BY with an ORDER BY clause but this is planned.

The GROUP BY clause does not allow any of the following:

- Aliasing properties or aliasing system functions (aliasing is still allowed within the SELECT clause)
- Subqueries
- Aggregate system functions (these are only allowed in the SELECT clause)

Queries with an aggregate system function and a subquery with `GROUP BY` are not supported. For example, the following query is not supported:

```
SELECT COUNT(UniqueLastNames)
FROM (
SELECT AVG(f.age)
FROM f
GROUP BY f.lastName
) AS UniqueLastNames
```

Examples

These examples use the nutrition data set available through the Azure Cosmos DB Query Playground.

For example, here's a query which returns the total count of items in each foodGroup:

```
SELECT TOP 4 COUNT(1) AS foodGroupCount, f.foodGroup
FROM Food f
GROUP BY f.foodGroup
```

Some results are (TOP keyword is used to limit results):

```
[
    {
        "foodGroupCount": 183,
        "foodGroup": "Cereal Grains and Pasta"
    },
    {
        "foodGroupCount": 133,
        "foodGroup": "Nut and Seed Products"
    },
    {
        "foodGroupCount": 113,
        "foodGroup": "Meals, Entrees, and Side Dishes"
    },
    {
        "foodGroupCount": 64,
        "foodGroup": "Spices and Herbs"
    }
]
```

This query has two expressions used to divide results:

```
SELECT TOP 4 COUNT(1) AS foodGroupCount, f.foodGroup, f.version
FROM Food f
GROUP BY f.foodGroup, f.version
```

Some results are:

```
[
    {
        "foodGroupCount": 183,
        "foodGroup": "Cereal Grains and Pasta",
        "version": 1
    },
    {
        "foodGroupCount": 133,
        "foodGroup": "Nut and Seed Products",
        "version": 1
    },
    {
        "foodGroupCount": 113,
        "foodGroup": "Meals, Entrees, and Side Dishes",
        "version": 1
    },
    {
        "foodGroupCount": 64,
        "foodGroup": "Spices and Herbs",
        "version": 1
    }
]
```

This query has a system function in the GROUP BY clause:

```
SELECT TOP 4 COUNT(1) AS foodGroupCount, UPPER(f.foodGroup) AS upperFoodGroup
FROM Food f
GROUP BY UPPER(f.foodGroup)
```

Some results are:

```
[
    {
        "foodGroupCount": 183,
        "upperFoodGroup": "CEREAL GRAINS AND PASTA"
    },
    {
        "foodGroupCount": 133,
        "upperFoodGroup": "NUT AND SEED PRODUCTS"
    },
    {
        "foodGroupCount": 113,
        "upperFoodGroup": "MEALS, ENTREES, AND SIDE DISHES"
    },
    {
        "foodGroupCount": 64,
        "upperFoodGroup": "SPICES AND HERBS"
    }
]
```

This query uses both keywords and system functions in the item property expression:

```
SELECT COUNT(1) AS foodGroupCount, ARRAY_CONTAINS(f.tags, {name: 'orange'}) AS containsOrangeTag,  f.version
BETWEEN 0 AND 2 AS correctVersion
FROM Food f
GROUP BY ARRAY_CONTAINS(f.tags, {name: 'orange'}), f.version BETWEEN 0 AND 2
```

The results are:

```
[
    {
        "foodGroupCount": 10,
        "containsOrangeTag": true,
        "correctVersion": true
    },
    {
        "foodGroupCount": 8608,
        "containsOrangeTag": false,
        "correctVersion": true
    }
]
```

Next steps

- Getting started
- SELECT clause
- Aggregate functions

OFFSET LIMIT clause in Azure Cosmos DB

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: SQL API

The OFFSET LIMIT clause is an optional clause to skip then take some number of values from the query. The OFFSET count and the LIMIT count are required in the OFFSET LIMIT clause.

When OFFSET LIMIT is used in conjunction with an ORDER BY clause, the result set is produced by doing skip and take on the ordered values. If no ORDER BY clause is used, it will result in a deterministic order of values.

Syntax

```
OFFSET <offset_amount> LIMIT <limit_amount>
```

Arguments

- `<offset_amount>`

  Specifies the integer number of items that the query results should skip.

- `<limit_amount>`

  Specifies the integer number of items that the query results should include

Remarks

Both the `OFFSET` count and the `LIMIT` count are required in the `OFFSET LIMIT` clause. If an optional `ORDER BY` clause is used, the result set is produced by doing the skip over the ordered values. Otherwise, the query will return a fixed order of values.

The RU charge of a query with `OFFSET LIMIT` will increase as the number of terms being offset increases. For queries that have multiple pages of results, we typically recommend using continuation tokens. Continuation tokens are a "bookmark" for the place where the query can later resume. If you use `OFFSET LIMIT`, there is no "bookmark". If you wanted to return the query's next page, you would have to start from the beginning.

You should use `OFFSET LIMIT` for cases when you would like to skip items entirely and save client resources. For example, you should use `OFFSET LIMIT` if you want to skip to the 1000th query result and have no need to view results 1 through 999. On the backend, `OFFSET LIMIT` still loads each item, including those that are skipped. The performance advantage is a savings in client resources by avoiding processing items that are not needed.

Examples

For example, here's a query that skips the first value and returns the second value (in order of the resident city's name):

```
SELECT f.id, f.address.city
FROM Families f
ORDER BY f.address.city
OFFSET 1 LIMIT 1
```

The results are:

```
[
  {
    "id": "AndersenFamily",
    "city": "Seattle"
  }
]
```

Here's a query that skips the first value and returns the second value (without ordering):

```
SELECT f.id, f.address.city
FROM Families f
OFFSET 1 LIMIT 1
```

The results are:

```
[
  {
    "id": "WakefieldFamily",
    "city": "Seattle"
  }
]
```

Next steps

- Getting started
- SELECT clause
- ORDER BY clause

Working with JSON in Azure Cosmos DB

12/4/2020 • 4 minutes to read • Edit Online

APPLIES TO: SQL API

In Azure Cosmos DB's SQL (Core) API, items are stored as JSON. The type system and expressions are restricted to deal only with JSON types. For more information, see the JSON specification.

We'll summarize some important aspects of working with JSON:

- JSON objects always begin with a `{` left brace and end with a `}` right brace
- You can have JSON properties nested within one another
- JSON property values can be arrays
- JSON property names are case sensitive
- JSON property name can be any string value (including spaces or characters that aren't letters)

Nested properties

You can access nested JSON using a dot accessor. You can use nested JSON properties in your queries the same way that you can use any other properties.

Here's a document with nested JSON:

```
{
  "id": "AndersenFamily",
  "lastName": "Andersen",
  "address": {
      "state": "WA",
      "county": "King",
      "city": "Seattle"
      },
  "creationDate": 1431620472,
  "isRegistered": true
}
```

In this case, the `state`, `country`, and `city` properties are all nested within the `address` property.

The following example projects two nested properties: `f.address.state` and `f.address.city`.

```
SELECT f.address.state, f.address.city
FROM Families f
WHERE f.id = "AndersenFamily"
```

The results are:

```
[{
  "state": "WA",
  "city": "Seattle"
}]
```

Working with arrays

In addition to nested properties, JSON also supports arrays.

Here's an example document with an array:

```
{
  "id": "WakefieldFamily",
  "children": [
      {
        "familyName": "Merriam",
        "givenName": "Jesse",
        "gender": "female",
        "grade": 1,
      },
      {
        "familyName": "Miller",
         "givenName": "Lisa",
         "gender": "female",
         "grade": 8
      }
  ],
}
```

When working with arrays, you can access a specific element within the array by referencing its position:

```
SELECT *
FROM Families f
WHERE f.children[0].givenName = "Jesse"
```

In most cases, however, you'll use a subquery or self-join when working with arrays.

For example, here's a document that shows the daily balance of a customer's bank account.

```
{
  "id": "Contoso-Checking-Account-2020",
  "balance": [
      {
        "checkingAccount": 1000,
        "savingsAccount": 5000
      },
      {
        "checkingAccount": 100,
        "savingsAccount": 5000
      },
      {
        "checkingAccount": -10,
        "savingsAccount": 5000,
      },
      {
        "checkingAccount": 5000,
        "savingsAccount": 5000,
      }
         ...
  ]
}
```

If you wanted to run a query that showed all the customers that had a negative balance at some point, you could use EXISTS with a subquery:

```
SELECT c.id
FROM c
WHERE EXISTS(
    SELECT VALUE n
    FROM n IN c.balance
    WHERE n.checkingAccount < 0
)
```

Difference between null and undefined

If a property is not defined in an item, then its value is `undefined`. A property with the value `null` must be explicitly defined and assigned a `null` value.

For example, consider this sample item:

```
{
  "id": "AndersenFamily",
  "lastName": "Andersen",
  "address": {
      "state": "WA",
      "county": "King",
      "city": "Seattle"
      },
  "creationDate": null
}
```

In this example, the property `isRegistered` has a value of `undefined` because it is omitted from the item. The property `creationDate` has a `null` value.

Azure Cosmos DB supports two helpful type checking system functions for `null` and `undefined` properties:

- IS_NULL - checks if a property value is `null`
- IS_DEFINED - checks if a property value is defined

You can learn about supported operators and their behavior for `null` and `undefined` values.

Reserved keywords and special characters in JSON

You can access properties using the quoted property operator `[]`. For example, `SELECT c.grade` and `SELECT c["grade"]` are equivalent. This syntax is useful to escape a property that contains spaces, special characters, or has the same name as a SQL keyword or reserved word.

For example, here's a document with a property named `order` and a property `price($)` that contains special characters:

```
{
  "id": "AndersenFamily",
  "order": {
         "orderId": "12345",
         "productId": "A17849",
         "price($)": 59.33
   },
  "creationDate": 1431620472,
  "isRegistered": true
}
```

If you run a queries that includes the `order` property or `price($)` property, you will receive a syntax error.

```
SELECT * FROM c where c.order.orderid = "12345"
```

```
SELECT * FROM c where c.order.price($) > 50
```

The result is:

`Syntax error, incorrect syntax near 'order'`

You should rewrite the same queries as below:

```
SELECT * FROM c WHERE c["order"].orderId = "12345"
```

```
SELECT * FROM c WHERE c["order"]["price($)"] > 50
```

JSON expressions

Projection also supports JSON expressions, as shown in the following example:

```
    SELECT { "state": f.address.state, "city": f.address.city, "name": f.id }
    FROM Families f
    WHERE f.id = "AndersenFamily"
```

The results are:

```
    [{
      "$1": {
        "state": "WA",
        "city": "Seattle",
        "name": "AndersenFamily"
      }
    }]
```

In the preceding example, the `SELECT` clause needs to create a JSON object, and since the sample provides no key, the clause uses the implicit argument variable name `$1`. The following query returns two implicit argument variables: `$1` and `$2`.

```
    SELECT { "state": f.address.state, "city": f.address.city },
           { "name": f.id }
    FROM Families f
    WHERE f.id = "AndersenFamily"
```

The results are:

```
    [{
      "$1": {
        "state": "WA",
        "city": "Seattle"
      },
      "$2": {
        "name": "AndersenFamily"
      }
    }]
```

Aliasing

You can explicitly alias values in queries. If a query has two properties with the same name, use aliasing to rename one or both of the properties so they're disambiguated in the projected result.

Examples

The `AS` keyword used for aliasing is optional, as shown in the following example when projecting the second value as `NameInfo`:

```
SELECT
       { "state": f.address.state, "city": f.address.city } AS AddressInfo,
       { "name": f.id } NameInfo
FROM Families f
WHERE f.id = "AndersenFamily"
```

The results are:

```
[{
  "AddressInfo": {
    "state": "WA",
    "city": "Seattle"
  },
  "NameInfo": {
    "name": "AndersenFamily"
  }
}]
```

Aliasing with reserved keywords or special characters

You can't use aliasing to project a value as a property name with a space, special character, or reserved word. If you wanted to change a value's projection to, for example, have a property name with a space, you could use a JSON expression.

Here's an example:

```
SELECT
       {"JSON expression with a space": { "state": f.address.state, "city": f.address.city }},
       {"JSON expression with a special character!": { "name": f.id }}
FROM Families f
WHERE f.id = "AndersenFamily"
```

Next steps

- Getting started
- SELECT clause
- WHERE clause

SQL subquery examples for Azure Cosmos DB

12/4/2020 • 12 minutes to read • Edit Online

APPLIES TO: SQL API

A subquery is a query nested within another query. A subquery is also called an inner query or inner select. The statement that contains a subquery is typically called an outer query.

This article describes SQL subqueries and their common use cases in Azure Cosmos DB. All sample queries in this doc can be run against a nutrition dataset that is preloaded on the Azure Cosmos DB Query Playground.

Types of subqueries

There are two main types of subqueries:

- **Correlated**: A subquery that references values from the outer query. The subquery is evaluated once for each row that the outer query processes.
- **Non-correlated**: A subquery that's independent of the outer query. It can be run on its own without relying on the outer query.

> **NOTE**
> Azure Cosmos DB supports only correlated subqueries.

Subqueries can be further classified based on the number of rows and columns that they return. There are three types:

- **Table**: Returns multiple rows and multiple columns.
- **Multi-value**: Returns multiple rows and a single column.
- **Scalar**: Returns a single row and a single column.

SQL queries in Azure Cosmos DB always return a single column (either a simple value or a complex document). Therefore, only multi-value and scalar subqueries are applicable in Azure Cosmos DB. You can use a multi-value subquery only in the FROM clause as a relational expression. You can use a scalar subquery as a scalar expression in the SELECT or WHERE clause, or as a relational expression in the FROM clause.

Multi-value subqueries

Multi-value subqueries return a set of documents and are always used within the FROM clause. They're used for:

- Optimizing JOIN expressions.
- Evaluating expensive expressions once and referencing multiple times.

Optimize JOIN expressions

Multi-value subqueries can optimize JOIN expressions by pushing predicates after each select-many expression rather than after all cross-joins in the WHERE clause.

Consider the following query:

```
SELECT Count(1) AS Count
FROM c
JOIN t IN c.tags
JOIN n IN c.nutrients
JOIN s IN c.servings
WHERE t.name = 'infant formula' AND (n.nutritionValue > 0
AND n.nutritionValue < 10) AND s.amount > 1
```

For this query, the index will match any document that has a tag with the name "infant formula." It's a nutrient item with a value between 0 and 10, and a serving item with an amount greater than 1. The JOIN expression here will perform the cross-product of all items of tags, nutrients, and servings arrays for each matching document before any filter is applied.

The WHERE clause will then apply the filter predicate on each <c, t, n, s> tuple. For instance, if a matching document had 10 items in each of the three arrays, it will expand to 1 x 10 x 10 x 10 (that is, 1,000) tuples. Using subqueries here can help in filtering out joined array items before joining with the next expression.

This query is equivalent to the preceding one but uses subqueries:

```
SELECT Count(1) AS Count
FROM c
JOIN (SELECT VALUE t FROM t IN c.tags WHERE t.name = 'infant formula')
JOIN (SELECT VALUE n FROM n IN c.nutrients WHERE n.nutritionValue > 0 AND n.nutritionValue < 10)
JOIN (SELECT VALUE s FROM s IN c.servings WHERE s.amount > 1)
```

Assume that only one item in the tags array matches the filter, and there are five items for both nutrients and servings arrays. The JOIN expressions will then expand to 1 x 1 x 5 x 5 = 25 items, as opposed to 1,000 items in the first query.

Evaluate once and reference many times

Subqueries can help optimize queries with expensive expressions such as user-defined functions (UDFs), complex strings, or arithmetic expressions. You can use a subquery along with a JOIN expression to evaluate the expression once but reference it many times.

The following query runs the UDF `GetMaxNutritionValue` twice:

```
SELECT c.id, udf.GetMaxNutritionValue(c.nutrients) AS MaxNutritionValue
FROM c
WHERE udf.GetMaxNutritionValue(c.nutrients) > 100
```

Here's an equivalent query that runs the UDF only once:

```
SELECT TOP 1000 c.id, MaxNutritionValue
FROM c
JOIN (SELECT VALUE udf.GetMaxNutritionValue(c.nutrients)) MaxNutritionValue
WHERE MaxNutritionValue > 100
```

> **NOTE**
>
> Keep in mind the cross-product behavior of JOIN expressions. If the UDF expression can evaluate to undefined, you should ensure that the JOIN expression always produces a single row by returning an object from the subquery rather than the value directly.

Here's a similar example that returns an object rather than a value:

```
SELECT TOP 1000 c.id, m.MaxNutritionValue
FROM c
JOIN (SELECT udf.GetMaxNutritionValue(c.nutrients) AS MaxNutritionValue) m
WHERE m.MaxNutritionValue > 100
```

The approach is not limited to UDFs. It applies to any potentially expensive expression. For example, you can take the same approach with the mathematical function `avg`:

```
SELECT TOP 1000 c.id, AvgNutritionValue
FROM c
JOIN (SELECT VALUE avg(n.nutritionValue) FROM n IN c.nutrients) AvgNutritionValue
WHERE AvgNutritionValue > 80
```

Mimic join with external reference data

You might often need to reference static data that rarely changes, such as units of measurement or country codes. It's better not to duplicate such data for each document. Avoiding this duplication will save on storage and improve write performance by keeping the document size smaller. You can use a subquery to mimic inner-join semantics with a collection of reference data.

For instance, consider this set of reference data:

UNIT	NAME	MULTIPLIER	BASE UNIT
ng	Nanogram	1.00E-09	Gram
µg	Microgram	1.00E-06	Gram
mg	Milligram	1.00E-03	Gram
g	Gram	1.00E+00	Gram
kg	Kilogram	1.00E+03	Gram
Mg	Megagram	1.00E+06	Gram
Gg	Gigagram	1.00E+09	Gram
nJ	Nanojoule	1.00E-09	Joule
µJ	Microjoule	1.00E-06	Joule
mJ	Millijoule	1.00E-03	Joule
J	Joule	1.00E+00	Joule
kJ	Kilojoule	1.00E+03	Joule
MJ	Megajoule	1.00E+06	Joule
GJ	Gigajoule	1.00E+09	Joule

UNIT	NAME	MULTIPLIER	BASE UNIT
cal	Calorie	1.00E+00	calorie
kcal	Calorie	1.00E+03	calorie
IU	International units		

The following query mimics joining with this data so that you add the name of the unit to the output:

```
SELECT TOP 10 n.id, n.description, n.nutritionValue, n.units, r.name
FROM food
JOIN n IN food.nutrients
JOIN r IN (
    SELECT VALUE [
        {unit: 'ng', name: 'nanogram', multiplier: 0.000000001, baseUnit: 'gram'},
        {unit: 'µg', name: 'microgram', multiplier: 0.000001, baseUnit: 'gram'},
        {unit: 'mg', name: 'milligram', multiplier: 0.001, baseUnit: 'gram'},
        {unit: 'g', name: 'gram', multiplier: 1, baseUnit: 'gram'},
        {unit: 'kg', name: 'kilogram', multiplier: 1000, baseUnit: 'gram'},
        {unit: 'Mg', name: 'megagram', multiplier: 1000000, baseUnit: 'gram'},
        {unit: 'Gg', name: 'gigagram', multiplier: 1000000000, baseUnit: 'gram'},
        {unit: 'nJ', name: 'nanojoule', multiplier: 0.000000001, baseUnit: 'joule'},
        {unit: 'µJ', name: 'microjoule', multiplier: 0.000001, baseUnit: 'joule'},
        {unit: 'mJ', name: 'millijoule', multiplier: 0.001, baseUnit: 'joule'},
        {unit: 'J', name: 'joule', multiplier: 1, baseUnit: 'joule'},
        {unit: 'kJ', name: 'kilojoule', multiplier: 1000, baseUnit: 'joule'},
        {unit: 'MJ', name: 'megajoule', multiplier: 1000000, baseUnit: 'joule'},
        {unit: 'GJ', name: 'gigajoule', multiplier: 1000000000, baseUnit: 'joule'},
        {unit: 'cal', name: 'calorie', multiplier: 1, baseUnit: 'calorie'},
        {unit: 'kcal', name: 'Calorie', multiplier: 1000, baseUnit: 'calorie'},
        {unit: 'IU', name: 'International units'}
    ]
)
WHERE n.units = r.unit
```

Scalar subqueries

A scalar subquery expression is a subquery that evaluates to a single value. The value of the scalar subquery expression is the value of the projection (SELECT clause) of the subquery. You can use a scalar subquery expression in many places where a scalar expression is valid. For instance, you can use a scalar subquery in any expression in both the SELECT and WHERE clauses.

Using a scalar subquery doesn't always help optimize, though. For example, passing a scalar subquery as an argument to either a system or user-defined functions provides no benefit in resource unit (RU) consumption or latency.

Scalar subqueries can be further classified as:

- Simple-expression scalar subqueries
- Aggregate scalar subqueries

Simple-expression scalar subqueries

A simple-expression scalar subquery is a correlated subquery that has a SELECT clause that doesn't contain any aggregate expressions. These subqueries provide no optimization benefits because the compiler converts them into one larger simple expression. There's no correlated context between the inner and outer queries.

Here are few examples:

Example 1

```
SELECT 1 AS a, 2 AS b
```

You can rewrite this query, by using a simple-expression scalar subquery, to:

```
SELECT (SELECT VALUE 1) AS a, (SELECT VALUE 2) AS b
```

Both queries produce this output:

```
[
  { "a": 1, "b": 2 }
]
```

Example 2

```
SELECT TOP 5 Concat('id_', f.id) AS id
FROM food f
```

You can rewrite this query, by using a simple-expression scalar subquery, to:

```
SELECT TOP 5 (SELECT VALUE Concat('id_', f.id)) AS id
FROM food f
```

Query output:

```
[
  { "id": "id_03226" },
  { "id": "id_03227" },
  { "id": "id_03228" },
  { "id": "id_03229" },
  { "id": "id_03230" }
]
```

Example 3

```
SELECT TOP 5 f.id, Contains(f.description, 'fruit') = true ? f.description : undefined
FROM food f
```

You can rewrite this query, by using a simple-expression scalar subquery, to:

```
SELECT TOP 10 f.id, (SELECT f.description WHERE Contains(f.description, 'fruit')).description
FROM food f
```

Query output:

```
[
  { "id": "03230" },
  { "id": "03238", "description":"Babyfood, dessert, tropical fruit, junior" },
  { "id": "03229" },
  { "id": "03226", "description":"Babyfood, dessert, fruit pudding, orange, strained" },
  { "id": "03227" }
]
```

Aggregate scalar subqueries

An aggregate scalar subquery is a subquery that has an aggregate function in its projection or filter that evaluates to a single value.

Example 1:

Here's a subquery with a single aggregate function expression in its projection:

```
SELECT TOP 5
    f.id,
    (SELECT VALUE Count(1) FROM n IN f.nutrients WHERE n.units = 'mg'
) AS count_mg
FROM food f
```

Query output:

```
[
  { "id": "03230", "count_mg": 13 },
  { "id": "03238", "count_mg": 14 },
  { "id": "03229", "count_mg": 13 },
  { "id": "03226", "count_mg": 15 },
  { "id": "03227", "count_mg": 19 }
]
```

Example 2

Here's a subquery with multiple aggregate function expressions:

```
SELECT TOP 5 f.id, (
    SELECT Count(1) AS count, Sum(n.nutritionValue) AS sum
    FROM n IN f.nutrients
    WHERE n.units = 'mg'
) AS unit_mg
FROM food f
```

Query output:

```
[
  { "id": "03230","unit_mg": { "count": 13,"sum": 147.072 } },
  { "id": "03238","unit_mg": { "count": 14,"sum": 107.385 } },
  { "id": "03229","unit_mg": { "count": 13,"sum": 141.579 } },
  { "id": "03226","unit_mg": { "count": 15,"sum": 183.91399999999996 } },
  { "id": "03227","unit_mg": { "count": 19,"sum": 94.788999999999987 } }
]
```

Example 3

Here's a query with an aggregate subquery in both the projection and the filter:

```
SELECT TOP 5
    f.id,
 (SELECT VALUE Count(1) FROM n IN f.nutrients WHERE n.units = 'mg') AS count_mg
FROM food f
WHERE (SELECT VALUE Count(1) FROM n IN f.nutrients WHERE n.units = 'mg') > 20
```

Query output:

```
[
  { "id": "03235", "count_mg": 27 },
  { "id": "03246", "count_mg": 21 },
  { "id": "03267", "count_mg": 21 },
  { "id": "03269", "count_mg": 21 },
  { "id": "03274", "count_mg": 21 }
]
```

A more optimal way to write this query is to join on the subquery and reference the subquery alias in both the SELECT and WHERE clauses. This query is more efficient because you need to execute the subquery only within the join statement, and not in both the projection and filter.

```
SELECT TOP 5 f.id, count_mg
FROM food f
JOIN (SELECT VALUE Count(1) FROM n IN f.nutrients WHERE n.units = 'mg') AS count_mg
WHERE count_mg > 20
```

EXISTS expression

Azure Cosmos DB supports EXISTS expressions. This is an aggregate scalar subquery built into the Azure Cosmos DB SQL API. EXISTS is a Boolean expression that takes a subquery expression and returns true if the subquery returns any rows. Otherwise, it returns false.

Because the Azure Cosmos DB SQL API doesn't differentiate between Boolean expressions and any other scalar expressions, you can use EXISTS in both SELECT and WHERE clauses. This is unlike T-SQL, where a Boolean expression (for example, EXISTS, BETWEEN, and IN) is restricted to the filter.

If the EXISTS subquery returns a single value that's undefined, EXISTS will evaluate to false. For instance, consider the following query that evaluates to false:

```
SELECT EXISTS (SELECT VALUE undefined)
```

If the VALUE keyword in the preceding subquery is omitted, the query will evaluate to true:

```
SELECT EXISTS (SELECT undefined)
```

The subquery will enclose the list of values in the selected list in an object. If the selected list has no values, the subquery will return the single value '{}'. This value is defined, so EXISTS evaluates to true.

Example: Rewriting ARRAY_CONTAINS and JOIN as EXISTS

A common use case of ARRAY_CONTAINS is to filter a document by the existence of an item in an array. In this case, we're checking to see if the tags array contains an item named "orange."

```
SELECT TOP 5 f.id, f.tags
FROM food f
WHERE ARRAY_CONTAINS(f.tags, {name: 'orange'})
```

You can rewrite the same query to use EXISTS:

```
SELECT TOP 5 f.id, f.tags
FROM food f
WHERE EXISTS(SELECT VALUE t FROM t IN f.tags WHERE t.name = 'orange')
```

Additionally, ARRAY_CONTAINS can only check if a value is equal to any element within an array. If you need more complex filters on array properties, use JOIN.

Consider the following query that filters based on the units and `nutritionValue` properties in the array:

```
SELECT VALUE c.description
FROM c
JOIN n IN c.nutrients
WHERE n.units= "mg" AND n.nutritionValue > 0
```

For each of the documents in the collection, a cross-product is performed with its array elements. This JOIN operation makes it possible to filter on properties within the array. However, this query's RU consumption will be significant. For instance, if 1,000 documents had 100 items in each array, it will expand to 1,000 x 100 (that is, 100,000) tuples.

Using EXISTS can help to avoid this expensive cross-product:

```
SELECT VALUE c.description
FROM c
WHERE EXISTS(
    SELECT VALUE n
    FROM n IN c.nutrients
    WHERE n.units = "mg" AND n.nutritionValue > 0
)
```

In this case, you filter on array elements within the EXISTS subquery. If an array element matches the filter, then you project it and EXISTS evaluates to true.

You can also alias EXISTS and reference it in the projection:

```
SELECT TOP 1 c.description, EXISTS(
    SELECT VALUE n
    FROM n IN c.nutrients
    WHERE n.units = "mg" AND n.nutritionValue > 0) as a
FROM c
```

Query output:

```
[
    {
        "description": "Babyfood, dessert, fruit pudding, orange, strained",
        "a": true
    }
]
```

ARRAY expression

You can use the ARRAY expression to project the results of a query as an array. You can use this expression only within the SELECT clause of the query.

```
SELECT TOP 1   f.id, ARRAY(SELECT VALUE t.name FROM t in f.tags) AS tagNames
FROM  food f
```

Query output:

```
[
    {
        "id": "03238",
        "tagNames": [
            "babyfood",
            "dessert",
            "tropical fruit",
            "junior"
        ]
    }
]
```

As with other subqueries, filters with the ARRAY expression are possible.

```
SELECT TOP 1 c.id, ARRAY(SELECT VALUE t FROM t in c.tags WHERE t.name != 'infant formula') AS tagNames
FROM c
```

Query output:

```
[
    {
        "id": "03226",
        "tagNames": [
            {
                "name": "babyfood"
            },
            {
                "name": "dessert"
            },
            {
                "name": "fruit pudding"
            },
            {
                "name": "orange"
            },
            {
                "name": "strained"
            }
        ]
    }
]
```

Array expressions can also come after the FROM clause in subqueries.

```
SELECT TOP 1 c.id, ARRAY(SELECT VALUE t.name FROM t in c.tags) as tagNames
FROM c
JOIN n IN (SELECT VALUE ARRAY(SELECT t FROM t in c.tags WHERE t.name != 'infant formula'))
```

Query output:

```
[
    {
        "id": "03238",
        "tagNames": [
            "babyfood",
            "dessert",
            "tropical fruit",
            "junior"
        ]
    }
]
```

Next steps

- Azure Cosmos DB .NET samples
- Model document data

Joins in Azure Cosmos DB

12/4/2020 • 5 minutes to read • Edit Online

APPLIES TO: SQL API

In a relational database, joins across tables are the logical corollary to designing normalized schemas. In contrast, the SQL API uses the denormalized data model of schema-free items, which is the logical equivalent of a *self-join*.

Inner joins result in a complete cross product of the sets participating in the join. The result of an N-way join is a set of N-element tuples, where each value in the tuple is associated with the aliased set participating in the join and can be accessed by referencing that alias in other clauses.

Syntax

The language supports the syntax `<from_source1> JOIN <from_source2> JOIN ... JOIN <from_sourceN>`. This query returns a set of tuples with `N` values. Each tuple has values produced by iterating all container aliases over their respective sets.

Let's look at the following FROM clause: `<from_source1> JOIN <from_source2> JOIN ... JOIN <from_sourceN>`

Let each source define `input_alias1, input_alias2, …, input_aliasN`. This FROM clause returns a set of N-tuples (tuple with N values). Each tuple has values produced by iterating all container aliases over their respective sets.

Example 1 - 2 sources

- Let `<from_source1>` be container-scoped and represent set {A, B, C}.
- Let `<from_source2>` be document-scoped referencing input_alias1 and represent sets:

  {1, 2} for `input_alias1 = A,`

  {3} for `input_alias1 = B,`

  {4, 5} for `input_alias1 = C,`
- The FROM clause `<from_source1> JOIN <from_source2>` will result in the following tuples:

  (`input_alias1, input_alias2`):

  `(A, 1), (A, 2), (B, 3), (C, 4), (C, 5)`

Example 2 - 3 sources

- Let `<from_source1>` be container-scoped and represent set {A, B, C}.
- Let `<from_source2>` be document-scoped referencing `input_alias1` and represent sets:

  {1, 2} for `input_alias1 = A,`

  {3} for `input_alias1 = B,`

  {4, 5} for `input_alias1 = C,`
- Let `<from_source3>` be document-scoped referencing `input_alias2` and represent sets:

  {100, 200} for `input_alias2 = 1,`

{300} for `input_alias2 = 3,`

- The FROM clause `<from_source1> JOIN <from_source2> JOIN <from_source3>` will result in the following tuples:

  (input_alias1, input_alias2, input_alias3):

  (A, 1, 100), (A, 1, 200), (B, 3, 300)

> **NOTE**
> Lack of tuples for other values of `input_alias1`, `input_alias2`, for which the `<from_source3>` did not return any values.

Example 3 - 3 sources

- Let <from_source1> be container-scoped and represent set {A, B, C}.
- Let `<from_source1>` be container-scoped and represent set {A, B, C}.
- Let <from_source2> be document-scoped referencing input_alias1 and represent sets:

  {1, 2} for `input_alias1 = A,`

  {3} for `input_alias1 = B,`

  {4, 5} for `input_alias1 = C,`

- Let `<from_source3>` be scoped to `input_alias1` and represent sets:

  {100, 200} for `input_alias2 = A,`

  {300} for `input_alias2 = C,`

- The FROM clause `<from_source1> JOIN <from_source2> JOIN <from_source3>` will result in the following tuples:

  (`input_alias1, input_alias2, input_alias3`):

  (A, 1, 100), (A, 1, 200), (A, 2, 100), (A, 2, 200), (C, 4, 300) , (C, 5, 300)

> **NOTE**
> This resulted in cross product between `<from_source2>` and `<from_source3>` because both are scoped to the same `<from_source1>`. This resulted in 4 (2x2) tuples having value A, 0 tuples having value B (1x0) and 2 (2x1) tuples having value C.

Examples

The following examples show how the JOIN clause works. Before you run these examples, upload the sample family data. In the following example, the result is empty, since the cross product of each item from source and an empty set is empty:

```
SELECT f.id
FROM Families f
JOIN f.NonExistent
```

The result is:

```
[{
}]
```

In the following example, the join is a cross product between two JSON objects, the item root `id` and the `children` subroot. The fact that `children` is an array isn't effective in the join, because it deals with a single root that is the `children` array. The result contains only two results, because the cross product of each item with the array yields exactly only one item.

```
SELECT f.id
FROM Families f
JOIN f.children
```

The results are:

```
[
  {
    "id": "AndersenFamily"
  },
  {
    "id": "WakefieldFamily"
  }
]
```

The following example shows a more conventional join:

```
SELECT f.id
FROM Families f
JOIN c IN f.children
```

The results are:

```
[
  {
    "id": "AndersenFamily"
  },
  {
    "id": "WakefieldFamily"
  },
  {
    "id": "WakefieldFamily"
  }
]
```

The FROM source of the JOIN clause is an iterator. So, the flow in the preceding example is:

1. Expand each child element `c` in the array.
2. Apply a cross product with the root of the item `f` with each child element `c` that the first step flattened.
3. Finally, project the root object `f` `id` property alone.

The first item, `AndersenFamily`, contains only one `children` element, so the result set contains only a single object. The second item, `WakefieldFamily`, contains two `children`, so the cross product produces two objects, one for each `children` element. The root fields in both these items are the same, just as you would expect in a cross product.

The real utility of the JOIN clause is to form tuples from the cross product in a shape that's otherwise difficult to

project. The example below filters on the combination of a tuple that lets the user choose a condition satisfied by the tuples overall.

```
SELECT
    f.id AS familyName,
    c.givenName AS childGivenName,
    c.firstName AS childFirstName,
    p.givenName AS petName
FROM Families f
JOIN c IN f.children
JOIN p IN c.pets
```

The results are:

```
[
  {
    "familyName": "AndersenFamily",
    "childFirstName": "Henriette Thaulow",
    "petName": "Fluffy"
  },
  {
    "familyName": "WakefieldFamily",
    "childGivenName": "Jesse",
    "petName": "Goofy"
  },
  {
   "familyName": "WakefieldFamily",
   "childGivenName": "Jesse",
   "petName": "Shadow"
  }
]
```

The following extension of the preceding example performs a double join. You could view the cross product as the following pseudo-code:

```
for-each(Family f in Families)
{
    for-each(Child c in f.children)
    {
        for-each(Pet p in c.pets)
        {
            return (Tuple(f.id AS familyName,
              c.givenName AS childGivenName,
              c.firstName AS childFirstName,
              p.givenName AS petName));
        }
    }
}
```

`AndersenFamily` has one child who has one pet, so the cross product yields one row (1*1*1) from this family. `WakefieldFamily` has two children, only one of whom has pets, but that child has two pets. The cross product for this family yields 1*1*2 = 2 rows.

In the next example, there is an additional filter on `pet`, which excludes all the tuples where the pet name is not `Shadow`. You can build tuples from arrays, filter on any of the elements of the tuple, and project any combination of the elements.

```
SELECT
    f.id AS familyName,
    c.givenName AS childGivenName,
    c.firstName AS childFirstName,
    p.givenName AS petName
FROM Families f
JOIN c IN f.children
JOIN p IN c.pets
WHERE p.givenName = "Shadow"
```

The results are:

```
[
  {
   "familyName": "WakefieldFamily",
   "childGivenName": "Jesse",
   "petName": "Shadow"
  }
]
```

Next steps

- Getting started
- Azure Cosmos DB .NET samples
- Subqueries

Working with arrays and objects in Azure Cosmos DB

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: SQL API

A key feature of the Azure Cosmos DB SQL API is array and object creation.

Arrays

You can construct arrays, as shown in the following example:

```
SELECT [f.address.city, f.address.state] AS CityState
FROM Families f
```

The results are:

```
[
  {
    "CityState": [
      "Seattle",
      "WA"
    ]
  },
  {
    "CityState": [
      "NY",
      "NY"
    ]
  }
]
```

You can also use the ARRAY expression to construct an array from subquery's results. This query gets all the distinct given names of children in an array.

```
SELECT f.id, ARRAY(SELECT DISTINCT VALUE c.givenName FROM c IN f.children) as ChildNames
FROM f
```

Iteration

The SQL API provides support for iterating over JSON arrays, with a new construct added via the IN keyword in the FROM source. In the following example:

```
SELECT *
FROM Families.children
```

The results are:

```
[
  [
    {
      "firstName": "Henriette Thaulow",
      "gender": "female",
      "grade": 5,
      "pets": [{ "givenName": "Fluffy"}]
    }
  ],
  [
    {
        "familyName": "Merriam",
        "givenName": "Jesse",
        "gender": "female",
        "grade": 1
    },
    {
        "familyName": "Miller",
        "givenName": "Lisa",
        "gender": "female",
        "grade": 8
    }
  ]
]
```

The next query performs iteration over `children` in the `Families` container. The output array is different from the preceding query. This example splits `children`, and flattens the results into a single array:

```
SELECT *
FROM c IN Families.children
```

The results are:

```
[
  {
      "firstName": "Henriette Thaulow",
      "gender": "female",
      "grade": 5,
      "pets": [{ "givenName": "Fluffy" }]
  },
  {
      "familyName": "Merriam",
      "givenName": "Jesse",
      "gender": "female",
      "grade": 1
  },
  {
      "familyName": "Miller",
      "givenName": "Lisa",
      "gender": "female",
      "grade": 8
  }
]
```

You can filter further on each individual entry of the array, as shown in the following example:

```
SELECT c.givenName
FROM c IN Families.children
WHERE c.grade = 8
```

The results are:

```
[{
  "givenName": "Lisa"
}]
```

You can also aggregate over the result of an array iteration. For example, the following query counts the number of children among all families:

```
SELECT COUNT(child)
FROM child IN Families.children
```

The results are:

```
[
  {
    "$1": 3
  }
]
```

Next steps

- Getting started
- Azure Cosmos DB .NET samples
- Joins

Keywords in Azure Cosmos DB

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: SQL API

This article details keywords which may be used in Azure Cosmos DB SQL queries.

BETWEEN

You can use the `BETWEEN` keyword to express queries against ranges of string or numerical values. For example, the following query returns all items in which the first child's grade is 1-5, inclusive.

```
SELECT *
FROM Families.children[0] c
WHERE c.grade BETWEEN 1 AND 5
```

You can also use the `BETWEEN` keyword in the `SELECT` clause, as in the following example.

```
SELECT (c.grade BETWEEN 0 AND 10)
FROM Families.children[0] c
```

In SQL API, unlike ANSI SQL, you can express range queries against properties of mixed types. For example, `grade` might be a number like `5` in some items and a string like `grade4` in others. In these cases, as in JavaScript, the comparison between the two different types results in `Undefined`, so the item is skipped.

DISTINCT

The `DISTINCT` keyword eliminates duplicates in the query's projection.

In this example, the query projects values for each last name:

```
SELECT DISTINCT VALUE f.lastName
FROM Families f
```

The results are:

```
[
    "Andersen"
]
```

You can also project unique objects. In this case, the lastName field does not exist in one of the two documents, so the query returns an empty object.

```
SELECT DISTINCT f.lastName
FROM Families f
```

The results are:

```
[
    {
        "lastName": "Andersen"
    },
    {}
]
```

`DISTINCT` can also be used in the projection within a subquery:

```
SELECT f.id, ARRAY(SELECT DISTINCT VALUE c.givenName FROM c IN f.children) as ChildNames
FROM f
```

This query projects an array which contains each child's givenName with duplicates removed. This array is aliased as ChildNames and projected in the outer query.

The results are:

```
[
    {
        "id": "AndersenFamily",
        "ChildNames": []
    },
    {
        "id": "WakefieldFamily",
        "ChildNames": [
            "Jesse",
            "Lisa"
        ]
    }
]
```

Queries with an aggregate system function and a subquery with `DISTINCT` are not supported. For example, the following query is not supported:

```
SELECT COUNT(1) FROM (SELECT DISTINCT f.lastName FROM f)
```

IN

Use the IN keyword to check whether a specified value matches any value in a list. For example, the following query returns all family items where the `id` is `WakefieldFamily` or `AndersenFamily`.

```
    SELECT *
    FROM Families
    WHERE Families.id IN ('AndersenFamily', 'WakefieldFamily')
```

The following example returns all items where the state is any of the specified values:

```
    SELECT *
    FROM Families
    WHERE Families.address.state IN ("NY", "WA", "CA", "PA", "OH", "OR", "MI", "WI", "MN", "FL")
```

The SQL API provides support for iterating over JSON arrays, with a new construct added via the in keyword in the FROM source.

If you include your partition key in the `IN` filter, your query will automatically filter to only the relevant partitions.

TOP

The TOP keyword returns the first `N` number of query results in an undefined order. As a best practice, use TOP with the `ORDER BY` clause to limit results to the first `N` number of ordered values. Combining these two clauses is the only way to predictably indicate which rows TOP affects.

You can use TOP with a constant value, as in the following example, or with a variable value using parameterized queries.

```
SELECT TOP 1 *
FROM Families f
```

The results are:

```
[{
    "id": "AndersenFamily",
    "lastName": "Andersen",
    "parents": [
       { "firstName": "Thomas" },
       { "firstName": "Mary Kay"}
    ],
    "children": [
       {
           "firstName": "Henriette Thaulow", "gender": "female", "grade": 5,
           "pets": [{ "givenName": "Fluffy" }]
       }
    ],
    "address": { "state": "WA", "county": "King", "city": "Seattle" },
    "creationDate": 1431620472,
    "isRegistered": true
}]
```

Next steps

- Getting started
- Joins
- Subqueries

Azure Cosmos DB SQL query constants

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: SQL API

A constant, also known as a literal or a scalar value, is a symbol that represents a specific data value. The format of a constant depends on the data type of the value it represents.

Supported scalar data types:

TYPE	VALUES ORDER
Undefined	Single value: **undefined**
Null	Single value: **null**
Boolean	Values: **false**, **true**.
Number	A double-precision floating-point number, IEEE 754 standard.
String	A sequence of zero or more Unicode characters. Strings must be enclosed in single or double quotes.
Array	A sequence of zero or more elements. Each element can be a value of any scalar data type, except **Undefined**.
Object	An unordered set of zero or more name/value pairs. Name is a Unicode string, value can be of any scalar data type, except **Undefined**.

Syntax

```
<constant> ::=
   <undefined_constant>
     | <null_constant>
     | <boolean_constant>
     | <number_constant>
     | <string_constant>
     | <array_constant>
     | <object_constant>

<undefined_constant> ::= undefined

<null_constant> ::= null

<boolean_constant> ::= false | true

<number_constant> ::= decimal_literal | hexadecimal_literal

<string_constant> ::= string_literal

<array_constant> ::=
    '[' [<constant>][,...n] ']'

<object_constant> ::=
   '{' [{property_name | "property_name"} : <constant>][,...n] '}'
```

Arguments

- `<undefined_constant>; Undefined`

  Represents undefined value of type Undefined.

- `<null_constant>; null`

  Represents **null** value of type **Null**.

- `<boolean_constant>`

  Represents constant of type Boolean.

- `false`

  Represents **false** value of type Boolean.

- `true`

  Represents **true** value of type Boolean.

- `<number_constant>`

  Represents a constant.

- `decimal_literal`

  Decimal literals are numbers represented using either decimal notation, or scientific notation.

- `hexadecimal_literal`

  Hexadecimal literals are numbers represented using prefix '0x' followed by one or more hexadecimal digits.

- `<string_constant>`

  Represents a constant of type String.

- `string _literal`

  String literals are Unicode strings represented by a sequence of zero or more Unicode characters or escape sequences. String literals are enclosed in single quotes (apostrophe: ') or double quotes (quotation mark: ").

  Following escape sequences are allowed:

ESCAPE SEQUENCE	DESCRIPTION	UNICODE CHARACTER
\'	apostrophe (')	U+0027
\"	quotation mark (")	U+0022
\\	reverse solidus (\)	U+005C
\/	solidus (/)	U+002F
\b	backspace	U+0008
\f	form feed	U+000C
\n	line feed	U+000A
\r	carriage return	U+000D
\t	tab	U+0009
\uXXXX	A Unicode character defined by 4 hexadecimal digits.	U+XXXX

Next steps

- Azure Cosmos DB .NET samples
- Model document data

Operators in Azure Cosmos DB

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: SQL API

This article details the various operators supported by Azure Cosmos DB.

Equality and Comparison Operators

The following table shows the result of equality comparisons in the SQL API between any two JSON types.

OP	UNDEFINED	NULL	BOOLEAN	NUMBER	STRING	OBJECT	ARRAY
Undefined	Undefined	Undefined	Undefined	Undefined	Undefined	Undefined	Undefined
Null	Undefined	**Ok**	Undefined	Undefined	Undefined	Undefined	Undefined
Boolean	Undefined	Undefined	**Ok**	Undefined	Undefined	Undefined	Undefined
Number	Undefined	Undefined	Undefined	**Ok**	Undefined	Undefined	Undefined
String	Undefined	Undefined	Undefined	Undefined	**Ok**	Undefined	Undefined
Object	Undefined	Undefined	Undefined	Undefined	Undefined	**Ok**	Undefined
Array	Undefined	Undefined	Undefined	Undefined	Undefined	Undefined	**Ok**

For comparison operators such as `>`, `>=`, `!=`, `<`, and `<=`, comparison across types or between two objects or arrays produces `Undefined`.

If the result of the scalar expression is `Undefined`, the item isn't included in the result, because `Undefined` doesn't equal `true`.

For example, the following query's comparison between a number and string value produces `Undefined`. Therefore, the filter does not include any results.

```
SELECT *
FROM c
WHERE 7 = 'a'
```

Logical (AND, OR and NOT) operators

Logical operators operate on Boolean values. The following tables show the logical truth tables for these operators:

OR operator

Returns `true` when either of the conditions is `true`.

	TRUE	FALSE	UNDEFINED
True	True	True	True
False	True	False	Undefined
Undefined	True	Undefined	Undefined

AND operator

Returns `true` when both expressions are `true`.

	TRUE	FALSE	UNDEFINED
True	True	False	Undefined
False	False	False	False
Undefined	Undefined	False	Undefined

NOT operator

Reverses the value of any Boolean expression.

	NOT
True	False
False	True
Undefined	Undefined

Operator Precedence

The logical operators `OR`, `AND`, and `NOT` have the precedence level shown below:

OPERATOR	PRIORITY
NOT	1
AND	2
OR	3

* operator

The special operator * projects the entire item as is. When used, it must be the only projected field. A query like `SELECT * FROM Families f` is valid, but `SELECT VALUE * FROM Families f` and `SELECT *, f.id FROM Families f` are not valid.

? and ?? operators

You can use the Ternary (?) and Coalesce (??) operators to build conditional expressions, as in programming languages like C# and JavaScript.

You can use the ? operator to construct new JSON properties on the fly. For example, the following query classifies grade levels into `elementary` or `other`:

```
SELECT (c.grade < 5)? "elementary": "other" AS gradeLevel
FROM Families.children[0] c
```

You can also nest calls to the ? operator, as in the following query:

```
SELECT (c.grade < 5)? "elementary": ((c.grade < 9)? "junior": "high") AS gradeLevel
FROM Families.children[0] c
```

As with other query operators, the ? operator excludes items if the referenced properties are missing or the types being compared are different.

Use the ?? operator to efficiently check for a property in an item when querying against semi-structured or mixed-type data. For example, the following query returns `lastName` if present, or `surname` if `lastName` isn't present.

```
SELECT f.lastName ?? f.surname AS familyName
FROM Families f
```

Next steps

- Azure Cosmos DB .NET samples
- Keywords
- SELECT clause

Scalar expressions in Azure Cosmos DB SQL queries

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: SQL API

The SELECT clause supports scalar expressions. A scalar expression is a combination of symbols and operators that can be evaluated to obtain a single value. Examples of scalar expressions include: constants, property references, array element references, alias references, or function calls. Scalar expressions can be combined into complex expressions using operators.

Syntax

```
<scalar_expression> ::=
       <constant>
     | input_alias
     | parameter_name
     | <scalar_expression>.property_name
     | <scalar_expression>'['"property_name"|array_index']'
     | unary_operator <scalar_expression>
     | <scalar_expression> binary_operator <scalar_expression>
     | <scalar_expression> ? <scalar_expression> : <scalar_expression>
     | <scalar_function_expression>
     | <create_object_expression>
     | <create_array_expression>
     | (<scalar_expression>)

<scalar_function_expression> ::=
        'udf.' Udf_scalar_function([<scalar_expression>][,…n])
        | builtin_scalar_function([<scalar_expression>][,…n])

<create_object_expression> ::=
   '{' [{property_name | "property_name"} : <scalar_expression>][,…n] '}'

<create_array_expression> ::=
   '[' [<scalar_expression>][,…n] ']'
```

Arguments

- `<constant>`

  Represents a constant value. See Constants section for details.

- `input_alias`

  Represents a value defined by the `input_alias` introduced in the `FROM` clause.
  This value is guaranteed to not be **undefined** –**undefined** values in the input are skipped.

- `<scalar_expression>.property_name`

  Represents a value of the property of an object. If the property does not exist or property is referenced on a value, which is not an object, then the expression evaluates to **undefined** value.

- `<scalar_expression>'['"property_name"|array_index']'`

  Represents a value of the property with name `property_name` or array element with index `array_index` of

an array. If the property/array index does not exist or the property/array index is referenced on a value that is not an object/array, then the expression evaluates to undefined value.

- `unary_operator <scalar_expression>`

  Represents an operator that is applied to a single value. See Operators section for details.

- `<scalar_expression> binary_operator <scalar_expression>`

  Represents an operator that is applied to two values. See Operators section for details.

- `<scalar_function_expression>`

  Represents a value defined by a result of a function call.

- `udf_scalar_function`

  Name of the user-defined scalar function.

- `builtin_scalar_function`

  Name of the built-in scalar function.

- `<create_object_expression>`

  Represents a value obtained by creating a new object with specified properties and their values.

- `<create_array_expression>`

  Represents a value obtained by creating a new array with specified values as elements

- `parameter_name`

  Represents a value of the specified parameter name. Parameter names must have a single @ as the first character.

Remarks

When calling a built-in or user-defined scalar function, all arguments must be defined. If any of the arguments is undefined, the function will not be called and the result will be undefined.

When creating an object, any property that is assigned undefined value will be skipped and not included in the created object.

When creating an array, any element value that is assigned **undefined** value will be skipped and not included in the created object. This will cause the next defined element to take its place in such a way that the created array will not have skipped indexes.

Examples

```
SELECT ((2 + 11 % 7)-2)/3
```

The results are:

```
[{
  "$1": 1.33333
}]
```

In the following query, the result of the scalar expression is a Boolean:

```
SELECT f.address.city = f.address.state AS AreFromSameCityState
FROM Families f
```

The results are:

```
[
  {
    "AreFromSameCityState": false
  },
  {
    "AreFromSameCityState": true
  }
]
```

Next steps

- Introduction to Azure Cosmos DB
- Azure Cosmos DB .NET samples
- Subqueries

User-defined functions (UDFs) in Azure Cosmos DB

12/4/2020 • 3 minutes to read • Edit Online

APPLIES TO: SQL API

The SQL API provides support for user-defined functions (UDFs). With scalar UDFs, you can pass in zero or many arguments and return a single argument result. The API checks each argument for being legal JSON values.

UDF use cases

The API extends the SQL syntax to support custom application logic using UDFs. You can register UDFs with the SQL API, and reference them in SQL queries. Unlike stored procedures and triggers, UDFs are read-only.

Using UDFs, you can extend Azure Cosmos DB's query language. UDFs are a great way to express complex business logic in a query's projection.

However, we recommending avoiding UDFs when:

- An equivalent system function already exists in Azure Cosmos DB. System functions will always use fewer RU's than the equivalent UDF.
- The UDF is the only filter in the `WHERE` clause of your query. UDF's do not utilize the index so evaluating the UDF will require loading documents. Combining additional filter predicates that use the index, in combination with a UDF, in the `WHERE` clause will reduce the number of documents processed by the UDF.

If you must use the same UDF multiple times in a query, you should reference the UDF in a subquery, allowing you to use a JOIN expression to evaluate the UDF once but reference it many times.

Examples

The following example registers a UDF under an item container in the Cosmos database. The example creates a UDF whose name is `REGEX_MATCH`. It accepts two JSON string values, `input` and `pattern`, and checks if the first matches the pattern specified in the second using JavaScript's `string.match()` function.

```
UserDefinedFunction regexMatchUdf = new UserDefinedFunction
{
    Id = "REGEX_MATCH",
    Body = @"function (input, pattern) {
                return input.match(pattern) !== null;
            };",
};

UserDefinedFunction createdUdf = client.CreateUserDefinedFunctionAsync(
    UriFactory.CreateDocumentCollectionUri("myDatabase", "families"),
    regexMatchUdf).Result;
```

Now, use this UDF in a query projection. You must qualify UDFs with the case-sensitive prefix `udf.` when calling them from within queries.

```
SELECT udf.REGEX_MATCH(Families.address.city, ".*eattle")
FROM Families
```

The results are:

```
[
  {
    "$1": true
  },
  {
    "$1": false
  }
]
```

You can use the UDF qualified with the `udf.` prefix inside a filter, as in the following example:

```
SELECT Families.id, Families.address.city
FROM Families
WHERE udf.REGEX_MATCH(Families.address.city, ".*eattle")
```

The results are:

```
[{
    "id": "AndersenFamily",
    "city": "Seattle"
}]
```

In essence, UDFs are valid scalar expressions that you can use in both projections and filters.

To expand on the power of UDFs, look at another example with conditional logic:

```
   UserDefinedFunction seaLevelUdf = new UserDefinedFunction()
   {
       Id = "SEALEVEL",
       Body = @"function(city) {
               switch (city) {
                   case 'Seattle':
                       return 520;
                   case 'NY':
                       return 410;
                   case 'Chicago':
                       return 673;
                   default:
                       return -1;
                }"
        };

        UserDefinedFunction createdUdf = await client.CreateUserDefinedFunctionAsync(
            UriFactory.CreateDocumentCollectionUri("myDatabase", "families"),
            seaLevelUdf);
```

The following example exercises the UDF:

```
SELECT f.address.city, udf.SEALEVEL(f.address.city) AS seaLevel
FROM Families f
```

The results are:

```
[
 {
   "city": "Seattle",
   "seaLevel": 520
 },
 {
   "city": "NY",
   "seaLevel": 410
 }
]
```

If the properties referred to by the UDF parameters aren't available in the JSON value, the parameter is considered as undefined and the UDF invocation is skipped. Similarly, if the result of the UDF is undefined, it's not included in the result.

As the preceding examples show, UDFs integrate the power of JavaScript language with the SQL API. UDFs provide a rich programmable interface to do complex procedural, conditional logic with the help of built-in JavaScript runtime capabilities. The SQL API provides the arguments to the UDFs for each source item at the current WHERE or SELECT clause stage of processing. The result is seamlessly incorporated in the overall execution pipeline. In summary, UDFs are great tools to do complex business logic as part of queries.

Next steps

- Introduction to Azure Cosmos DB
- System functions
- Aggregates

Aggregate functions in Azure Cosmos DB

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: SQL API

Aggregate functions perform a calculation on a set of values in the `SELECT` clause and return a single value. For example, the following query returns the count of items within a container:

```
SELECT COUNT(1)
FROM c
```

Types of aggregate functions

The SQL API supports the following aggregate functions. `SUM` and `AVG` operate on numeric values, and `COUNT`, `MIN`, and `MAX` work on numbers, strings, Booleans, and nulls.

FUNCTION	DESCRIPTION
AVG	Returns the average of the values in the expression.
COUNT	Returns the number of items in the expression.
MAX	Returns the maximum value in the expression.
MIN	Returns the minimum value in the expression.
SUM	Returns the sum of all the values in the expression.

You can also return only the scalar value of the aggregate by using the VALUE keyword. For example, the following query returns the count of values as a single number:

```
SELECT VALUE COUNT(1)
FROM Families f
```

The results are:

```
[ 2 ]
```

You can also combine aggregations with filters. For example, the following query returns the count of items with the address state of `WA`.

```
SELECT VALUE COUNT(1)
FROM Families f
WHERE f.address.state = "WA"
```

The results are:

```
[ 1 ]
```

Remarks

These aggregate system functions will benefit from a range index. If you expect to do an `AVG`, `COUNT`, `MAX`, `MIN`, or `SUM` on a property, you should include the relevant path in the indexing policy.

Next steps

- Introduction to Azure Cosmos DB
- System functions
- User defined functions

AVG (Azure Cosmos DB)

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: SQL API

This aggregate function returns the average of the values in the expression.

Syntax

```
AVG(<numeric_expr>)
```

Arguments

numeric_expr
Is a numeric expression.

Return types

Returns a numeric expression.

Examples

The following example returns the average value of `propertyA`:

```
SELECT AVG(c.propertyA)
FROM c
```

Remarks

This system function will benefit from a range index. If any arguments in `AVG` are string, boolean, or null, the entire aggregate system function will return `undefined`. If any argument has an `undefined` value, it will not impact the `AVG` calculation.

Next steps

- Mathematical functions in Azure Cosmos DB
- System functions in Azure Cosmos DB
- Aggregate functions in Azure Cosmos DB

COUNT (Azure Cosmos DB)

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: SQL API

This system function returns the count of the values in the expression.

Syntax

```
COUNT(<scalar_expr>)
```

Arguments

scalar_expr
Is any scalar expression

Return types

Returns a numeric expression.

Examples

The following example returns the total count of items in a container:

```
SELECT COUNT(1)
FROM c
```

COUNT can take any scalar expression as input. The below query will produce an equivalent results:

```
SELECT COUNT(2)
FROM c
```

Remarks

This system function will benefit from a range index for any properties in the query's filter.

Next steps

- Mathematical functions in Azure Cosmos DB
- System functions in Azure Cosmos DB
- Aggregate functions in Azure Cosmos DB

MAX (Azure Cosmos DB)

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: SQL API

This aggregate function returns the maximum of the values in the expression.

Syntax

```
MAX(<scalar_expr>)
```

Arguments

scalar_expr
Is a scalar expression.

Return types

Returns a scalar expression.

Examples

The following example returns the maximum value of `propertyA`:

```
SELECT MAX(c.propertyA)
FROM c
```

Remarks

This system function will benefit from a range index. The arguments in `MAX` can be number, string, boolean, or null. Any undefined values will be ignored.

When comparing different types data, the following priority order is used (in descending order):

- string
- number
- boolean
- null

Next steps

- Mathematical functions in Azure Cosmos DB
- System functions in Azure Cosmos DB
- Aggregate functions in Azure Cosmos DB

MIN (Azure Cosmos DB)

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: SQL API

This aggregate function returns the minimum of the values in the expression.

Syntax

```
MIN(<scalar_expr>)
```

Arguments

scalar_expr
Is a scalar expression.

Return types

Returns a scalar expression.

Examples

The following example returns the minimum value of `propertyA`:

```
SELECT MIN(c.propertyA)
FROM c
```

Remarks

This system function will benefit from a range index. The arguments in `MIN` can be number, string, boolean, or null. Any undefined values will be ignored.

When comparing different types data, the following priority order is used (in ascending order):

- null
- boolean
- number
- string

Next steps

- Mathematical functions in Azure Cosmos DB
- System functions in Azure Cosmos DB
- Aggregate functions in Azure Cosmos DB

SUM (Azure Cosmos DB)

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: SQL API

This aggregate function returns the sum of the values in the expression.

Syntax

```
SUM(<numeric_expr>)
```

Arguments

numeric_expr
Is a numeric expression.

Return types

Returns a numeric expression.

Examples

The following example returns the sum of `propertyA`:

```
SELECT SUM(c.propertyA)
FROM c
```

Remarks

This system function will benefit from a range index. If any arguments in `SUM` are string, boolean, or null, the entire aggregate system function will return `undefined`. If any argument has an `undefined` value, it will be not impact the `SUM` calculation.

Next steps

- Mathematical functions in Azure Cosmos DB
- System functions in Azure Cosmos DB
- Aggregate functions in Azure Cosmos DB

System functions (Azure Cosmos DB)

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: SQL API

Cosmos DB provides many built-in SQL functions. The categories of built-in functions are listed below.

FUNCTION GROUP	DESCRIPTION	OPERATIONS
Array functions	The array functions perform an operation on an array input value and return numeric, Boolean, or array value.	ARRAY_CONCAT, ARRAY_CONTAINS, ARRAY_LENGTH, ARRAY_SLICE
Date and Time functions	The date and time functions allow you to get the current UTC date and time in two forms; a numeric timestamp whose value is the Unix epoch in milliseconds or as a string which conforms to the ISO 8601 format.	GetCurrentDateTime, GetCurrentTimestamp
Mathematical functions	The mathematical functions each perform a calculation, usually based on input values that are provided as arguments, and return a numeric value.	ABS, ACOS, ASIN, ATAN, ATN2, CEILING, COS, COT, DEGREES, EXP, FLOOR, LOG, LOG10, PI, POWER, RADIANS, RAND, ROUND, SIGN, SIN, SQRT, SQUARE, TAN, TRUNC
Spatial functions	The spatial functions perform an operation on a spatial object input value and return a numeric or Boolean value.	ST_DISTANCE, ST_INTERSECTS, ST_ISVALID, ST_ISVALIDDETAILED, ST_WITHIN
String functions	The string functions perform an operation on a string input value and return a string, numeric or Boolean value.	CONCAT, CONTAINS, ENDSWITH, INDEX_OF, LEFT, LENGTH, LOWER, LTRIM, REGEXMATCHREPLACE, REPLICATE, REVERSE, RIGHT, RTRIM, STARTSWITH, StringToArray, StringToBoolean, StringToNull, StringToNumber, StringToObject, SUBSTRING, ToString, TRIM, UPPER
Type checking functions	The type checking functions allow you to check the type of an expression within SQL queries.	IS_ARRAY, IS_BOOL, IS_DEFINED, IS_NULL, IS_NUMBER, IS_OBJECT, IS_PRIMITIVE, IS_STRING

Built-in versus User Defined Functions (UDFs)

If you're currently using a user-defined function (UDF) for which a built-in function is now available, the corresponding built-in function will be quicker to run and more efficient.

Built-in versus ANSI SQL functions

The main difference between Cosmos DB functions and ANSI SQL functions is that Cosmos DB functions are designed to work well with schemaless and mixed-schema data. For example, if a property is missing or has a non-numeric value like `unknown`, the item is skipped instead of returning an error.

Next steps

- Introduction to Azure Cosmos DB
- Array functions
- Date and time functions
- Mathematical functions
- Spatial functions
- String functions
- Type checking functions
- User Defined Functions
- Aggregates

Array functions (Azure Cosmos DB)

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: SQL API

The array functions let you perform operations on arrays in Azure Cosmos DB.

Functions

The following scalar functions perform an operation on an array input value and return numeric, boolean or array value:

- ARRAY_CONCAT
- ARRAY_CONTAINS
- ARRAY_LENGTH
- ARRAY_SLICE

Next steps

- System functions Azure Cosmos DB
- Introduction to Azure Cosmos DB
- User Defined Functions
- Aggregates

ARRAY_CONCAT (Azure Cosmos DB)

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: SQL API

Returns an array that is the result of concatenating two or more array values.

Syntax

```
ARRAY_CONCAT (<arr_expr1>, <arr_expr2> [, <arr_exprN>])
```

Arguments

arr_expr
Is an array expression to concatenate to the other values. The `ARRAY_CONCAT` function requires at least two *arr_expr* arguments.

Return types

Returns an array expression.

Examples

The following example how to concatenate two arrays.

```
SELECT ARRAY_CONCAT(["apples", "strawberries"], ["bananas"]) AS arrayConcat
```

Here is the result set.

```
[{"arrayConcat": ["apples", "strawberries", "bananas"]}]
```

Remarks

This system function will not utilize the index.

Next steps

- Array functions Azure Cosmos DB
- System functions Azure Cosmos DB
- Introduction to Azure Cosmos DB

ARRAY_CONTAINS (Azure Cosmos DB)

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: SQL API

Returns a Boolean indicating whether the array contains the specified value. You can check for a partial or full match of an object by using a boolean expression within the command.

Syntax

```
ARRAY_CONTAINS (<arr_expr>, <expr> [, bool_expr])
```

Arguments

arr_expr
Is the array expression to be searched.

expr
Is the expression to be found.

bool_expr
Is a boolean expression. If it evaluates to 'true' and if the specified search value is an object, the command checks for a partial match (the search object is a subset of one of the objects). If it evaluates to 'false', the command checks for a full match of all objects within the array. The default value if not specified is false.

Return types

Returns a Boolean value.

Examples

The following example how to check for membership in an array using `ARRAY_CONTAINS`.

```
SELECT
           ARRAY_CONTAINS(["apples", "strawberries", "bananas"], "apples") AS b1,
           ARRAY_CONTAINS(["apples", "strawberries", "bananas"], "mangoes") AS b2
```

Here is the result set.

```
[{"b1": true, "b2": false}]
```

The following example how to check for a partial match of a JSON in an array using ARRAY_CONTAINS.

```
SELECT
    ARRAY_CONTAINS([{"name": "apples", "fresh": true}, {"name": "strawberries", "fresh": true}], {"name":
"apples"}, true) AS b1,
    ARRAY_CONTAINS([{"name": "apples", "fresh": true}, {"name": "strawberries", "fresh": true}], {"name":
"apples"}) AS b2,
    ARRAY_CONTAINS([{"name": "apples", "fresh": true}, {"name": "strawberries", "fresh": true}], {"name":
"mangoes"}, true) AS b3
```

Here is the result set.

```
[{
  "b1": true,
  "b2": false,
  "b3": false
}]
```

Remarks

This system function will benefit from a range index.

Next steps

- Array functions Azure Cosmos DB
- System functions Azure Cosmos DB
- Introduction to Azure Cosmos DB

ARRAY_LENGTH (Azure Cosmos DB)

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: SQL API

Returns the number of elements of the specified array expression.

Syntax

```
ARRAY_LENGTH(<arr_expr>)
```

Arguments

arr_expr
Is an array expression.

Return types

Returns a numeric expression.

Examples

The following example how to get the length of an array using `ARRAY_LENGTH`.

```
SELECT ARRAY_LENGTH(["apples", "strawberries", "bananas"]) AS len
```

Here is the result set.

```
[{"len": 3}]
```

Remarks

This system function will not utilize the index.

Next steps

- Array functions Azure Cosmos DB
- System functions Azure Cosmos DB
- Introduction to Azure Cosmos DB

ARRAY_SLICE (Azure Cosmos DB)

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: SQL API

Returns part of an array expression.

Syntax

```
ARRAY_SLICE (<arr_expr>, <num_expr> [, <num_expr>])
```

Arguments

arr_expr
Is any array expression.

num_expr
Zero-based numeric index at which to begin the array. Negative values may be used to specify the starting index relative to the last element of the array i.e. -1 references the last element in the array.

num_expr Optional numeric expression that sets the maximum number of elements in the resulting array.

Return types

Returns an array expression.

Examples

The following example shows how to get different slices of an array using `ARRAY_SLICE`.

```
SELECT
           ARRAY_SLICE(["apples", "strawberries", "bananas"], 1) AS s1,
           ARRAY_SLICE(["apples", "strawberries", "bananas"], 1, 1) AS s2,
           ARRAY_SLICE(["apples", "strawberries", "bananas"], -2, 1) AS s3,
           ARRAY_SLICE(["apples", "strawberries", "bananas"], -2, 2) AS s4,
           ARRAY_SLICE(["apples", "strawberries", "bananas"], 1, 0) AS s5,
           ARRAY_SLICE(["apples", "strawberries", "bananas"], 1, 1000) AS s6,
           ARRAY_SLICE(["apples", "strawberries", "bananas"], 1, -100) AS s7
```

Here is the result set.

```
[{
           "s1": ["strawberries", "bananas"],
           "s2": ["strawberries"],
           "s3": ["strawberries"],
           "s4": ["strawberries", "bananas"],
           "s5": [],
           "s6": ["strawberries", "bananas"],
           "s7": []
}]
```

Remarks

This system function will not utilize the index.

Next steps

- Array functions Azure Cosmos DB
- System functions Azure Cosmos DB
- Introduction to Azure Cosmos DB

Date and time functions (Azure Cosmos DB)

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: SQL API

The date and time functions let you perform DateTime and timestamp operations in Azure Cosmos DB.

Functions to obtain the date and time

The following scalar functions allow you to get the current UTC date and time in three forms: a string which conforms to the ISO 8601 format, a numeric timestamp whose value is the number of milliseconds which have elapsed since the Unix epoch, or numeric ticks whose value is the number of 100 nanosecond ticks which have elapsed since the Unix epoch:

- GetCurrentDateTime
- GetCurrentTimestamp
- GetCurrentTicks

Functions to work with DateTime values

The following functions allow you to easily manipulate DateTime, timestamp, and tick values:

- DateTimeAdd
- DateTimeDiff
- DateTimeFromParts
- DateTimePart
- DateTimeToTicks
- DateTimeToTimestamp
- TicksToDateTime
- TimestampToDateTime

Next steps

- System functions Azure Cosmos DB
- Introduction to Azure Cosmos DB
- User Defined Functions
- Aggregates

DateTimeAdd (Azure Cosmos DB)

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: SQL API

Returns DateTime string value resulting from adding a specified number value (as a signed integer) to a specified DateTime string

Syntax

```
DateTimeAdd (<DateTimePart> , <numeric_expr> ,<DateTime>)
```

Arguments

DateTimePart

The part of date to which DateTimeAdd adds an integer number. This table lists all valid DateTimePart arguments:

DATETIMEPART	ABBREVIATIONS
Year	"year", "yyyy", "yy"
Month	"month", "mm", "m"
Day	"day", "dd", "d"
Hour	"hour", "hh"
Minute	"minute", "mi", "n"
Second	"second", "ss", "s"
Millisecond	"millisecond", "ms"
Microsecond	"microsecond", "mcs"
Nanosecond	"nanosecond", "ns"

numeric_expr

Is a signed integer value that will be added to the DateTimePart of the specified DateTime

DateTime

UTC date and time ISO 8601 string value in the format `YYYY-MM-DDThh:mm:ss.fffffffZ` where:

FORMAT	DESCRIPTION	
YYYY	four-digit year	
MM	two-digit month (01 = January, etc.)	

FORMAT	DESCRIPTION	
DD	two-digit day of month (01 through 31)	
T	signifier for beginning of time elements	
hh	two-digit hour (00 through 23)	
mm	two-digit minutes (00 through 59)	
ss	two-digit seconds (00 through 59)	
.fffffff	seven-digit fractional seconds	
Z	UTC (Coordinated Universal Time) designator	

For more information on the ISO 8601 format, see ISO_8601

Return types

Returns a UTC date and time ISO 8601 string value in the format `YYYY-MM-DDThh:mm:ss.fffffffZ` where:

FORMAT	DESCRIPTION	
YYYY	four-digit year	
MM	two-digit month (01 = January, etc.)	
DD	two-digit day of month (01 through 31)	
T	signifier for beginning of time elements	
hh	two-digit hour (00 through 23)	
mm	two-digit minutes (00 through 59)	
ss	two-digit seconds (00 through 59)	
.fffffff	seven-digit fractional seconds	
Z	UTC (Coordinated Universal Time) designator	

Remarks

DateTimeAdd will return `undefined` for the following reasons:

- The DateTimePart value specified is invalid
- The numeric_expr specified is not a valid integer
- The DateTime in the argument or result is not a valid ISO 8601 DateTime.

Examples

The following example adds 1 month to the DateTime: `2020-07-09T23:20:13.4575530Z`

```
SELECT DateTimeAdd("mm", 1, "2020-07-09T23:20:13.4575530Z") AS OneMonthLater
```

```
[
    {
        "OneMonthLater": "2020-08-09T23:20:13.4575530Z"
    }
]
```

The following example subtracts 2 hours from the DateTime: `2020-07-09T23:20:13.4575530Z`

```
SELECT DateTimeAdd("hh", -2, "2020-07-09T23:20:13.4575530Z") AS TwoHoursEarlier
```

```
[
    {
        "TwoHoursEarlier": "2020-07-09T21:20:13.4575530Z"
    }
]
```

Next steps

- Date and time functions Azure Cosmos DB
- System functions Azure Cosmos DB
- Introduction to Azure Cosmos DB

DateTimeDiff (Azure Cosmos DB)

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: SQL API

Returns the count (as a signed integer value) of the specified DateTimePart boundaries crossed between the specified *StartDate* and *EndDate*.

Syntax

```
DateTimeDiff (<DateTimePart> , <StartDate> , <EndDate>)
```

Arguments

DateTimePart

The part of date to which DateTimeAdd adds an integer number. This table lists all valid DateTimePart arguments:

DATETIMEPART	ABBREVIATIONS
Year	"year", "yyyy", "yy"
Month	"month", "mm", "m"
Day	"day", "dd", "d"
Hour	"hour", "hh"
Minute	"minute", "mi", "n"
Second	"second", "ss", "s"
Millisecond	"millisecond", "ms"
Microsecond	"microsecond", "mcs"
Nanosecond	"nanosecond", "ns"

StartDate

UTC date and time ISO 8601 string value in the format `YYYY-MM-DDThh:mm:ss.fffffffZ` where:

FORMAT	DESCRIPTION	
YYYY	four-digit year	
MM	two-digit month (01 = January, etc.)	
DD	two-digit day of month (01 through 31)	

FORMAT	DESCRIPTION	
T	signifier for beginning of time elements	
hh	two-digit hour (00 through 23)	
mm	two-digit minutes (00 through 59)	
ss	two-digit seconds (00 through 59)	
.fffffff	seven-digit fractional seconds	
Z	UTC (Coordinated Universal Time) designator	

For more information on the ISO 8601 format, see ISO_8601

EndDate
UTC date and time ISO 8601 string value in the format `YYYY-MM-DDThh:mm:ss.fffffffZ`

Return types

Returns a signed integer value.

Remarks

DateTimeDiff will return `undefined` for the following reasons:

- The DateTimePart value specified is invalid
- The StartDate or EndDate is not a valid ISO 8601 DateTime

DateTimeDiff will always return a signed integer value and is a measurement of the number of DateTimePart boundaries crossed, not measurement of the time interval.

Examples

The following example computes the number of day boundaries crossed between `2020-01-01T01:02:03.1234527Z` and `2020-01-03T01:02:03.1234567Z`.

```
SELECT DateTimeDiff("day", "2020-01-01T01:02:03.1234527Z", "2020-01-03T01:02:03.1234567Z") AS DifferenceInDays
```

```
[
    {
        "DifferenceInDays": 2
    }
]
```

The following example computes the number of year boundaries crossed between `2028-01-01T01:02:03.1234527Z` and `2020-01-03T01:02:03.1234567Z`.

```
SELECT DateTimeDiff("yyyy", "2028-01-01T01:02:03.1234527Z", "2020-01-03T01:02:03.1234567Z") AS
DifferenceInYears
```

```
[
    {
        "DifferenceInYears": -8
    }
]
```

The following example computes the number of hour boundaries crossed between `2020-01-01T01:00:00.1234527Z` and `2020-01-01T01:59:59.1234567Z`. Even though these DateTime values are over 0.99 hours apart, `DateTimeDiff` returns 0 because no hour boundaries were crossed.

```
SELECT DateTimeDiff("hh", "2020-01-01T01:00:00.1234527Z", "2020-01-01T01:59:59.1234567Z") AS DifferenceInHours
```

```
[
    {
        "DifferenceInHours": 0
    }
]
```

Next steps

- Date and time functions Azure Cosmos DB
- System functions Azure Cosmos DB
- Introduction to Azure Cosmos DB

DateTimeFromParts (Azure Cosmos DB)

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: SQL API

Returns a string DateTime value constructed from input values.

Syntax

```
DateTimeFromParts(<numberYear>, <numberMonth>, <numberDay> [, numberHour]  [, numberMinute]  [, numberSecond]
[, numberOfFractionsOfSecond])
```

Arguments

numberYear Integer value for the year in the format `YYYY`

numberMonth
Integer value for the month in the format `MM`

numberDay
Integer value for the day in the format `DD`

numberHour (optional) Integer value for the hour in the format `hh`

numberMinute (optional) Integer value for the minute in the format `mm`

numberSecond (optional) Integer value for the second in the format `ss`

numberOfFractionsOfSecond (optional) Integer value for the fractional of a second in the format `.fffffff`

Return types

Returns a UTC date and time ISO 8601 string value in the format `YYYY-MM-DDThh:mm:ss.fffffffZ` where:

FORMAT	DESCRIPTION	
YYYY	four-digit year	
MM	two-digit month (01 = January, etc.)	
DD	two-digit day of month (01 through 31)	
T	signifier for beginning of time elements	
hh	two-digit hour (00 through 23)	
mm	two-digit minutes (00 through 59)	
ss	two-digit seconds (00 through 59)	
.fffffff	seven-digit fractional seconds	

FORMAT	DESCRIPTION	
Z	UTC (Coordinated Universal Time) designator	

For more information on the ISO 8601 format, see ISO_8601

Remarks

If the specified integers would create an invalid DateTime, DateTimeFromParts will return `undefined`.

If an optional argument isn't specified, its value will be 0.

Examples

Here's an example that only includes required arguments to construct a DateTime:

```
SELECT DateTimeFromParts(2020, 9, 4) AS DateTime
```

```
[
    {
        "DateTime": "2020-09-04T00:00:00.0000000Z"
    }
]
```

Here's another example that also uses some optional arguments to construct a DateTime:

```
SELECT DateTimeFromParts(2020, 9, 4, 10, 52) AS DateTime
```

```
[
    {
        "DateTime": "2020-09-04T10:52:00.0000000Z"
    }
]
```

Here's another example that also uses all optional arguments to construct a DateTime:

```
SELECT DateTimeFromParts(2020, 9, 4, 10, 52, 12, 3456789) AS DateTime
```

```
[
    {
        "DateTime": "2020-09-04T10:52:12.3456789Z"
    }
]
```

Next steps

- Date and time functions Azure Cosmos DB
- System functions Azure Cosmos DB
- Introduction to Azure Cosmos DB

DateTimePart (Azure Cosmos DB)

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: SQL API

Returns the value of the specified DateTimePart between the specified DateTime.

Syntax

```
DateTimePart (<DateTimePart> , <DateTime>)
```

Arguments

DateTimePart

The part of the date for which DateTimePart will return the value. This table lists all valid DateTimePart arguments:

DATETIMEPART	ABBREVIATIONS
Year	"year", "yyyy", "yy"
Month	"month", "mm", "m"
Day	"day", "dd", "d"
Hour	"hour", "hh"
Minute	"minute", "mi", "n"
Second	"second", "ss", "s"
Millisecond	"millisecond", "ms"
Microsecond	"microsecond", "mcs"
Nanosecond	"nanosecond", "ns"

DateTime

UTC date and time ISO 8601 string value in the format `YYYY-MM-DDThh:mm:ss.fffffffZ`

Return types

Returns a positive integer value.

Remarks

DateTimePart will return `undefined` for the following reasons:

- The DateTimePart value specified is invalid
- The DateTime is not a valid ISO 8601 DateTime

This system function will not utilize the index.

Examples

Here's an example that returns the integer value of the month:

```
SELECT DateTimePart("m", "2020-01-02T03:04:05.6789123Z") AS MonthValue
```

```
[
    {
        "MonthValue": 1
    }
]
```

Here's an example that returns the number of microseconds:

```
SELECT DateTimePart("mcs", "2020-01-02T03:04:05.6789123Z") AS MicrosecondsValue
```

```
[
    {
        "MicrosecondsValue": 678912
    }
]
```

Next steps

- Date and time functions Azure Cosmos DB
- System functions Azure Cosmos DB
- Introduction to Azure Cosmos DB

DateTimeToTicks (Azure Cosmos DB)

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: SQL API

Converts the specified DateTime to ticks. A single tick represents one hundred nanoseconds or one ten-millionth of a second.

Syntax

```
DateTimeToTicks (<DateTime>)
```

Arguments

DateTime
UTC date and time ISO 8601 string value in the format `YYYY-MM-DDThh:mm:ss.fffffffZ`

Return types

Returns a signed numeric value, the current number of 100-nanosecond ticks that have elapsed since the Unix epoch. In other words, DateTimeToTicks returns the number of 100-nanosecond ticks that have elapsed since 00:00:00 Thursday, 1 January 1970.

Remarks

DateTimeDateTimeToTicks will return `undefined` if the DateTime is not a valid ISO 8601 DateTime

This system function will not utilize the index.

Examples

Here's an example that returns the number of ticks:

```
SELECT DateTimeToTicks("2020-01-02T03:04:05.6789123Z") AS Ticks
```

```
[
    {
        "Ticks": 15779342456789124
    }
]
```

Here's an example that returns the number of ticks without specifying the number of fractional seconds:

```
SELECT DateTimeToTicks("2020-01-02T03:04:05Z") AS Ticks
```

```
[
    {
        "Ticks": 15779342450000000
    }
]
```

Next steps

- Date and time functions Azure Cosmos DB
- System functions Azure Cosmos DB
- Introduction to Azure Cosmos DB

DateTimeToTimestamp (Azure Cosmos DB)

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: SQL API

Converts the specified DateTime to a timestamp.

Syntax

```
DateTimeToTimestamp (<DateTime>)
```

Arguments

DateTime
UTC date and time ISO 8601 string value in the format `YYYY-MM-DDThh:mm:ss.fffffffZ` where:

FORMAT	DESCRIPTION	
YYYY	four-digit year	
MM	two-digit month (01 = January, etc.)	
DD	two-digit day of month (01 through 31)	
T	signifier for beginning of time elements	
hh	two-digit hour (00 through 23)	
mm	two-digit minutes (00 through 59)	
ss	two-digit seconds (00 through 59)	
.fffffff	seven-digit fractional seconds	
Z	UTC (Coordinated Universal Time) designator	

For more information on the ISO 8601 format, see ISO_8601

Return types

Returns a signed numeric value, the current number of milliseconds that have elapsed since the Unix epoch i.e. the number of milliseconds that have elapsed since 00:00:00 Thursday, 1 January 1970.

Remarks

DateTimeToTimestamp will return `undefined` if the DateTime value specified is invalid

Examples

The following example converts the DateTime to a timestamp:

```
SELECT DateTimeToTimestamp("2020-07-09T23:20:13.4575530Z") AS Timestamp
```

```
[
    {
        "Timestamp": 1594336813457
    }
]
```

Here's another example:

```
SELECT DateTimeToTimestamp("2020-07-09") AS Timestamp
```

```
[
    {
        "Timestamp": 1594252800000
    }
]
```

Next steps

- Date and time functions Azure Cosmos DB
- System functions Azure Cosmos DB
- Introduction to Azure Cosmos DB

GetCurrentDateTime (Azure Cosmos DB)

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: SQL API

Returns the current UTC (Coordinated Universal Time) date and time as an ISO 8601 string.

Syntax

```
GetCurrentDateTime ()
```

Return types

Returns the current UTC date and time ISO 8601 string value in the format `YYYY-MM-DDThh:mm:ss.fffffffZ` where:

FORMAT	DESCRIPTION	
YYYY	four-digit year	
MM	two-digit month (01 = January, etc.)	
DD	two-digit day of month (01 through 31)	
T	signifier for beginning of time elements	
hh	two-digit hour (00 through 23)	
mm	two-digit minutes (00 through 59)	
ss	two-digit seconds (00 through 59)	
.fffffff	seven-digit fractional seconds	
Z	UTC (Coordinated Universal Time) designator	

For more information on the ISO 8601 format, see ISO_8601

Remarks

GetCurrentDateTime() is a nondeterministic function. The result returned is UTC. Precision is 7 digits, with an accuracy of 100 nanoseconds.

This system function will not utilize the index.

Examples

The following example shows how to get the current UTC Date Time using the GetCurrentDateTime() built-in function.

```
SELECT GetCurrentDateTime() AS currentUtcDateTime
```

Here is an example result set.

```
[{
  "currentUtcDateTime": "2019-05-03T20:36:17.1234567Z"
}]
```

Next steps

- Date and time functions Azure Cosmos DB
- System functions Azure Cosmos DB
- Introduction to Azure Cosmos DB

GetCurrentTicks (Azure Cosmos DB)

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: SQL API

Returns the number of 100-nanosecond ticks that have elapsed since 00:00:00 Thursday, 1 January 1970.

Syntax

```
GetCurrentTicks ()
```

Return types

Returns a signed numeric value, the current number of 100-nanosecond ticks that have elapsed since the Unix epoch. In other words, GetCurrentTicks returns the number of 100 nanosecond ticks that have elapsed since 00:00:00 Thursday, 1 January 1970.

Remarks

GetCurrentTicks() is a nondeterministic function. The result returned is UTC (Coordinated Universal Time).

This system function will not utilize the index.

Examples

Here's an example that returns the current time, measured in ticks:

```
SELECT GetCurrentTicks() AS CurrentTimeInTicks
```

```
[
    {
        "CurrentTimeInTicks": 15973607943002652
    }
]
```

Next steps

- Date and time functions Azure Cosmos DB
- System functions Azure Cosmos DB
- Introduction to Azure Cosmos DB

GetCurrentTimestamp (Azure Cosmos DB)

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: SQL API

Returns the number of milliseconds that have elapsed since 00:00:00 Thursday, 1 January 1970.

Syntax

```
GetCurrentTimestamp ()
```

Return types

Returns a signed numeric value, the current number of milliseconds that have elapsed since the Unix epoch i.e. the number of milliseconds that have elapsed since 00:00:00 Thursday, 1 January 1970.

Remarks

GetCurrentTimestamp() is a nondeterministic function. The result returned is UTC (Coordinated Universal Time).

This system function will not utilize the index.

Examples

The following example shows how to get the current timestamp using the GetCurrentTimestamp() built-in function.

```
SELECT GetCurrentTimestamp() AS currentUtcTimestamp
```

Here is an example result set.

```
[{
  "currentUtcTimestamp": 1556916469065
}]
```

Next steps

- Date and time functions Azure Cosmos DB
- System functions Azure Cosmos DB
- Introduction to Azure Cosmos DB

TicksToDateTime (Azure Cosmos DB)

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: SQL API

Converts the specified ticks value to a DateTime.

Syntax

```
TicksToDateTime (<Ticks>)
```

Arguments

Ticks

A signed numeric value, the current number of 100 nanosecond ticks that have elapsed since the Unix epoch. In other words, it is the number of 100 nanosecond ticks that have elapsed since 00:00:00 Thursday, 1 January 1970.

Return types

Returns the UTC date and time ISO 8601 string value in the format `YYYY-MM-DDThh:mm:ss.fffffffZ` where:

FORMAT	DESCRIPTION	
YYYY	four-digit year	
MM	two-digit month (01 = January, etc.)	
DD	two-digit day of month (01 through 31)	
T	signifier for beginning of time elements	
hh	two-digit hour (00 through 23)	
mm	two-digit minutes (00 through 59)	
ss	two-digit seconds (00 through 59)	
.fffffff	seven-digit fractional seconds	
Z	UTC (Coordinated Universal Time) designator	

For more information on the ISO 8601 format, see ISO_8601

Remarks

TicksToDateTime will return `undefined` if the ticks value specified is invalid.

Examples

The following example converts the ticks to a DateTime:

```
SELECT TicksToDateTime(15943368134575530) AS DateTime
```

```
[
    {
        "DateTime": "2020-07-09T23:20:13.4575530Z"
    }
]
```

Next steps

- Date and time functions Azure Cosmos DB
- System functions Azure Cosmos DB
- Introduction to Azure Cosmos DB

TimestampToDateTime (Azure Cosmos DB)

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: SQL API

Converts the specified timestamp value to a DateTime.

Syntax

```
TimestampToDateTime (<Timestamp>)
```

Arguments

Timestamp

A signed numeric value, the current number of milliseconds that have elapsed since the Unix epoch. In other words, the number of milliseconds that have elapsed since 00:00:00 Thursday, 1 January 1970.

Return types

Returns the UTC date and time ISO 8601 string value in the format `YYYY-MM-DDThh:mm:ss.fffffffZ` where:

FORMAT	DESCRIPTION	
YYYY	four-digit year	
MM	two-digit month (01 = January, etc.)	
DD	two-digit day of month (01 through 31)	
T	signifier for beginning of time elements	
hh	two-digit hour (00 through 23)	
mm	two-digit minutes (00 through 59)	
ss	two-digit seconds (00 through 59)	
.fffffff	seven-digit fractional seconds	
Z	UTC (Coordinated Universal Time) designator	

For more information on the ISO 8601 format, see ISO_8601

Remarks

TimestampToDateTime will return `undefined` if the timestamp value specified is invalid.

Examples

The following example converts the timestamp to a DateTime:

```
SELECT TimestampToDateTime(1594227912345) AS DateTime
```

```
[
    {
        "DateTime": "2020-07-08T17:05:12.3450000Z"
    }
]
```

Next steps

- Date and time functions Azure Cosmos DB
- System functions Azure Cosmos DB
- Introduction to Azure Cosmos DB

Mathematical functions (Azure Cosmos DB)

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: SQL API

The mathematical functions each perform a calculation, based on input values that are provided as arguments, and return a numeric value.

You can run queries like the following example:

```
SELECT VALUE ABS(-4)
```

The result is:

```
[4]
```

Functions

The following supported built-in mathematical functions perform a calculation, usually based on input arguments, and return a numeric expression:

- ABS
- ACOS
- ASIN
- ATAN
- ATN2
- CEILING
- COS
- COT
- DEGREES
- EXP
- FLOOR
- LOG
- LOG10
- PI
- POWER
- RADIANS
- RAND
- ROUND
- SIGN
- SIN
- SQRT
- SQUARE
- TAN
- TRUNC

All mathematical functions, except for RAND, are deterministic functions. This means they return the same results each time they are called with a specific set of input values.

Next steps

- System functions Azure Cosmos DB
- Introduction to Azure Cosmos DB
- User Defined Functions
- Aggregates

ABS (Azure Cosmos DB)

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: SQL API

Returns the absolute (positive) value of the specified numeric expression.

Syntax

```
ABS (<numeric_expr>)
```

Arguments

numeric_expr
Is a numeric expression.

Return types

Returns a numeric expression.

Examples

The following example shows the results of using the `ABS` function on three different numbers.

```
SELECT ABS(-1) AS abs1, ABS(0) AS abs2, ABS(1) AS abs3
```

Here is the result set.

```
[{abs1: 1, abs2: 0, abs3: 1}]
```

Remarks

This system function will benefit from a range index.

Next steps

- Mathematical functions Azure Cosmos DB
- System functions Azure Cosmos DB
- Introduction to Azure Cosmos DB

ACOS (Azure Cosmos DB)

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: SQL API

Returns the angle, in radians, whose cosine is the specified numeric expression; also called arccosine.

Syntax

```
ACOS(<numeric_expr>)
```

Arguments

numeric_expr
Is a numeric expression.

Return types

Returns a numeric expression.

Examples

The following example returns the `ACOS` of -1.

```
SELECT ACOS(-1) AS acos
```

Here is the result set.

```
[{"acos": 3.1415926535897931}]
```

Remarks

This system function will not utilize the index.

Next steps

- Mathematical functions Azure Cosmos DB
- System functions Azure Cosmos DB
- Introduction to Azure Cosmos DB

ASIN (Azure Cosmos DB)

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: SQL API

Returns the angle, in radians, whose sine is the specified numeric expression. This is also called arcsine.

Syntax

```
ASIN(<numeric_expr>)
```

Arguments

numeric_expr
Is a numeric expression.

Return types

Returns a numeric expression.

Examples

The following example returns the `ASIN` of -1.

```
SELECT ASIN(-1) AS asin
```

Here is the result set.

```
[{"asin": -1.5707963267948966}]
```

Remarks

This system function will not utilize the index.

Next steps

- Mathematical functions Azure Cosmos DB
- System functions Azure Cosmos DB
- Introduction to Azure Cosmos DB

ATAN (Azure Cosmos DB)

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: SQL API

Returns the angle, in radians, whose tangent is the specified numeric expression. This is also called arctangent.

Syntax

```
ATAN(<numeric_expr>)
```

Arguments

numeric_expr
Is a numeric expression.

Return types

Returns a numeric expression.

Examples

The following example returns the `ATAN` of the specified value.

```
SELECT ATAN(-45.01) AS atan
```

Here is the result set.

```
[{"atan": -1.5485826962062663}]
```

Remarks

This system function will not utilize the index.

Next steps

- Mathematical functions Azure Cosmos DB
- System functions Azure Cosmos DB
- Introduction to Azure Cosmos DB

ATN2 (Azure Cosmos DB)

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: SQL API

Returns the principal value of the arc tangent of y/x, expressed in radians.

Syntax

```
ATN2(<numeric_expr>, <numeric_expr>)
```

Arguments

numeric_expr
Is a numeric expression.

Return types

Returns a numeric expression.

Examples

The following example calculates the ATN2 for the specified x and y components.

```
SELECT ATN2(35.175643, 129.44) AS atn2
```

Here is the result set.

```
[{"atn2": 1.3054517947300646}]
```

Remarks

This system function will not utilize the index.

Next steps

- Mathematical functions Azure Cosmos DB
- System functions Azure Cosmos DB
- Introduction to Azure Cosmos DB

CEILING (Azure Cosmos DB)

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: SQL API

Returns the smallest integer value greater than, or equal to, the specified numeric expression.

Syntax

```
CEILING (<numeric_expr>)
```

Arguments

numeric_expr
Is a numeric expression.

Return types

Returns a numeric expression.

Examples

The following example shows positive numeric, negative, and zero values with the `CEILING` function.

```
SELECT CEILING(123.45) AS c1, CEILING(-123.45) AS c2, CEILING(0.0) AS c3
```

Here is the result set.

```
[{c1: 124, c2: -123, c3: 0}]
```

Remarks

This system function will benefit from a range index.

Next steps

- Mathematical functions Azure Cosmos DB
- System functions Azure Cosmos DB
- Introduction to Azure Cosmos DB

COS (Azure Cosmos DB)

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: SQL API

Returns the trigonometric cosine of the specified angle, in radians, in the specified expression.

Syntax

```
COS(<numeric_expr>)
```

Arguments

numeric_expr
Is a numeric expression.

Return types

Returns a numeric expression.

Examples

The following example calculates the `cos` of the specified angle.

```
SELECT COS(14.78) AS cos
```

Here is the result set.

```
[{"cos": -0.59946542619465426}]
```

Remarks

This system function will not utilize the index.

Next steps

- Mathematical functions Azure Cosmos DB
- System functions Azure Cosmos DB
- Introduction to Azure Cosmos DB

COT (Azure Cosmos DB)

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: SQL API

Returns the trigonometric cotangent of the specified angle, in radians, in the specified numeric expression.

Syntax

```
COT(<numeric_expr>)
```

Arguments

numeric_expr
Is a numeric expression.

Return types

Returns a numeric expression.

Examples

The following example calculates the `COT` of the specified angle.

```
SELECT COT(124.1332) AS cot
```

Here is the result set.

```
[{"cot": -0.040311998371148884}]
```

Remarks

This system function will not utilize the index.

Next steps

- Mathematical functions Azure Cosmos DB
- System functions Azure Cosmos DB
- Introduction to Azure Cosmos DB

DEGREES (Azure Cosmos DB)

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: SQL API

Returns the corresponding angle in degrees for an angle specified in radians.

Syntax

```
DEGREES (<numeric_expr>)
```

Arguments

numeric_expr
Is a numeric expression.

Return types

Returns a numeric expression.

Examples

The following example returns the number of degrees in an angle of PI/2 radians.

```
SELECT DEGREES(PI()/2) AS degrees
```

Here is the result set.

```
[{"degrees": 90}]
```

Remarks

This system function will not utilize the index.

Next steps

- Mathematical functions Azure Cosmos DB
- System functions Azure Cosmos DB
- Introduction to Azure Cosmos DB

EXP (Azure Cosmos DB)

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: SQL API

Returns the exponential value of the specified numeric expression.

Syntax

```
EXP (<numeric_expr>)
```

Arguments

numeric_expr
Is a numeric expression.

Return types

Returns a numeric expression.

Remarks

The constant **e** (2.718281...), is the base of natural logarithms.

The exponent of a number is the constant **e** raised to the power of the number. For example, EXP(1.0) = e^1.0 = 2.71828182845905 and EXP(10) = e^10 = 22026.4657948067.

The exponential of the natural logarithm of a number is the number itself: EXP (LOG (n)) = n. And the natural logarithm of the exponential of a number is the number itself: LOG (EXP (n)) = n.

Examples

The following example declares a variable and returns the exponential value of the specified variable (10).

```
SELECT EXP(10) AS exp
```

Here is the result set.

```
[{exp: 22026.465794806718}]
```

The following example returns the exponential value of the natural logarithm of 20 and the natural logarithm of the exponential of 20. Because these functions are inverse functions of one another, the return value with rounding for floating point math in both cases is 20.

```
SELECT EXP(LOG(20)) AS exp1, LOG(EXP(20)) AS exp2
```

Here is the result set.

```
[{exp1: 19.999999999999996, exp2: 20}]
```

Next steps

- Mathematical functions Azure Cosmos DB
- System functions Azure Cosmos DB
- Introduction to Azure Cosmos DB

FLOOR (Azure Cosmos DB)

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: SQL API

Returns the largest integer less than or equal to the specified numeric expression.

Syntax

```
FLOOR (<numeric_expr>)
```

Arguments

numeric_expr
Is a numeric expression.

Return types

Returns a numeric expression.

Examples

The following example shows positive numeric, negative, and zero values with the `FLOOR` function.

```
SELECT FLOOR(123.45) AS fl1, FLOOR(-123.45) AS fl2, FLOOR(0.0) AS fl3
```

Here is the result set.

```
[{fl1: 123, fl2: -124, fl3: 0}]
```

Remarks

This system function will benefit from a range index.

Next steps

- Mathematical functions Azure Cosmos DB
- System functions Azure Cosmos DB
- Introduction to Azure Cosmos DB

LOG (Azure Cosmos DB)

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: SQL API

Returns the natural logarithm of the specified numeric expression.

Syntax

```
LOG (<numeric_expr> [, <base>])
```

Arguments

numeric_expr
Is a numeric expression.

base
Optional numeric argument that sets the base for the logarithm.

Return types

Returns a numeric expression.

Remarks

By default, LOG() returns the natural logarithm. You can change the base of the logarithm to another value by using the optional base parameter.

The natural logarithm is the logarithm to the base **e**, where **e** is an irrational constant approximately equal to 2.718281828.

The natural logarithm of the exponential of a number is the number itself: LOG(EXP(n)) = n. And the exponential of the natural logarithm of a number is the number itself: EXP(LOG(n)) = n.

This system function will not utilize the index.

Examples

The following example declares a variable and returns the logarithm value of the specified variable (10).

```
SELECT LOG(10) AS log
```

Here is the result set.

```
[{log: 2.3025850929940459}]
```

The following example calculates the `LOG` for the exponent of a number.

```
SELECT EXP(LOG(10)) AS expLog
```

Here is the result set.

```
[{expLog: 10.000000000000002}]
```

Next steps

- Mathematical functions Azure Cosmos DB
- System functions Azure Cosmos DB
- Introduction to Azure Cosmos DB

LOG10 (Azure Cosmos DB)

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: SQL API

Returns the base-10 logarithm of the specified numeric expression.

Syntax

```
LOG10 (<numeric_expr>)
```

Arguments

numeric_expression
Is a numeric expression.

Return types

Returns a numeric expression.

Remarks

The LOG10 and POWER functions are inversely related to one another. For example, 10 ^ LOG10(n) = n. This system function will not utilize the index.

Examples

The following example declares a variable and returns the LOG10 value of the specified variable (100).

```
SELECT LOG10(100) AS log10
```

Here is the result set.

```
[{log10: 2}]
```

Next steps

- Mathematical functions Azure Cosmos DB
- System functions Azure Cosmos DB
- Introduction to Azure Cosmos DB

PI (Azure Cosmos DB)

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: SQL API

Returns the constant value of PI.

Syntax

```
PI ()
```

Return types

Returns a numeric expression.

Examples

The following example returns the value of `PI`.

```
SELECT PI() AS pi
```

Here is the result set.

```
[{"pi": 3.1415926535897931}]
```

Next steps

- Mathematical functions Azure Cosmos DB
- System functions Azure Cosmos DB
- Introduction to Azure Cosmos DB

POWER (Azure Cosmos DB)

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: SQL API

Returns the value of the specified expression to the specified power.

Syntax

```
POWER (<numeric_expr1>, <numeric_expr2>)
```

Arguments

numeric_expr1
Is a numeric expression.

numeric_expr2
Is the power to which to raise *numeric_expr1*.

Return types

Returns a numeric expression.

Examples

The following example demonstrates raising a number to the power of 3 (the cube of the number).

```
SELECT POWER(2, 3) AS pow1, POWER(2.5, 3) AS pow2
```

Here is the result set.

```
[{pow1: 8, pow2: 15.625}]
```

Next steps

- Mathematical functions Azure Cosmos DB
- System functions Azure Cosmos DB
- Introduction to Azure Cosmos DB

RADIANS (Azure Cosmos DB)

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: SQL API

Returns radians when a numeric expression, in degrees, is entered.

Syntax

```
RADIANS (<numeric_expr>)
```

Arguments

numeric_expr
Is a numeric expression.

Return types

Returns a numeric expression.

Examples

The following example takes a few angles as input and returns their corresponding radian values.

```
SELECT RADIANS(-45.01) AS r1, RADIANS(-181.01) AS r2, RADIANS(0) AS r3, RADIANS(0.1472738) AS r4,
RADIANS(197.1099392) AS r5
```

Here is the result set.

```
[{
       "r1": -0.7855726963226477,
       "r2": -3.1592204790349356,
       "r3": 0,
       "r4": 0.0025704127119236249,
       "r5": 3.4402174274458375
   }]
```

Remarks

This system function will not utilize the index.

Next steps

- Mathematical functions Azure Cosmos DB
- System functions Azure Cosmos DB
- Introduction to Azure Cosmos DB

RAND (Azure Cosmos DB)

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: SQL API

Returns a randomly generated numeric value from [0,1).

Syntax

```
RAND ()
```

Return types

Returns a numeric expression.

Remarks

`RAND` is a nondeterministic function. Repetitive calls of `RAND` do not return the same results. This system function will not utilize the index.

Examples

The following example returns a randomly generated numeric value.

```
SELECT RAND() AS rand
```

Here is the result set.

```
[{"rand": 0.87860053195618093}]
```

Next steps

- Mathematical functions Azure Cosmos DB
- System functions Azure Cosmos DB
- Introduction to Azure Cosmos DB

ROUND (Azure Cosmos DB)

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: SQL API

Returns a numeric value, rounded to the closest integer value.

Syntax

```
ROUND(<numeric_expr>)
```

Arguments

numeric_expr
Is a numeric expression.

Return types

Returns a numeric expression.

Remarks

The rounding operation performed follows midpoint rounding away from zero. If the input is a numeric expression which falls exactly between two integers then the result will be the closest integer value away from zero. This system function will benefit from a range index.

<NUMERIC_EXPR>	ROUNDED	
-6.5000	-7	
-0.5	-1	
0.5	1	
6.5000	7	

Examples

The following example rounds the following positive and negative numbers to the nearest integer.

```
SELECT ROUND(2.4) AS r1, ROUND(2.6) AS r2, ROUND(2.5) AS r3, ROUND(-2.4) AS r4, ROUND(-2.6) AS r5
```

Here is the result set.

```
[{r1: 2, r2: 3, r3: 3, r4: -2, r5: -3}]
```

Next steps

- Mathematical functions Azure Cosmos DB
- System functions Azure Cosmos DB
- Introduction to Azure Cosmos DB

SIGN (Azure Cosmos DB)

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: SQL API

Returns the positive (+1), zero (0), or negative (-1) sign of the specified numeric expression.

Syntax

```
SIGN(<numeric_expr>)
```

Arguments

numeric_expr
Is a numeric expression.

Return types

Returns a numeric expression.

Examples

The following example returns the `SIGN` values of numbers from -2 to 2.

```
SELECT SIGN(-2) AS s1, SIGN(-1) AS s2, SIGN(0) AS s3, SIGN(1) AS s4, SIGN(2) AS s5
```

Here is the result set.

```
[{s1: -1, s2: -1, s3: 0, s4: 1, s5: 1}]
```

Remarks

This system function will not utilize the index.

Next steps

- Mathematical functions Azure Cosmos DB
- System functions Azure Cosmos DB
- Introduction to Azure Cosmos DB

SIN (Azure Cosmos DB)

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: SQL API

Returns the trigonometric sine of the specified angle, in radians, in the specified expression.

Syntax

```
SIN(<numeric_expr>)
```

Arguments

numeric_expr
Is a numeric expression.

Return types

Returns a numeric expression.

Examples

The following example calculates the `SIN` of the specified angle.

```
SELECT SIN(45.175643) AS sin
```

Here is the result set.

```
[{"sin": 0.929607286611012}]
```

Remarks

This system function will not utilize the index.

Next steps

- Mathematical functions Azure Cosmos DB
- System functions Azure Cosmos DB
- Introduction to Azure Cosmos DB

SQRT (Azure Cosmos DB)

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: SQL API

Returns the square root of the specified numeric value.

Syntax

```
SQRT(<numeric_expr>)
```

Arguments

numeric_expr
Is a numeric expression.

Return types

Returns a numeric expression.

Examples

The following example returns the square roots of numbers 1-3.

```
SELECT SQRT(1) AS s1, SQRT(2.0) AS s2, SQRT(3) AS s3
```

Here is the result set.

```
[{s1: 1, s2: 1.4142135623730952, s3: 1.7320508075688772}]
```

Remarks

This system function will not utilize the index.

Next steps

- Mathematical functions Azure Cosmos DB
- System functions Azure Cosmos DB
- Introduction to Azure Cosmos DB

SQUARE (Azure Cosmos DB)

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: SQL API

Returns the square of the specified numeric value.

Syntax

```
SQUARE(<numeric_expr>)
```

Arguments

numeric_expr
Is a numeric expression.

Return types

Returns a numeric expression.

Examples

The following example returns the squares of numbers 1-3.

```
SELECT SQUARE(1) AS s1, SQUARE(2.0) AS s2, SQUARE(3) AS s3
```

Here is the result set.

```
[{s1: 1, s2: 4, s3: 9}]
```

Remarks

This system function will not utilize the index.

Next steps

- Mathematical functions Azure Cosmos DB
- System functions Azure Cosmos DB
- Introduction to Azure Cosmos DB

TAN (Azure Cosmos DB)

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: SQL API

Returns the tangent of the specified angle, in radians, in the specified expression.

Syntax

```
TAN (<numeric_expr>)
```

Arguments

numeric_expr
Is a numeric expression.

Return types

Returns a numeric expression.

Examples

The following example calculates the tangent of PI()/2.

```
SELECT TAN(PI()/2) AS tan
```

Here is the result set.

```
[{"tan": 16331239353195370 }]
```

Remarks

This system function will not utilize the index.

Next steps

- Mathematical functions Azure Cosmos DB
- System functions Azure Cosmos DB
- Introduction to Azure Cosmos DB

TRUNC (Azure Cosmos DB)

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: SQL API

Returns a numeric value, truncated to the closest integer value.

Syntax

```
TRUNC(<numeric_expr>)
```

Arguments

numeric_expr
Is a numeric expression.

Return types

Returns a numeric expression.

Examples

The following example truncates the following positive and negative numbers to the nearest integer value.

```
SELECT TRUNC(2.4) AS t1, TRUNC(2.6) AS t2, TRUNC(2.5) AS t3, TRUNC(-2.4) AS t4, TRUNC(-2.6) AS t5
```

Here is the result set.

```
[{t1: 2, t2: 2, t3: 2, t4: -2, t5: -2}]
```

Remarks

This system function will benefit from a range index.

Next steps

- Mathematical functions Azure Cosmos DB
- System functions Azure Cosmos DB
- Introduction to Azure Cosmos DB

Spatial functions (Azure Cosmos DB)

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: SQL API

Cosmos DB supports the following Open Geospatial Consortium (OGC) built-in functions for geospatial querying.

Functions

The following scalar functions perform an operation on a spatial object input value and return a numeric or Boolean value.

- ST_DISTANCE
- ST_INTERSECTS
- ST_ISVALID
- ST_ISVALIDDETAILED
- ST_WITHIN

Next steps

- System functions Azure Cosmos DB
- Introduction to Azure Cosmos DB
- User Defined Functions
- Aggregates

ST_DISTANCE (Azure Cosmos DB)

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: SQL API

Returns the distance between the two GeoJSON Point, Polygon, MultiPolygon or LineString expressions. To learn more, see the Geospatial and GeoJSON location data article.

Syntax

```
ST_DISTANCE (<spatial_expr>, <spatial_expr>)
```

Arguments

spatial_expr
Is any valid GeoJSON Point, Polygon, or LineString object expression.

Return types

Returns a numeric expression containing the distance. This is expressed in meters for the default reference system.

Examples

The following example shows how to return all family documents that are within 30 km of the specified location using the `ST_DISTANCE` built-in function. .

```
SELECT f.id
FROM Families f
WHERE ST_DISTANCE(f.location, {'type': 'Point', 'coordinates':[31.9, -4.8]}) < 30000
```

Here is the result set.

```
[{
  "id": "WakefieldFamily"
}]
```

Remarks

This system function will benefit from a geospatial index.

Next steps

- Spatial functions Azure Cosmos DB
- System functions Azure Cosmos DB
- Introduction to Azure Cosmos DB

ST_WITHIN (Azure Cosmos DB)

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: SQL API

Returns a Boolean expression indicating whether the GeoJSON object (Point, Polygon, MultiPolygon, or LineString) specified in the first argument is within the GeoJSON (Point, Polygon, MultiPolygon, or LineString) in the second argument.

Syntax

```
ST_WITHIN (<spatial_expr>, <spatial_expr>)
```

Arguments

spatial_expr
Is a GeoJSON Point, Polygon, or LineString object expression.

Return types

Returns a Boolean value.

Examples

The following example shows how to find all family documents within a polygon using `ST_WITHIN`.

```
SELECT f.id
FROM Families f
WHERE ST_WITHIN(f.location, {
    'type':'Polygon',
    'coordinates': [[[31.8, -5], [32, -5], [32, -4.7], [31.8, -4.7], [31.8, -5]]]
})
```

Here is the result set.

```
[{ "id": "WakefieldFamily" }]
```

Remarks

This system function will benefit from a geospatial index.

Next steps

- Spatial functions Azure Cosmos DB
- System functions Azure Cosmos DB
- Introduction to Azure Cosmos DB

ST_INTERSECTS (Azure Cosmos DB)

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: SQL API

Returns a Boolean expression indicating whether the GeoJSON object (Point, Polygon, MultiPolygon, or LineString) specified in the first argument intersects the GeoJSON (Point, Polygon, MultiPolygon, or LineString) in the second argument.

Syntax

```
ST_INTERSECTS (<spatial_expr>, <spatial_expr>)
```

Arguments

spatial_expr
Is a GeoJSON Point, Polygon, or LineString object expression.

Return types

Returns a Boolean value.

Examples

The following example shows how to find all areas that intersect with the given polygon.

```
SELECT a.id
FROM Areas a
WHERE ST_INTERSECTS(a.location, {
    'type':'Polygon',
    'coordinates': [[[31.8, -5], [32, -5], [32, -4.7], [31.8, -4.7], [31.8, -5]]]
})
```

Here is the result set.

```
[{ "id": "IntersectingPolygon" }]
```

Remarks

This system function will benefit from a geospatial index.

Next steps

- Spatial functions Azure Cosmos DB
- System functions Azure Cosmos DB
- Introduction to Azure Cosmos DB

ST_ISVALID (Azure Cosmos DB)

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: SQL API

Returns a Boolean value indicating whether the specified GeoJSON Point, Polygon, MultiPolygon, or LineString expression is valid.

Syntax

```
ST_ISVALID(<spatial_expr>)
```

Arguments

spatial_expr
Is a GeoJSON Point, Polygon, or LineString expression.

Return types

Returns a Boolean expression.

Examples

The following example shows how to check if a point is valid using ST_VALID.

For example, this point has a latitude value that's not in the valid range of values [-90, 90], so the query returns false.

For polygons, the GeoJSON specification requires that the last coordinate pair provided should be the same as the first, to create a closed shape. Points within a polygon must be specified in counter-clockwise order. A polygon specified in clockwise order represents the inverse of the region within it.

```
SELECT ST_ISVALID({ "type": "Point", "coordinates": [31.9, -132.8] }) AS b
```

Here is the result set.

```
[{ "b": false }]
```

Next steps

- Spatial functions Azure Cosmos DB
- System functions Azure Cosmos DB
- Introduction to Azure Cosmos DB

ST_ISVALIDDETAILED (Azure Cosmos DB)

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: SQL API

Returns a JSON value containing a Boolean value if the specified GeoJSON Point, Polygon, or LineString expression is valid, and if invalid, additionally the reason as a string value.

Syntax

```
ST_ISVALIDDETAILED(<spatial_expr>)
```

Arguments

spatial_expr
Is a GeoJSON point or polygon expression.

Return types

Returns a JSON value containing a Boolean value if the specified GeoJSON point or polygon expression is valid, and if invalid, additionally the reason as a string value.

Examples

The following example how to check validity (with details) using `ST_ISVALIDDETAILED`.

```
SELECT ST_ISVALIDDETAILED({
  "type": "Polygon",
  "coordinates": [[ [ 31.8, -5 ], [ 31.8, -4.7 ], [ 32, -4.7 ], [ 32, -5 ] ]]
}) AS b
```

Here is the result set.

```
[{
  "b": {
    "valid": false,
    "reason": "The Polygon input is not valid because the start and end points of the ring number 1 are not
the same. Each ring of a polygon must have the same start and end points."
  }
}]
```

Next steps

- Spatial functions Azure Cosmos DB
- System functions Azure Cosmos DB
- Introduction to Azure Cosmos DB

String functions (Azure Cosmos DB)

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: SQL API

The string functions let you perform operations on strings in Azure Cosmos DB.

Functions

The following scalar functions perform an operation on a string input value and return a string, numeric, or Boolean value:

- CONCAT
- CONTAINS
- ENDSWITH
- INDEX_OF
- LEFT
- LENGTH
- LOWER
- LTRIM
- REGEXMATCH
- REPLACE
- REPLICATE
- REVERSE
- RIGHT
- RTRIM
- STARTSWITH
- StringToArray
- StringToBoolean
- StringToNull
- StringToNumber
- StringToObject
- SUBSTRING
- ToString
- TRIM
- UPPER

Next steps

- System functions Azure Cosmos DB
- Introduction to Azure Cosmos DB
- User Defined Functions
- Aggregates

CONCAT (Azure Cosmos DB)

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: SQL API

Returns a string that is the result of concatenating two or more string values.

Syntax

```
CONCAT(<str_expr1>, <str_expr2> [, <str_exprN>])
```

Arguments

str_expr
Is a string expression to concatenate to the other values. The `CONCAT` function requires at least two *str_expr* arguments.

Return types

Returns a string expression.

Examples

The following example returns the concatenated string of the specified values.

```
SELECT CONCAT("abc", "def") AS concat
```

Here is the result set.

```
[{"concat": "abcdef"}]
```

Remarks

This system function will not utilize the index.

Next steps

- String functions Azure Cosmos DB
- System functions Azure Cosmos DB
- Introduction to Azure Cosmos DB

CONTAINS (Azure Cosmos DB)

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: SQL API

Returns a Boolean indicating whether the first string expression contains the second.

Syntax

```
CONTAINS(<str_expr1>, <str_expr2> [, <bool_expr>])
```

Arguments

str_expr1
Is the string expression to be searched.

str_expr2
Is the string expression to find.

bool_expr Optional value for ignoring case. When set to true, CONTAINS will do a case-insensitive search. When unspecified, this value is false.

Return types

Returns a Boolean expression.

Examples

The following example checks if "abc" contains "ab" and if "abc" contains "A".

```
SELECT CONTAINS("abc", "ab", false) AS c1, CONTAINS("abc", "A", false) AS c2, CONTAINS("abc", "A", true) AS c3
```

Here is the result set.

```
[
    {
        "c1": true,
        "c2": false,
        "c3": true
    }
]
```

Remarks

This system function will benefit from a range index.

The RU consumption of Contains will increase as the cardinality of the property in the system function increases. In other words, if you are checking whether a property value contains a certain string, the query RU charge will depend on the number of possible values for that property.

For example, consider two properties: town and country. The cardinality of town is 5,000 and the cardinality of country is 200. Here are two example queries:

```
SELECT * FROM c WHERE CONTAINS(c.town, "Red", false)
```

```
SELECT * FROM c WHERE CONTAINS(c.country, "States", false)
```

The first query will likely use more RUs than the second query because the cardinality of town is higher than country.

If the property size in Contains is greater than 1 KB for some documents, the query engine will need to load those documents. In this case, the query engine won't be able to fully evaluate Contains with an index. The RU charge for Contains will be high if you have a large number of documents with property sizes more than 1 KB.

Next steps

- String functions Azure Cosmos DB
- System functions Azure Cosmos DB
- Introduction to Azure Cosmos DB

ENDSWITH (Azure Cosmos DB)

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: SQL API

Returns a Boolean indicating whether the first string expression ends with the second.

Syntax

```
ENDSWITH(<str_expr1>, <str_expr2> [, <bool_expr>])
```

Arguments

str_expr1
Is a string expression.

str_expr2
Is a string expression to be compared to the end of *str_expr1*.

bool_expr Optional value for ignoring case. When set to true, ENDSWITH will do a case-insensitive search. When unspecified, this value is false.

Return types

Returns a Boolean expression.

Examples

The following example checks if the string "abc" ends with "b" and "bC".

```
SELECT ENDSWITH("abc", "b", false) AS e1, ENDSWITH("abc", "bC", false) AS e2, ENDSWITH("abc", "bC", true) AS
e3
```

Here is the result set.

```
[
    {
        "e1": false,
        "e2": false,
        "e3": true
    }
]
```

Remarks

This system function will benefit from a range index.

The RU consumption of EndsWith will increase as the cardinality of the property in the system function increases. In other words, if you are checking whether a property value ends with a certain string, the query RU charge will depend on the number of possible values for that property.

For example, consider two properties: town and country. The cardinality of town is 5,000 and the cardinality of country is 200. Here are two example queries:

```
SELECT * FROM c WHERE ENDSWITH(c.town, "York", false)
```

```
SELECT * FROM c WHERE ENDSWITH(c.country, "States", false)
```

The first query will likely use more RUs than the second query because the cardinality of town is higher than country.

If the property size in EndsWith is greater than 1 KB for some documents, the query engine will need to load those documents. In this case, the query engine won't be able to fully evaluate EndsWith with an index. The RU charge for EndsWith will be high if you have a large number of documents with property sizes greater than 1 KB.

Next steps

- String functions Azure Cosmos DB
- System functions Azure Cosmos DB
- Introduction to Azure Cosmos DB

INDEX_OF (Azure Cosmos DB)

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: SQL API

Returns the starting position of the first occurrence of the second string expression within the first specified string expression, or -1 if the string is not found.

Syntax

```
INDEX_OF(<str_expr1>, <str_expr2> [, <numeric_expr>])
```

Arguments

str_expr1
Is the string expression to be searched.

str_expr2
Is the string expression to search for.

numeric_expr Optional numeric expression that sets the position the search will start. The first position in *str_expr1* is 0.

Return types

Returns a numeric expression.

Examples

The following example returns the index of various substrings inside "abc".

```
SELECT INDEX_OF("abc", "ab") AS i1, INDEX_OF("abc", "b") AS i2, INDEX_OF("abc", "c") AS i3
```

Here is the result set.

```
[{"i1": 0, "i2": 1, "i3": -1}]
```

Next steps

- String functions Azure Cosmos DB
- System functions Azure Cosmos DB
- Introduction to Azure Cosmos DB

LEFT (Azure Cosmos DB)

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: SQL API

Returns the left part of a string with the specified number of characters.

Syntax

```
LEFT(<str_expr>, <num_expr>)
```

Arguments

str_expr
Is the string expression to extract characters from.

num_expr
Is a numeric expression which specifies the number of characters.

Return types

Returns a string expression.

Examples

The following example returns the left part of "abc" for various length values.

```
SELECT LEFT("abc", 1) AS l1, LEFT("abc", 2) AS l2
```

Here is the result set.

```
[{"l1": "a", "l2": "ab"}]
```

Remarks

This system function will benefit from a range index.

Next steps

- String functions Azure Cosmos DB
- System functions Azure Cosmos DB
- Introduction to Azure Cosmos DB

LENGTH (Azure Cosmos DB)

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: SQL API

Returns the number of characters of the specified string expression.

Syntax

```
LENGTH(<str_expr>)
```

Arguments

str_expr
Is the string expression to be evaluated.

Return types

Returns a numeric expression.

Examples

The following example returns the length of a string.

```
SELECT LENGTH("abc") AS len
```

Here is the result set.

```
[{"len": 3}]
```

Remarks

This system function will not utilize the index.

Next steps

- String functions Azure Cosmos DB
- System functions Azure Cosmos DB
- Introduction to Azure Cosmos DB

LOWER (Azure Cosmos DB)

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: SQL API

Returns a string expression after converting uppercase character data to lowercase.

The LOWER system function does not utilize the index. If you plan to do frequent case insensitive comparisons, the LOWER system function may consume a significant amount of RU's. If this is the case, instead of using the LOWER system function to normalize data each time for comparisons, you can normalize the casing upon insertion. Then a query such as SELECT * FROM c WHERE LOWER(c.name) = 'bob' simply becomes SELECT * FROM c WHERE c.name = 'bob'.

Syntax

```
LOWER(<str_expr>)
```

Arguments

str_expr
Is a string expression.

Return types

Returns a string expression.

Examples

The following example shows how to use `LOWER` in a query.

```
SELECT LOWER("Abc") AS lower
```

Here is the result set.

```
[{"lower": "abc"}]
```

Remarks

This system function will not utilize the index.

Next steps

- String functions Azure Cosmos DB
- System functions Azure Cosmos DB
- Introduction to Azure Cosmos DB

LTRIM (Azure Cosmos DB)

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: SQL API

Returns a string expression after it removes leading blanks.

Syntax

```
LTRIM(<str_expr>)
```

Arguments

str_expr
Is a string expression.

Return types

Returns a string expression.

Examples

The following example shows how to use `LTRIM` inside a query.

```
SELECT LTRIM("  abc") AS l1, LTRIM("abc") AS l2, LTRIM("abc   ") AS l3
```

Here is the result set.

```
[{"l1": "abc", "l2": "abc", "l3": "abc   "}]
```

Remarks

This system function will not utilize the index.

Next steps

- String functions Azure Cosmos DB
- System functions Azure Cosmos DB
- Introduction to Azure Cosmos DB

REGEXMATCH (Azure Cosmos DB)

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: SQL API

Provides regular expression capabilities. Regular expressions are a concise and flexible notation for finding patterns of text. Azure Cosmos DB uses PERL compatible regular expressions (PCRE).

Syntax

```
RegexMatch(<str_expr1>, <str_expr2>, [, <str_expr3>])
```

Arguments

str_expr1
Is the string expression to be searched.

str_expr2
Is the regular expression.

str_expr3
Is the string of selected modifiers to use with the regular expression. This string value is optional. If you'd like to run RegexMatch with no modifiers, you can either add an empty string or omit entirely.

You can learn about syntax for creating regular expressions in Perl.

Azure Cosmos DB supports the following four modifiers:

MODIFIER	DESCRIPTION
`m`	Treat the string expression to be searched as multiple lines. Without this option, "^" and "$" will match at the beginning or end of the string and not each individual line.
`s`	Allow "." to match any character, including a newline character.
`i`	Ignore case when pattern matching.
`x`	Ignore all whitespace characters.

Return types

Returns a Boolean expression. Returns undefined if the string expression to be searched, the regular expression, or the selected modifiers are invalid.

Examples

The following simple RegexMatch example checks the string "abcd" for regular expression match using a few different modifiers.

```
SELECT RegexMatch ("abcd", "ABC", "") AS NoModifiers,
RegexMatch ("abcd", "ABC", "i") AS CaseInsensitive,
RegexMatch ("abcd", "ab.", "") AS WildcardCharacter,
RegexMatch ("abcd", "ab c", "x") AS IgnoreWhiteSpace,
RegexMatch ("abcd", "aB c", "ix") AS CaseInsensitiveAndIgnoreWhiteSpace
```

Here is the result set.

```
[
    {
        "NoModifiers": false,
        "CaseInsensitive": true,
        "WildcardCharacter": true,
        "IgnoreWhiteSpace": true,
        "CaseInsensitiveAndIgnoreWhiteSpace": true
    }
]
```

With RegexMatch, you can use metacharacters to do more complex string searches that wouldn't otherwise be possible with the StartsWith, EndsWith, Contains, or StringEquals system functions. Here are some additional examples, which you can run using the nutrition data set available through the Azure Cosmos DB Query Playground.

> **NOTE**
> If you need to use a metacharacter in a regular expression and don't want it to have special meaning, you should escape the metacharacter using `\`.

Check items that have a description that contains the word "salt" exactly once:

```
SELECT *
FROM c
WHERE RegexMatch (c.description, "salt{1}","")
```

Check items that have a description that contain a number between 0 and 99:

```
SELECT *
FROM c
WHERE RegexMatch (c.description, "[0-99]","")
```

Check items that have a description that contain four letter words starting with "S" or "s":

```
SELECT *
FROM c
WHERE RegexMatch (c.description, " s... ","i")
```

Remarks

This system function will benefit from a range index if the regular expression can be broken down into either StartsWith, EndsWith, Contains, or StringEquals system functions.

Next steps

- String functions Azure Cosmos DB

- System functions Azure Cosmos DB
- Introduction to Azure Cosmos DB

REPLACE (Azure Cosmos DB)

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: SQL API

Replaces all occurrences of a specified string value with another string value.

Syntax

```
REPLACE(<str_expr1>, <str_expr2>, <str_expr3>)
```

Arguments

str_expr1
Is the string expression to be searched.

str_expr2
Is the string expression to be found.

str_expr3
Is the string expression to replace occurrences of *str_expr2* in *str_expr1*.

Return types

Returns a string expression.

Examples

The following example shows how to use `REPLACE` in a query.

```
SELECT REPLACE("This is a Test", "Test", "desk") AS replace
```

Here is the result set.

```
[{"replace": "This is a desk"}]
```

Remarks

This system function will not utilize the index.

Next steps

- String functions Azure Cosmos DB
- System functions Azure Cosmos DB
- Introduction to Azure Cosmos DB

REPLICATE (Azure Cosmos DB)

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: SQL API

Repeats a string value a specified number of times.

Syntax

```
REPLICATE(<str_expr>, <num_expr>)
```

Arguments

str_expr
Is a string expression.

num_expr
Is a numeric expression. If *num_expr* is negative or non-finite, the result is undefined.

Return types

Returns a string expression.

Remarks

The maximum length of the result is 10,000 characters i.e. (length(*str_expr*) * *num_expr*) <= 10,000. This system function will not utilize the index.

Examples

The following example shows how to use `REPLICATE` in a query.

```
SELECT REPLICATE("a", 3) AS replicate
```

Here is the result set.

```
[{"replicate": "aaa"}]
```

Next steps

- String functions Azure Cosmos DB
- System functions Azure Cosmos DB
- Introduction to Azure Cosmos DB

REVERSE (Azure Cosmos DB)

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: SQL API

Returns the reverse order of a string value.

Syntax

```
REVERSE(<str_expr>)
```

Arguments

str_expr
Is a string expression.

Return types

Returns a string expression.

Examples

The following example shows how to use `REVERSE` in a query.

```
SELECT REVERSE("Abc") AS reverse
```

Here is the result set.

```
[{"reverse": "cbA"}]
```

Remarks

This system function will not utilize the index.

Next steps

- String functions Azure Cosmos DB
- System functions Azure Cosmos DB
- Introduction to Azure Cosmos DB

RIGHT (Azure Cosmos DB)

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: SQL API

Returns the right part of a string with the specified number of characters.

Syntax

```
RIGHT(<str_expr>, <num_expr>)
```

Arguments

str_expr
Is the string expression to extract characters from.

num_expr
Is a numeric expression which specifies the number of characters.

Return types

Returns a string expression.

Examples

The following example returns the right part of "abc" for various length values.

```
SELECT RIGHT("abc", 1) AS r1, RIGHT("abc", 2) AS r2
```

Here is the result set.

```
[{"r1": "c", "r2": "bc"}]
```

Remarks

This system function will not utilize the index.

Next steps

- String functions Azure Cosmos DB
- System functions Azure Cosmos DB
- Introduction to Azure Cosmos DB

RTRIM (Azure Cosmos DB)

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: SQL API

Returns a string expression after it removes trailing blanks.

Syntax

```
RTRIM(<str_expr>)
```

Arguments

str_expr
Is any valid string expression.

Return types

Returns a string expression.

Examples

The following example shows how to use `RTRIM` inside a query.

```
SELECT RTRIM("  abc") AS r1, RTRIM("abc") AS r2, RTRIM("abc   ") AS r3
```

Here is the result set.

```
[{"r1": "   abc", "r2": "abc", "r3": "abc"}]
```

Remarks

This system function will not utilize the index.

Next steps

- String functions Azure Cosmos DB
- System functions Azure Cosmos DB
- Introduction to Azure Cosmos DB

STARTSWITH (Azure Cosmos DB)

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: SQL API

Returns a Boolean indicating whether the first string expression starts with the second.

Syntax

```
STARTSWITH(<str_expr1>, <str_expr2> [, <bool_expr>])
```

Arguments

str_expr1
Is a string expression.

str_expr2
Is a string expression to be compared to the beginning of *str_expr1*.

bool_expr Optional value for ignoring case. When set to true, STARTSWITH will do a case-insensitive search. When unspecified, this value is false.

Return types

Returns a Boolean expression.

Examples

The following example checks if the string "abc" begins with "b" and "A".

```
SELECT STARTSWITH("abc", "b", false) AS s1, STARTSWITH("abc", "A", false) AS s2, STARTSWITH("abc", "A", true)
AS s3
```

Here is the result set.

```
[
    {
        "s1": false,
        "s2": false,
        "s3": true
    }
]
```

Remarks

This system function will benefit from a range index.

Next steps

- String functions Azure Cosmos DB

- System functions Azure Cosmos DB
- Introduction to Azure Cosmos DB

STRINGEQUALS (Azure Cosmos DB)

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: SQL API

Returns a Boolean indicating whether the first string expression matches the second.

Syntax

```
STRINGEQUALS(<str_expr1>, <str_expr2> [, <bool_expr>])
```

Arguments

str_expr1
Is the first string expression to compare.

str_expr2
Is the second string expression to compare.

bool_expr Optional value for ignoring case. When set to true, StringEquals will do a case-insensitive search. When unspecified, this value is false.

Return types

Returns a Boolean expression.

Examples

The following example checks if "abc" matches "abc" and if "abc" matches "ABC".

```
SELECT STRINGEQUALS("abc", "abc", false) AS c1, STRINGEQUALS("abc", "ABC", false) AS c2,  STRINGEQUALS("abc",
"ABC", true) AS c3
```

Here is the result set.

```
[
    {
        "c1": true,
        "c2": false,
        "c3": true
    }
]
```

Remarks

This system function will benefit from a range index.

Next steps

- String functions Azure Cosmos DB

- System functions Azure Cosmos DB
- Introduction to Azure Cosmos DB

StringToArray (Azure Cosmos DB)

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: SQL API

Returns expression translated to an Array. If expression cannot be translated, returns undefined.

Syntax

```
StringToArray(<str_expr>)
```

Arguments

str_expr
Is a string expression to be parsed as a JSON Array expression.

Return types

Returns an array expression or undefined.

Remarks

Nested string values must be written with double quotes to be valid JSON. For details on the JSON format, see json.org. This system function will not utilize the index.

Examples

The following example shows how `StringToArray` behaves across different types.

The following are examples with valid input.

```
SELECT
    StringToArray('[]') AS a1,
    StringToArray("[1,2,3]") AS a2,
    StringToArray("[\"str\",2,3]") AS a3,
    StringToArray('[["5","6","7"],["8"],["9"]]') AS a4,
    StringToArray('[1,2,3, "[4,5,6]",[7,8]]') AS a5
```

Here is the result set.

```
[{"a1": [], "a2": [1,2,3], "a3": ["str",2,3], "a4": [["5","6","7"],["8"],["9"]], "a5": [1,2,3,"[4,5,6]",
[7,8]]}]
```

The following is an example of invalid input.

Single quotes within the array are not valid JSON. Even though they are valid within a query, they will not parse to valid arrays. Strings within the array string must either be escaped "[\"\"]" or the surrounding quote must be single '[""]'.

```
SELECT
    StringToArray("['5','6','7']")
```

Here is the result set.

```
[{}]
```

The following are examples of invalid input.

The expression passed will be parsed as a JSON array; the following do not evaluate to type array and thus return undefined.

```
SELECT
    StringToArray("["),
    StringToArray("1"),
    StringToArray(NaN),
    StringToArray(false),
    StringToArray(undefined)
```

Here is the result set.

```
[{}]
```

Next steps

- String functions Azure Cosmos DB
- System functions Azure Cosmos DB
- Introduction to Azure Cosmos DB

StringToBoolean (Azure Cosmos DB)

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: SQL API

Returns expression translated to a Boolean. If expression cannot be translated, returns undefined.

Syntax

```
StringToBoolean(<str_expr>)
```

Arguments

str_expr
Is a string expression to be parsed as a Boolean expression.

Return types

Returns a Boolean expression or undefined.

Examples

The following example shows how `StringToBoolean` behaves across different types.

The following are examples with valid input.

Whitespace is allowed only before or after "true"/"false".

```
SELECT
    StringToBoolean("true") AS b1,
    StringToBoolean("    false") AS b2,
    StringToBoolean("false    ") AS b3
```

Here is the result set.

```
[{"b1": true, "b2": false, "b3": false}]
```

The following are examples with invalid input.

Booleans are case sensitive and must be written with all lowercase characters i.e. "true" and "false".

```
SELECT
    StringToBoolean("TRUE"),
    StringToBoolean("False")
```

Here is the result set.

```
[{}]
```

The expression passed will be parsed as a Boolean expression; these inputs do not evaluate to type Boolean and thus return undefined.

```
SELECT
    StringToBoolean("null"),
    StringToBoolean(undefined),
    StringToBoolean(NaN),
    StringToBoolean(false),
    StringToBoolean(true)
```

Here is the result set.

```
[{}]
```

Remarks

This system function will not utilize the index.

Next steps

- String functions Azure Cosmos DB
- System functions Azure Cosmos DB
- Introduction to Azure Cosmos DB

StringToNull (Azure Cosmos DB)

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: SQL API

Returns expression translated to null. If expression cannot be translated, returns undefined.

Syntax

```
StringToNull(<str_expr>)
```

Arguments

str_expr
Is a string expression to be parsed as a null expression.

Return types

Returns a null expression or undefined.

Examples

The following example shows how `StringToNull` behaves across different types.

The following are examples with valid input.

Whitespace is allowed only before or after "null".

```
SELECT
    StringToNull("null") AS n1,
    StringToNull("  null ") AS n2,
    IS_NULL(StringToNull("null   ")) AS n3
```

Here is the result set.

```
[{"n1": null, "n2": null, "n3": true}]
```

The following are examples with invalid input.

Null is case sensitive and must be written with all lowercase characters i.e. "null".

```
SELECT
    StringToNull("NULL"),
    StringToNull("Null")
```

Here is the result set.

```
[{}]
```

The expression passed will be parsed as a null expression; these inputs do not evaluate to type null and thus return undefined.

```
SELECT
    StringToNull("true"),
    StringToNull(false),
    StringToNull(undefined),
    StringToNull(NaN)
```

Here is the result set.

```
[{}]
```

Remarks

This system function will not utilize the index.

Next steps

- String functions Azure Cosmos DB
- System functions Azure Cosmos DB
- Introduction to Azure Cosmos DB

StringToNumber (Azure Cosmos DB)

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: SQL API

Returns expression translated to a Number. If expression cannot be translated, returns undefined.

Syntax

```
StringToNumber(<str_expr>)
```

Arguments

str_expr
Is a string expression to be parsed as a JSON Number expression. Numbers in JSON must be an integer or a floating point. For details on the JSON format, see json.org

Return types

Returns a Number expression or undefined.

Examples

The following example shows how `StringToNumber` behaves across different types.

Whitespace is allowed only before or after the Number.

```
SELECT
    StringToNumber("1.000000") AS num1,
    StringToNumber("3.14") AS num2,
    StringToNumber("   60   ") AS num3,
    StringToNumber("-1.79769e+308") AS num4
```

Here is the result set.

```
{{"num1": 1, "num2": 3.14, "num3": 60, "num4": -1.79769e+308}}
```

In JSON a valid Number must be either be an integer or a floating point number.

```
SELECT
    StringToNumber("0xF")
```

Here is the result set.

```
{{}}
```

The expression passed will be parsed as a Number expression; these inputs do not evaluate to type Number and thus return undefined.

```
SELECT
    StringToNumber("99     54"),
    StringToNumber(undefined),
    StringToNumber("false"),
    StringToNumber(false),
    StringToNumber(" "),
    StringToNumber(NaN)
```

Here is the result set.

```
{{}}
```

Remarks

This system function will not utilize the index.

Next steps

- String functions Azure Cosmos DB
- System functions Azure Cosmos DB
- Introduction to Azure Cosmos DB

StringToObject (Azure Cosmos DB)

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: SQL API

Returns expression translated to an Object. If expression cannot be translated, returns undefined.

Syntax

```
StringToObject(<str_expr>)
```

Arguments

str_expr
Is a string expression to be parsed as a JSON object expression. Note that nested string values must be written with double quotes to be valid. For details on the JSON format, see json.org

Return types

Returns an object expression or undefined.

Examples

The following example shows how `StringToObject` behaves across different types.

The following are examples with valid input.

```
SELECT
    StringToObject("{}") AS obj1,
    StringToObject('{"A":[1,2,3]}') AS obj2,
    StringToObject('{"B":[{"b1":[5,6,7]},{"b2":8},{"b3":9}]}') AS obj3,
    StringToObject("{\"C\":[{\"c1\":[5,6,7]},{\"c2\":8},{\"c3\":9}]}") AS obj4
```

Here is the result set.

```
[{"obj1": {},
  "obj2": {"A": [1,2,3]},
  "obj3": {"B":[{"b1":[5,6,7]},{"b2":8},{"b3":9}]},
  "obj4": {"C":[{"c1":[5,6,7]},{"c2":8},{"c3":9}]}}]
```

The following are examples with invalid input. Even though they are valid within a query, they will not parse to valid objects. Strings within the string of object must either be escaped "{\"a\":\"str\"}" or the surrounding quote must be single '{"a": "str"}'.

Single quotes surrounding property names are not valid JSON.

```
SELECT
    StringToObject("{'a':[1,2,3]}")
```

Here is the result set.

```
[{}]
```

Property names without surrounding quotes are not valid JSON.

```
SELECT
    StringToObject("{a:[1,2,3]}")
```

Here is the result set.

```
[{}]
```

The following are examples with invalid input.

The expression passed will be parsed as a JSON object; these inputs do not evaluate to type object and thus return undefined.

```
SELECT
    StringToObject("}"),
    StringToObject("{"),
    StringToObject("1"),
    StringToObject(NaN),
    StringToObject(false),
    StringToObject(undefined)
```

Here is the result set.

```
[{}]
```

Remarks

This system function will not utilize the index.

Next steps

- String functions Azure Cosmos DB
- System functions Azure Cosmos DB
- Introduction to Azure Cosmos DB

SUBSTRING (Azure Cosmos DB)

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: SQL API

Returns part of a string expression starting at the specified character zero-based position and continues to the specified length, or to the end of the string.

Syntax

```
SUBSTRING(<str_expr>, <num_expr1>, <num_expr2>)
```

Arguments

str_expr
Is a string expression.

num_expr1
Is a numeric expression to denote the start character. A value of 0 is the first character of *str_expr*.

num_expr2
Is a numeric expression to denote the maximum number of characters of *str_expr* to be returned. A value of 0 or less results in empty string.

Return types

Returns a string expression.

Examples

The following example returns the substring of "abc" starting at 1 and for a length of 1 character.

```
SELECT SUBSTRING("abc", 1, 1) AS substring
```

Here is the result set.

```
[{"substring": "b"}]
```

Remarks

This system function will benefit from a range index if the starting position is `0`.

Next steps

- String functions Azure Cosmos DB
- System functions Azure Cosmos DB
- Introduction to Azure Cosmos DB

ToString (Azure Cosmos DB)

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: SQL API

Returns a string representation of scalar expression.

Syntax

```
ToString(<expr>)
```

Arguments

expr
Is any scalar expression.

Return types

Returns a string expression.

Examples

The following example shows how `ToString` behaves across different types.

```
SELECT
    ToString(1.0000) AS str1,
    ToString("Hello World") AS str2,
    ToString(NaN) AS str3,
    ToString(Infinity) AS str4,
    ToString(IS_STRING(ToString(undefined))) AS str5,
    ToString(0.1234) AS str6,
    ToString(false) AS str7,
    ToString(undefined) AS str8
```

Here is the result set.

```
[{"str1": "1", "str2": "Hello World", "str3": "NaN", "str4": "Infinity", "str5": "false", "str6": "0.1234",
"str7": "false"}]
```

Given the following input:

```
{"Products":[{"ProductID":1,"Weight":4,"WeightUnits":"lb"},{"ProductID":2,"Weight":32,"WeightUnits":"kg"},
{"ProductID":3,"Weight":400,"WeightUnits":"g"},{"ProductID":4,"Weight":8999,"WeightUnits":"mg"}]}
```

The following example shows how `ToString` can be used with other string functions like `CONCAT`.

```
SELECT
CONCAT(ToString(p.Weight), p.WeightUnits)
FROM p in c.Products
```

Here is the result set.

```
[{"$1":"4lb" },
{"$1":"32kg"},
{"$1":"400g" },
{"$1":"8999mg" }]
```

Given the following input.

```
{"id":"08259","description":"Cereals ready-to-eat, KELLOGG, KELLOGG'S CRISPIX","nutrients":
[{"id":"305","description":"Caffeine","units":"mg"},{"id":"306","description":"Cholesterol,
HDL","nutritionValue":30,"units":"mg"},{"id":"307","description":"Sodium,
NA","nutritionValue":612,"units":"mg"},{"id":"308","description":"Protein,
ABP","nutritionValue":60,"units":"mg"},{"id":"309","description":"Zinc,
ZN","nutritionValue":null,"units":"mg"}]}
```

The following example shows how `ToString` can be used with other string functions like `REPLACE`.

```
SELECT
    n.id AS nutrientID,
    REPLACE(ToString(n.nutritionValue), "6", "9") AS nutritionVal
FROM food
JOIN n IN food.nutrients
```

Here is the result set.

```
[{"nutrientID":"305"},
{"nutrientID":"306","nutritionVal":"30"},
{"nutrientID":"307","nutritionVal":"912"},
{"nutrientID":"308","nutritionVal":"90"},
{"nutrientID":"309","nutritionVal":"null"}]
```

Remarks

This system function will not utilize the index.

Next steps

- String functions Azure Cosmos DB
- System functions Azure Cosmos DB
- Introduction to Azure Cosmos DB

TRIM (Azure Cosmos DB)

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: SQL API

Returns a string expression after it removes leading and trailing blanks.

Syntax

```
TRIM(<str_expr>)
```

Arguments

str_expr
Is a string expression.

Return types

Returns a string expression.

Examples

The following example shows how to use `TRIM` inside a query.

```
SELECT TRIM("   abc") AS t1, TRIM("   abc   ") AS t2, TRIM("abc   ") AS t3, TRIM("abc") AS t4
```

Here is the result set.

```
[{"t1": "abc", "t2": "abc", "t3": "abc", "t4": "abc"}]
```

Remarks

This system function will not utilize the index.

Next steps

- String functions Azure Cosmos DB
- System functions Azure Cosmos DB
- Introduction to Azure Cosmos DB

UPPER (Azure Cosmos DB)

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: SQL API

Returns a string expression after converting lowercase character data to uppercase.

The UPPER system function does not utilize the index. If you plan to do frequent case insensitive comparisons, the UPPER system function may consume a significant amount of RU's. If this is the case, instead of using the UPPER system function to normalize data each time for comparisons, you can normalize the casing upon insertion. Then a query such as SELECT * FROM c WHERE UPPER(c.name) = 'BOB' simply becomes SELECT * FROM c WHERE c.name = 'BOB'.

Syntax

```
UPPER(<str_expr>)
```

Arguments

str_expr
Is a string expression.

Return types

Returns a string expression.

Examples

The following example shows how to use `UPPER` in a query

```
SELECT UPPER("Abc") AS upper
```

Here is the result set.

```
[{"upper": "ABC"}]
```

Remarks

This system function will not utilize the index.

Next steps

- String functions Azure Cosmos DB
- System functions Azure Cosmos DB
- Introduction to Azure Cosmos DB

Type checking functions (Azure Cosmos DB)

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: SQL API

The type-checking functions let you check the type of an expression within a SQL query. You can use type-checking functions to determine the types of properties within items on the fly, when they're variable or unknown.

Functions

Here's a table of supported built-in type-checking functions:

The following functions support type checking against input values, and each return a Boolean value.

- IS_ARRAY
- IS_BOOL
- IS_DEFINED
- IS_NULL
- IS_NUMBER
- IS_OBJECT
- IS_PRIMITIVE
- IS_STRING

Next steps

- System functions Azure Cosmos DB
- Introduction to Azure Cosmos DB
- User Defined Functions
- Aggregates

IS_ARRAY (Azure Cosmos DB)

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: SQL API

Returns a Boolean value indicating if the type of the specified expression is an array.

Syntax

```
IS_ARRAY(<expr>)
```

Arguments

expr
Is any expression.

Return types

Returns a Boolean expression.

Examples

The following example checks objects of JSON Boolean, number, string, null, object, array, and undefined types using the `IS_ARRAY` function.

```
SELECT
 IS_ARRAY(true) AS isArray1,
 IS_ARRAY(1) AS isArray2,
 IS_ARRAY("value") AS isArray3,
 IS_ARRAY(null) AS isArray4,
 IS_ARRAY({prop: "value"}) AS isArray5,
 IS_ARRAY([1, 2, 3]) AS isArray6,
 IS_ARRAY({prop: "value"}.prop2) AS isArray7
```

Here is the result set.

```
[{"isArray1":false,"isArray2":false,"isArray3":false,"isArray4":false,"isArray5":false,"isArray6":true,"isArra
y7":false}]
```

Remarks

This system function will benefit from a range index.

Next steps

- Type checking functions Azure Cosmos DB
- System functions Azure Cosmos DB
- Introduction to Azure Cosmos DB

IS_BOOL (Azure Cosmos DB)

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: SQL API

Returns a Boolean value indicating if the type of the specified expression is a Boolean.

Syntax

```
IS_BOOL(<expr>)
```

Arguments

expr
Is any expression.

Return types

Returns a Boolean expression.

Examples

The following example checks objects of JSON Boolean, number, string, null, object, array, and undefined types using the `IS_BOOL` function.

```
SELECT
    IS_BOOL(true) AS isBool1,
    IS_BOOL(1) AS isBool2,
    IS_BOOL("value") AS isBool3,
    IS_BOOL(null) AS isBool4,
    IS_BOOL({prop: "value"}) AS isBool5,
    IS_BOOL([1, 2, 3]) AS isBool6,
    IS_BOOL({prop: "value"}.prop2) AS isBool7
```

Here is the result set.

```
[{"isBool1":true,"isBool2":false,"isBool3":false,"isBool4":false,"isBool5":false,"isBool6":false,"isBool7":fal
se}]
```

Remarks

This system function will benefit from a range index.

Next steps

- Type checking functions Azure Cosmos DB
- System functions Azure Cosmos DB
- Introduction to Azure Cosmos DB

IS_DEFINED (Azure Cosmos DB)

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: SQL API

Returns a Boolean indicating if the property has been assigned a value.

Syntax

```
IS_DEFINED(<expr>)
```

Arguments

expr
Is any expression.

Return types

Returns a Boolean expression.

Examples

The following example checks for the presence of a property within the specified JSON document. The first returns true since "a" is present, but the second returns false since "b" is absent.

```
SELECT IS_DEFINED({ "a" : 5 }.a) AS isDefined1, IS_DEFINED({ "a" : 5 }.b) AS isDefined2
```

Here is the result set.

```
[{"isDefined1":true,"isDefined2":false}]
```

Remarks

This system function will benefit from a range index.

Next steps

- Type checking functions Azure Cosmos DB
- System functions Azure Cosmos DB
- Introduction to Azure Cosmos DB

IS_NULL (Azure Cosmos DB)

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: SQL API

Returns a Boolean value indicating if the type of the specified expression is null.

Syntax

```
IS_NULL(<expr>)
```

Arguments

expr
Is any expression.

Return types

Returns a Boolean expression.

Examples

The following example checks objects of JSON Boolean, number, string, null, object, array, and undefined types using the `IS_NULL` function.

```
SELECT
    IS_NULL(true) AS isNull1,
    IS_NULL(1) AS isNull2,
    IS_NULL("value") AS isNull3,
    IS_NULL(null) AS isNull4,
    IS_NULL({prop: "value"}) AS isNull5,
    IS_NULL([1, 2, 3]) AS isNull6,
    IS_NULL({prop: "value"}.prop2) AS isNull7
```

Here is the result set.

```
[{"isNull1":false,"isNull2":false,"isNull3":false,"isNull4":true,"isNull5":false,"isNull6":false,"isNull7":fal
se}]
```

Remarks

This system function will benefit from a range index.

Next steps

- Type checking functions Azure Cosmos DB
- System functions Azure Cosmos DB
- Introduction to Azure Cosmos DB

IS_NUMBER (Azure Cosmos DB)

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: SQL API

Returns a Boolean value indicating if the type of the specified expression is a number.

Syntax

```
IS_NUMBER(<expr>)
```

Arguments

expr
Is any expression.

Return types

Returns a Boolean expression.

Examples

The following example checks objects of JSON Boolean, number, string, null, object, array, and undefined types using the `IS_NUMBER` function.

```
SELECT
    IS_NUMBER(true) AS isNum1,
    IS_NUMBER(1) AS isNum2,
    IS_NUMBER("value") AS isNum3,
    IS_NUMBER(null) AS isNum4,
    IS_NUMBER({prop: "value"}) AS isNum5,
    IS_NUMBER([1, 2, 3]) AS isNum6,
    IS_NUMBER({prop: "value"}.prop2) AS isNum7
```

Here is the result set.

```
[{"isNum1":false,"isNum2":true,"isNum3":false,"isNum4":false,"isNum5":false,"isNum6":false,"isNum7":false}]
```

Remarks

This system function will benefit from a range index.

Next steps

- Type checking functions Azure Cosmos DB
- System functions Azure Cosmos DB
- Introduction to Azure Cosmos DB

IS_OBJECT (Azure Cosmos DB)

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: SQL API

Returns a Boolean value indicating if the type of the specified expression is a JSON object.

Syntax

```
IS_OBJECT(<expr>)
```

Arguments

expr
Is any expression.

Return types

Returns a Boolean expression.

Examples

The following example checks objects of JSON Boolean, number, string, null, object, array, and undefined types using the `IS_OBJECT` function.

```
SELECT
    IS_OBJECT(true) AS isObj1,
    IS_OBJECT(1) AS isObj2,
    IS_OBJECT("value") AS isObj3,
    IS_OBJECT(null) AS isObj4,
    IS_OBJECT({prop: "value"}) AS isObj5,
    IS_OBJECT([1, 2, 3]) AS isObj6,
    IS_OBJECT({prop: "value"}.prop2) AS isObj7
```

Here is the result set.

```
[{"isObj1":false,"isObj2":false,"isObj3":false,"isObj4":false,"isObj5":true,"isObj6":false,"isObj7":false}]
```

Remarks

This system function will benefit from a range index.

Next steps

- Type checking functions Azure Cosmos DB
- System functions Azure Cosmos DB
- Introduction to Azure Cosmos DB

IS_PRIMITIVE (Azure Cosmos DB)

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: SQL API

Returns a Boolean value indicating if the type of the specified expression is a primitive (string, Boolean, numeric, or null).

Syntax

```
IS_PRIMITIVE(<expr>)
```

Arguments

expr
Is any expression.

Return types

Returns a Boolean expression.

Examples

The following example checks objects of JSON Boolean, number, string, null, object, array and undefined types using the `IS_PRIMITIVE` function.

```
SELECT
           IS_PRIMITIVE(true) AS isPrim1,
           IS_PRIMITIVE(1) AS isPrim2,
           IS_PRIMITIVE("value") AS isPrim3,
           IS_PRIMITIVE(null) AS isPrim4,
           IS_PRIMITIVE({prop: "value"}) AS isPrim5,
           IS_PRIMITIVE([1, 2, 3]) AS isPrim6,
           IS_PRIMITIVE({prop: "value"}.prop2) AS isPrim7
```

Here is the result set.

```
[{"isPrim1": true, "isPrim2": true, "isPrim3": true, "isPrim4": true, "isPrim5": false, "isPrim6": false,
"isPrim7": false}]
```

Remarks

This system function will benefit from a range index.

Next steps

- Type checking functions Azure Cosmos DB
- System functions Azure Cosmos DB
- Introduction to Azure Cosmos DB

IS_STRING (Azure Cosmos DB)

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: SQL API

Returns a Boolean value indicating if the type of the specified expression is a string.

Syntax

```
IS_STRING(<expr>)
```

Arguments

expr
Is any expression.

Return types

Returns a Boolean expression.

Examples

The following example checks objects of JSON Boolean, number, string, null, object, array, and undefined types using the `IS_STRING` function.

```
SELECT
       IS_STRING(true) AS isStr1,
       IS_STRING(1) AS isStr2,
       IS_STRING("value") AS isStr3,
       IS_STRING(null) AS isStr4,
       IS_STRING({prop: "value"}) AS isStr5,
       IS_STRING([1, 2, 3]) AS isStr6,
       IS_STRING({prop: "value"}.prop2) AS isStr7
```

Here is the result set.

```
[{"isStr1":false,"isStr2":false,"isStr3":true,"isStr4":false,"isStr5":false,"isStr6":false,"isStr7":false}]
```

Remarks

This system function will benefit from a range index.

Next steps

- Type checking functions Azure Cosmos DB
- System functions Azure Cosmos DB
- Introduction to Azure Cosmos DB

Geospatial and GeoJSON location data in Azure Cosmos DB

12/4/2020 • 5 minutes to read • Edit Online

APPLIES TO: SQL API

This article is an introduction to the geospatial functionality in Azure Cosmos DB. Currently storing and accessing geospatial data is supported by Azure Cosmos DB SQL API accounts only. After reading our documentation on geospatial indexing you will be able to answer the following questions:

- How do I store spatial data in Azure Cosmos DB?
- How can I query geospatial data in Azure Cosmos DB in SQL and LINQ?
- How do I enable or disable spatial indexing in Azure Cosmos DB?

Spatial Data Use Cases

Geospatial data often involve proximity queries, for example, "find all coffee shops near my current location". Common use cases are:

- Geolocation Analytics, driving specific located marketing initiatives.
- Location based personalization, for multiple industries like Retail and Healthcare.
- Logistics enhancement, for transport optimization.
- Risk Analysis, especially for insurance and finance companies.
- Situational awareness, for alerts and notifications.

Introduction to spatial data

Spatial data describes the position and shape of objects in space. In most applications, these correspond to objects on the earth and geospatial data. Spatial data can be used to represent the location of a person, a place of interest, or the boundary of a city, or a lake.

Azure Cosmos DB's SQL API supports two spatial data types: the **geometry** data type and the **geography** data type.

- The **geometry** type represents data in a Euclidean (flat) coordinate system
- The **geography** type represents data in a round-earth coordinate system.

Supported data types

Azure Cosmos DB supports indexing and querying of geospatial point data that's represented using the GeoJSON specification. GeoJSON data structures are always valid JSON objects, so they can be stored and queried using Azure Cosmos DB without any specialized tools or libraries.

Azure Cosmos DB supports the following spatial data types:

- Point
- LineString
- Polygon
- MultiPolygon

Points

A **Point** denotes a single position in space. In geospatial data, a Point represents the exact location, which could be a street address of a grocery store, a kiosk, an automobile, or a city. A point is represented in GeoJSON (and Azure Cosmos DB) using its coordinate pair or longitude and latitude.

Here's an example JSON for a point:

Points in Azure Cosmos DB

```
{
    "type":"Point",
    "coordinates":[ 31.9, -4.8 ]
}
```

Spatial data types can be embedded in an Azure Cosmos DB document as shown in this example of a user profile containing location data:

Use Profile with Location stored in Azure Cosmos DB

```
{
    "id":"cosmosdb-profile",
    "screen_name":"@CosmosDB",
    "city":"Redmond",
    "topics":[ "global", "distributed" ],
    "location":{
        "type":"Point",
        "coordinates":[ 31.9, -4.8 ]
    }
}
```

Points in a geometry coordinate system

For the **geometry** data type, GeoJSON specification specifies the horizontal axis first and the vertical axis second.

Points in a geography coordinate system

For the **geography** data type, GeoJSON specification specifies longitude first and latitude second. Like in other mapping applications, longitude and latitude are angles and represented in terms of degrees. Longitude values are measured from the Prime Meridian and are between -180 degrees and 180.0 degrees, and latitude values are measured from the equator and are between -90.0 degrees and 90.0 degrees.

Azure Cosmos DB interprets coordinates as represented per the WGS-84 reference system. See below for more details about coordinate reference systems.

LineStrings

LineStrings represent a series of two or more points in space and the line segments that connect them. In geospatial data, LineStrings are commonly used to represent highways or rivers.

LineStrings in GeoJSON

```
    "type":"LineString",
    "coordinates":[ [
        [ 31.8, -5 ],
        [ 31.8, -4.7 ]
    ] ]
```

Polygons

A **Polygon** is a boundary of connected points that forms a closed LineString. Polygons are commonly used to represent natural formations like lakes or political jurisdictions like cities and states. Here's an example of a Polygon in Azure Cosmos DB:

Polygons in GeoJSON

```
{
    "type":"Polygon",
    "coordinates":[ [
        [ 31.8, -5 ],
        [ 31.8, -4.7 ],
        [ 32, -4.7 ],
        [ 32, -5 ],
        [ 31.8, -5 ]
    ] ]
}
```

> **NOTE**
>
> The GeoJSON specification requires that for valid Polygons, the last coordinate pair provided should be the same as the first, to create a closed shape.
>
> Points within a Polygon must be specified in counter-clockwise order. A Polygon specified in clockwise order represents the inverse of the region within it.

MultiPolygons

A **MultiPolygon** is an array of zero or more Polygons. **MultiPolygons** cannot overlap sides or have any common area. They may touch at one or more points.

MultiPolygons in GeoJSON

```
{
    "type":"MultiPolygon",
    "coordinates":[[[
        [52.0, 12.0],
        [53.0, 12.0],
        [53.0, 13.0],
        [52.0, 13.0],
        [52.0, 12.0]
        ]],
        [[
        [50.0, 0.0],
        [51.0, 0.0],
        [51.0, 5.0],
        [50.0, 5.0],
        [50.0, 0.0]
        ]]]
}
```

Coordinate reference systems

Since the shape of the earth is irregular, coordinates of geography geospatial data are represented in many coordinate reference systems (CRS), each with their own frames of reference and units of measurement. For example, the "National Grid of Britain" is a reference system is accurate for the United Kingdom, but not outside it.

The most popular CRS in use today is the World Geodetic System WGS-84. GPS devices, and many mapping services including Google Maps and Bing Maps APIs use WGS-84. Azure Cosmos DB supports indexing and querying of geography geospatial data using the WGS-84 CRS only.

Creating documents with spatial data

When you create documents that contain GeoJSON values, they are automatically indexed with a spatial index in

accordance to the indexing policy of the container. If you're working with an Azure Cosmos DB SDK in a dynamically typed language like Python or Node.js, you must create valid GeoJSON.

Create Document with Geospatial data in Node.js

```
var userProfileDocument = {
    "id":"cosmosdb",
    "location":{
        "type":"Point",
        "coordinates":[ -122.12, 47.66 ]
    }
};

client.createDocument(`dbs/${databaseName}/colls/${collectionName}`, userProfileDocument, (err, created) => {
    // additional code within the callback
});
```

If you're working with the SQL APIs, you can use the `Point`, `LineString`, `Polygon`, and `MultiPolygon` classes within the `Microsoft.Azure.Cosmos.Spatial` namespace to embed location information within your application objects. These classes help simplify the serialization and deserialization of spatial data into GeoJSON.

Create Document with Geospatial data in .NET

```
using Microsoft.Azure.Cosmos.Spatial;

public class UserProfile
{
    [JsonProperty("id")]
    public string id { get; set; }

    [JsonProperty("location")]
    public Point Location { get; set; }

    // More properties
}

await container.CreateItemAsync( new UserProfile
    {
        id = "cosmosdb",
        Location = new Point (-122.12, 47.66)
    });
```

If you don't have the latitude and longitude information, but have the physical addresses or location name like city or country/region, you can look up the actual coordinates by using a geocoding service like Bing Maps REST Services. Learn more about Bing Maps geocoding here.

Next steps

Now that you have learned how to get started with geospatial support in Azure Cosmos DB, next you can:

- Learn more about Azure Cosmos DB Query
- Learn more about Querying spatial data with Azure Cosmos DB
- Learn more about Index spatial data with Azure Cosmos DB

Querying geospatial data with Azure Cosmos DB

12/4/2020 • 4 minutes to read • Edit Online

APPLIES TO: SQL API

This article will cover how to query geospatial data in Azure Cosmos DB using SQL and LINQ. Currently storing and accessing geospatial data is supported by Azure Cosmos DB SQL API accounts only. Azure Cosmos DB supports the following Open Geospatial Consortium (OGC) built-in functions for geospatial querying. For more information on the complete set of built-in functions in the SQL language, see Query System Functions in Azure Cosmos DB.

Spatial SQL built-in functions

Here is a list of geospatial system functions useful for querying in Azure Cosmos DB:

USAGE	DESCRIPTION
ST_DISTANCE (spatial_expr, spatial_expr)	Returns the distance between the two GeoJSON Point, Polygon, or LineString expressions.
ST_WITHIN (spatial_expr, spatial_expr)	Returns a Boolean expression indicating whether the first GeoJSON object (Point, Polygon, or LineString) is within the second GeoJSON object (Point, Polygon, or LineString).
ST_INTERSECTS (spatial_expr, spatial_expr)	Returns a Boolean expression indicating whether the two specified GeoJSON objects (Point, Polygon, or LineString) intersect.
ST_ISVALID	Returns a Boolean value indicating whether the specified GeoJSON Point, Polygon, or LineString expression is valid.
ST_ISVALIDDETAILED	Returns a JSON value containing a Boolean value if the specified GeoJSON Point, Polygon, or LineString expression is valid. If invalid, it returns the reason as a string value.

Spatial functions can be used to perform proximity queries against spatial data. For example, here's a query that returns all family documents that are within 30 km of the specified location using the `ST_DISTANCE` built-in function.

Query

```
SELECT f.id
FROM Families f
WHERE ST_DISTANCE(f.location, {"type": "Point", "coordinates":[31.9, -4.8]}) < 30000
```

Results

```
[{
  "id": "WakefieldFamily"
}]
```

If you include spatial indexing in your indexing policy, then "distance queries" will be served efficiently through the

index. For more information on spatial indexing, see geospatial indexing. If you don't have a spatial index for the specified paths, the query will do a scan of the container.

`ST_WITHIN` can be used to check if a point lies within a Polygon. Commonly Polygons are used to represent boundaries like zip codes, state boundaries, or natural formations. Again if you include spatial indexing in your indexing policy, then "within" queries will be served efficiently through the index.

Polygon arguments in `ST_WITHIN` can contain only a single ring, that is, the Polygons must not contain holes in them.

Query

```
SELECT *
FROM Families f
WHERE ST_WITHIN(f.location, {
    "type":"Polygon",
    "coordinates": [[[31.8, -5], [32, -5], [32, -4.7], [31.8, -4.7], [31.8, -5]]]
})
```

Results

```
[{
  "id": "WakefieldFamily",
}]
```

> **NOTE**
>
> Similar to how mismatched types work in Azure Cosmos DB query, if the location value specified in either argument is malformed or invalid, then it evaluates to **undefined** and the evaluated document to be skipped from the query results. If your query returns no results, run `ST_ISVALIDDETAILED` to debug why the spatial type is invalid.

Azure Cosmos DB also supports performing inverse queries, that is, you can index polygons or lines in Azure Cosmos DB, then query for the areas that contain a specified point. This pattern is commonly used in logistics to identify, for example, when a truck enters or leaves a designated area.

Query

```
SELECT *
FROM Areas a
WHERE ST_WITHIN({"type": "Point", "coordinates":[31.9, -4.8]}, a.location)
```

Results

```
[{
  "id": "MyDesignatedLocation",
  "location": {
    "type":"Polygon",
    "coordinates": [[[31.8, -5], [32, -5], [32, -4.7], [31.8, -4.7], [31.8, -5]]]
  }
}]
```

`ST_ISVALID` and `ST_ISVALIDDETAILED` can be used to check if a spatial object is valid. For example, the following query checks the validity of a point with an out of range latitude value (-132.8). `ST_ISVALID` returns just a Boolean value, and `ST_ISVALIDDETAILED` returns the Boolean and a string containing the reason why it is considered invalid.

Query

```
SELECT ST_ISVALID({ "type": "Point", "coordinates": [31.9, -132.8] })
```

Results

```
[{
  "$1": false
}]
```

These functions can also be used to validate Polygons. For example, here we use `ST_ISVALIDDETAILED` to validate a Polygon that is not closed.

Query

```
SELECT ST_ISVALIDDETAILED({ "type": "Polygon", "coordinates": [[
    [ 31.8, -5 ], [ 31.8, -4.7 ], [ 32, -4.7 ], [ 32, -5 ]
    ]]})
```

Results

```
[{
   "$1": {
        "valid": false,
        "reason": "The Polygon input is not valid because the start and end points of the ring number 1
are not the same. Each ring of a Polygon must have the same start and end points."
      }
}]
```

LINQ querying in the .NET SDK

The SQL .NET SDK also providers stub methods `Distance()` and `Within()` for use within LINQ expressions. The SQL LINQ provider translates this method calls to the equivalent SQL built-in function calls (ST_DISTANCE and ST_WITHIN respectively).

Here's an example of a LINQ query that finds all documents in the Azure Cosmos container whose `location` value is within a radius of 30 km of the specified point using LINQ.

LINQ query for Distance

```
foreach (UserProfile user in container.GetItemLinqQueryable<UserProfile>(allowSynchronousQueryExecution:
true)
    .Where(u => u.ProfileType == "Public" && u.Location.Distance(new Point(32.33, -4.66)) < 30000))
{
    Console.WriteLine("\t" + user);
}
```

Similarly, here's a query for finding all the documents whose `location` is within the specified box/Polygon.

LINQ query for Within

```
    Polygon rectangularArea = new Polygon(
        new[]
        {
            new LinearRing(new [] {
                new Position(31.8, -5),
                new Position(32, -5),
                new Position(32, -4.7),
                new Position(31.8, -4.7),
                new Position(31.8, -5)
            })
        });

    foreach (UserProfile user in container.GetItemLinqQueryable<UserProfile>(allowSynchronousQueryExecution:
true)
         .Where(a => a.Location.Within(rectangularArea)))
    {
        Console.WriteLine("\t" + user);
    }
```

Next steps

Now that you have learned how to get started with geospatial support in Azure Cosmos DB, next you can:

- Learn more about Azure Cosmos DB Query
- Learn more about Geospatial and GeoJSON location data in Azure Cosmos DB
- Learn more about Index spatial data with Azure Cosmos DB

Index geospatial data with Azure Cosmos DB

12/4/2020 • 4 minutes to read • Edit Online

APPLIES TO: SQL API

We designed Azure Cosmos DB's database engine to be truly schema agnostic and provide first class support for JSON. The write optimized database engine of Azure Cosmos DB natively understands spatial data represented in the GeoJSON standard.

In a nutshell, the geometry is projected from geodetic coordinates onto a 2D plane then divided progressively into cells using a **quadtree**. These cells are mapped to 1D based on the location of the cell within a **Hilbert space filling curve**, which preserves locality of points. Additionally when location data is indexed, it goes through a process known as **tessellation**, that is, all the cells that intersect a location are identified and stored as keys in the Azure Cosmos DB index. At query time, arguments like points and Polygons are also tessellated to extract the relevant cell ID ranges, then used to retrieve data from the index.

If you specify an indexing policy that includes a spatial index for `/*` (all paths), then all data found within the container is indexed for efficient spatial queries.

> **NOTE**
>
> Azure Cosmos DB supports indexing of Points, LineStrings, Polygons, and MultiPolygons. If you index any one of these types, we will automatically index all other types. In other words, if you index Polygons, we'll also index Points, LineStrings, and MultiPolygons. Indexing a new spatial type does not affect the write RU charge or index size unless you have valid GeoJSON data of that type.

Modifying geospatial configuration

In your container, the **Geospatial Configuration** specifies how the spatial data will be indexed. Specify one **Geospatial Configuration** per container: geography or geometry.

You can toggle between the **geography** and **geometry** spatial type in the Azure portal. It's important that you create a valid spatial geometry indexing policy with a bounding box before switching to the geometry spatial type.

Here's how to set the **Geospatial Configuration** in **Data Explorer** within the Azure portal:

You can also modify the `geospatialConfig` in the .NET SDK to adjust the **Geospatial Configuration**:

If not specified, the `geospatialConfig` will default to the geography data type. When you modify the `geospatialConfig`, all existing geospatial data in the container will be reindexed.

Here is an example for modifying the geospatial data type to `geometry` by setting the `geospatialConfig` property and adding a **boundingBox**:

```
    //Retrieve the container's details
    ContainerResponse containerResponse = await client.GetContainer("db", "spatial").ReadContainerAsync();
    //Set GeospatialConfig to Geometry
    GeospatialConfig geospatialConfig = new GeospatialConfig(GeospatialType.Geometry);
    containerResponse.Resource.GeospatialConfig = geospatialConfig;
    // Add a spatial index including the required boundingBox
    SpatialPath spatialPath = new SpatialPath
            {
                Path = "/locations/*",
                BoundingBox = new BoundingBoxProperties(){
                    Xmin = 0,
                    Ymin = 0,
                    Xmax = 10,
                    Ymax = 10
                }
            };
    spatialPath.SpatialTypes.Add(SpatialType.Point);
    spatialPath.SpatialTypes.Add(SpatialType.LineString);
    spatialPath.SpatialTypes.Add(SpatialType.Polygon);
    spatialPath.SpatialTypes.Add(SpatialType.MultiPolygon);

    containerResponse.Resource.IndexingPolicy.SpatialIndexes.Add(spatialPath);

    // Update container with changes
    await client.GetContainer("db", "spatial").ReplaceContainerAsync(containerResponse.Resource);
```

Geography data indexing examples

The following JSON snippet shows an indexing policy with spatial indexing enabled for the **geography** data type. It is valid for spatial data with the geography data type and will index any GeoJSON Point, Polygon, MultiPolygon, or LineString found within documents for spatial querying. If you are modifying the indexing policy using the Azure portal, you can specify the following JSON for indexing policy to enable spatial indexing on your container:

Container indexing policy JSON with geography spatial indexing

```
{
    "automatic": true,
    "indexingMode": "Consistent",
    "includedPaths": [
        {
            "path": "/*"
        }
    ],
    "spatialIndexes": [
        {
            "path": "/*",
            "types": [
                "Point",
                "Polygon",
                "MultiPolygon",
                "LineString"
            ]
        }
    ],
    "excludedPaths": []
}
```

> **NOTE**
> If the location GeoJSON value within the document is malformed or invalid, then it will not get indexed for spatial querying. You can validate location values using ST_ISVALID and ST_ISVALIDDETAILED.

You can also modify indexing policy using the Azure CLI, PowerShell, or any SDK.

Geometry data indexing examples

With the **geometry** data type, similar to the geography data type, you must specify relevant paths and types to index. In addition, you must also specify a `boundingBox` within the indexing policy to indicate the desired area to be indexed for that specific path. Each geospatial path requires its own `boundingBox`.

The bounding box consists of the following properties:

- **xmin**: the minimum indexed x coordinate
- **ymin**: the minimum indexed y coordinate
- **xmax**: the maximum indexed x coordinate
- **ymax**: the maximum indexed y coordinate

A bounding box is required because geometric data occupies a plane that can be infinite. Spatial indexes, however, require a finite space. For the **geography** data type, the Earth is the boundary and you do not need to set a bounding box.

Create a bounding box that contains all (or most) of your data. Only operations computed on the objects that are entirely inside the bounding box will be able to utilize the spatial index. Making the bounding box larger than necessary will negatively impact query performance.

Here is an example indexing policy that indexes **geometry** data with **geospatialConfig** set to `geometry`:

```
{
    "indexingMode": "consistent",
    "automatic": true,
    "includedPaths": [
        {
            "path": "/*"
        }
    ],
    "excludedPaths": [
        {
            "path": "/\"_etag\"/?"
        }
    ],
    "spatialIndexes": [
        {
            "path": "/locations/*",
            "types": [
                "Point",
                "LineString",
                "Polygon",
                "MultiPolygon"
            ],
            "boundingBox": {
                "xmin": -10,
                "ymin": -20,
                "xmax": 10,
                "ymax": 20
            }
        }
    ]
}
```

The above indexing policy has a **boundingBox** of (-10, 10) for x coordinates and (-20, 20) for y coordinates. The container with the above indexing policy will index all Points, Polygons, MultiPolygons, and LineStrings that are entirely within this region.

> **NOTE**
>
> If you try to add an indexing policy with a **boundingBox** to a container with `geography` data type, it will fail. You should modify the container's **geospatialConfig** to be `geometry` before adding a **boundingBox**. You can add data and modify the remainder of your indexing policy (such as the paths and types) either before or after selecting the geospatial data type for the container.

Next steps

Now that you have learned how to get started with geospatial support in Azure Cosmos DB, next you can:

- Learn more about Azure Cosmos DB Query
- Learn more about Querying spatial data with Azure Cosmos DB
- Learn more about Geospatial and GeoJSON location data in Azure Cosmos DB

Working with Dates in Azure Cosmos DB

12/4/2020 • 3 minutes to read • Edit Online

APPLIES TO: SQL API

Azure Cosmos DB delivers schema flexibility and rich indexing via a native JSON data model. All Azure Cosmos DB resources including databases, containers, documents, and stored procedures are modeled and stored as JSON documents. As a requirement for being portable, JSON (and Azure Cosmos DB) supports only a small set of basic types: String, Number, Boolean, Array, Object, and Null. However, JSON is flexible and allow developers and frameworks to represent more complex types using these primitives and composing them as objects or arrays.

In addition to the basic types, many applications need the DateTime type to represent dates and timestamps. This article describes how developers can store, retrieve, and query dates in Azure Cosmos DB using the .NET SDK.

Storing DateTimes

Azure Cosmos DB supports JSON types such as - string, number, boolean, null, array, object. It does not directly support a DateTime type. Currently, Azure Cosmos DB doesn't support localization of dates. So, you need to store DateTimes as strings. The recommended format for DateTime strings in Azure Cosmos DB is `yyyy-MM-ddTHH:mm:ss.fffffffZ` which follows the ISO 8601 UTC standard. It is recommended to store all dates in Azure Cosmos DB as UTC. Converting the date strings to this format will allow sorting dates lexicographically. If non-UTC dates are stored, the logic must be handled at the client-side. To convert a local DateTime to UTC, the offset must be known/stored as a property in the JSON and the client can use the offset to compute the UTC DateTime value.

Range queries with DateTime strings as filters are only supported if the DateTime strings are all in UTC and the same length. In Azure Cosmos DB, the GetCurrentDateTime system function will return the current UTC date and time ISO 8601 string value in the format: `yyyy-MM-ddTHH:mm:ss.fffffffZ`.

Most applications can use the default string representation for DateTime for the following reasons:

- Strings can be compared, and the relative ordering of the DateTime values is preserved when they are transformed to strings.
- This approach doesn't require any custom code or attributes for JSON conversion.
- The dates as stored in JSON are human readable.
- This approach can take advantage of Azure Cosmos DB's index for fast query performance.

For example, the following snippet stores an `Order` object containing two DateTime properties - `ShipDate` and `OrderDate` as a document using the .NET SDK:

```
public class Order
{
    [JsonProperty(PropertyName="id")]
    public string Id { get; set; }
    public DateTime OrderDate { get; set; }
    public DateTime ShipDate { get; set; }
    public double Total { get; set; }
}

await container.CreateItemAsync(
    new Order
    {
        Id = "09152014101",
        OrderDate = DateTime.UtcNow.AddDays(-30),
        ShipDate = DateTime.UtcNow.AddDays(-14),
        Total = 113.39
    });
```

This document is stored in Azure Cosmos DB as follows:

```
{
    "id": "09152014101",
    "OrderDate": "2014-09-15T23:14:25.7251173Z",
    "ShipDate": "2014-09-30T23:14:25.7251173Z",
    "Total": 113.39
}
```

Alternatively, you can store DateTimes as Unix timestamps, that is, as a number representing the number of elapsed seconds since January 1, 1970. Azure Cosmos DB's internal Timestamp (`_ts`) property follows this approach. You can use the UnixDateTimeConverter class to serialize DateTimes as numbers.

Querying DateTimes in LINQ

The SQL .NET SDK automatically supports querying data stored in Azure Cosmos DB via LINQ. For example, the following snippet shows a LINQ query that filters orders that were shipped in the last three days:

```
    IQueryable<Order> orders = container.GetItemLinqQueryable<Order>(allowSynchronousQueryExecution:
true).Where(o => o.ShipDate >= DateTime.UtcNow.AddDays(-3));
```

Translated to the following SQL statement and executed on Azure Cosmos DB:

```
SELECT * FROM root WHERE (root["ShipDate"] >= "2014-09-30T23:14:25.7251173Z")
```

You can learn more about Azure Cosmos DB's SQL query language and the LINQ provider at Querying Cosmos DB in LINQ.

Indexing DateTimes for range queries

Queries are common with DateTime values. To execute these queries efficiently, you must have an index defined on any properties in the query's filter.

You can learn more about how to configure indexing policies at Azure Cosmos DB Indexing Policies.

Next Steps

- Download and run the Code samples on GitHub

- Learn more about SQL queries
- Learn more about Azure Cosmos DB Indexing Policies

Parameterized queries in Azure Cosmos DB

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: SQL API

Azure Cosmos DB supports queries with parameters expressed by the familiar @ notation. Parameterized SQL provides robust handling and escaping of user input, and prevents accidental exposure of data through SQL injection.

Examples

For example, you can write a query that takes `lastName` and `address.state` as parameters, and execute it for various values of `lastName` and `address.state` based on user input.

```
SELECT *
FROM Families f
WHERE f.lastName = @lastName AND f.address.state = @addressState
```

You can then send this request to Azure Cosmos DB as a parameterized JSON query like the following:

```
{
    "query": "SELECT * FROM Families f WHERE f.lastName = @lastName AND f.address.state = @addressState",
    "parameters": [
        {"name": "@lastName", "value": "Wakefield"},
        {"name": "@addressState", "value": "NY"},
    ]
}
```

The following example sets the TOP argument with a parameterized query:

```
{
    "query": "SELECT TOP @n * FROM Families",
    "parameters": [
        {"name": "@n", "value": 10},
    ]
}
```

Parameter values can be any valid JSON: strings, numbers, Booleans, null, even arrays or nested JSON. Since Azure Cosmos DB is schemaless, parameters aren't validated against any type.

Here are examples for parameterized queries in each Azure Cosmos DB SDK:

- .NET SDK
- Java
- Node.js
- Python

Next steps

- Azure Cosmos DB .NET samples
- Model document data

LINQ to SQL translation

12/4/2020 • 5 minutes to read • Edit Online

APPLIES TO: SQL API

The Azure Cosmos DB query provider performs a best effort mapping from a LINQ query into a Cosmos DB SQL query. If you want to get the SQL query that is translated from LINQ, use the `ToString()` method on the generated `IQueryable` object. The following description assumes a basic familiarity with LINQ. In addition to LINQ, Azure Cosmos DB also supports Entity Framework Core which works with SQL API.

The query provider type system supports only the JSON primitive types: numeric, Boolean, string, and null.

The query provider supports the following scalar expressions:

- Constant values, including constant values of the primitive data types at query evaluation time.
- Property/array index expressions that refer to the property of an object or an array element. For example:

```
family.Id;
family.children[0].familyName;
family.children[0].grade;
family.children[n].grade; //n is an int variable
```

- Arithmetic expressions, including common arithmetic expressions on numerical and Boolean values. For the complete list, see the Azure Cosmos DB SQL specification.

```
2 * family.children[0].grade;
x + y;
```

- String comparison expressions, which include comparing a string value to some constant string value.

```
mother.familyName == "Wakefield";
child.givenName == s; //s is a string variable
```

- Object/array creation expressions, which return an object of compound value type or anonymous type, or an array of such objects. You can nest these values.

```
new Parent { familyName = "Wakefield", givenName = "Robin" };
new { first = 1, second = 2 }; //an anonymous type with two fields
new int[] { 3, child.grade, 5 };
```

Using LINQ

You can create a LINQ query with `GetItemLinqQueryable`. This example shows LINQ query generation and asynchronous execution with a `FeedIterator`:

```
using (FeedIterator<Book> setIterator = container.GetItemLinqQueryable<Book>()
                      .Where(b => b.Title == "War and Peace")
                      .ToFeedIterator<Book>())
 {
     //Asynchronous query execution
     while (setIterator.HasMoreResults)
     {
         foreach(var item in await setIterator.ReadNextAsync()){
         {
             Console.WriteLine(item.cost);
         }
       }
     }
 }
```

Supported LINQ operators

The LINQ provider included with the SQL .NET SDK supports the following operators:

- **Select**: Projections translate to SELECT, including object construction.
- **Where**: Filters translate to WHERE, and support translation between `&&`, `||`, and `!` to the SQL operators
- **SelectMany**: Allows unwinding of arrays to the JOIN clause. Use to chain or nest expressions to filter on array elements.
- **OrderBy** and **OrderByDescending**: Translate to ORDER BY with ASC or DESC.
- **Count**, **Sum**, **Min**, **Max**, and **Average** operators for aggregation, and their async equivalents **CountAsync**, **SumAsync**, **MinAsync**, **MaxAsync**, and **AverageAsync**.
- **CompareTo**: Translates to range comparisons. Commonly used for strings, since they're not comparable in .NET.
- **Skip** and **Take**: Translates to OFFSET and LIMIT for limiting results from a query and doing pagination.
- **Math functions**: Supports translation from .NET `Abs`, `Acos`, `Asin`, `Atan`, `Ceiling`, `Cos`, `Exp`, `Floor`, `Log`, `Log10`, `Pow`, `Round`, `Sign`, `Sin`, `Sqrt`, `Tan`, and `Truncate` to the equivalent built-in mathematical functions.
- **String functions**: Supports translation from .NET `Concat`, `Contains`, `Count`, `EndsWith`, `IndexOf`, `Replace`, `Reverse`, `StartsWith`, `SubString`, `ToLower`, `ToUpper`, `TrimEnd`, and `TrimStart` to the equivalent built-in string functions.
- **Array functions**: Supports translation from .NET `Concat`, `Contains`, and `Count` to the equivalent built-in array functions.
- **Geospatial Extension functions**: Supports translation from stub methods `Distance`, `IsValid`, `IsValidDetailed`, and `Within` to the equivalent built-in geospatial functions.
- **User-Defined Function Extension function**: Supports translation from the stub method `UserDefinedFunctionProvider.Invoke` to the corresponding user-defined function.
- **Miscellaneous**: Supports translation of `Coalesce` and conditional operators. Can translate `Contains` to String CONTAINS, ARRAY_CONTAINS, or IN, depending on context.

Examples

The following examples illustrate how some of the standard LINQ query operators translate to queries in Azure Cosmos DB.

Select operator

The syntax is `input.Select(x => f(x))`, where `f` is a scalar expression. The `input`, in this case, would be an `IQueryable` object.

Select operator, example 1:

- **LINQ lambda expression**

```
input.Select(family => family.parents[0].familyName);
```

- SQL

```
SELECT VALUE f.parents[0].familyName
FROM Families f
```

Select operator, example 2:

- LINQ lambda expression

```
input.Select(family => family.children[0].grade + c); // c is an int variable
```

- SQL

```
SELECT VALUE f.children[0].grade + c
FROM Families f
```

Select operator, example 3:

- LINQ lambda expression

```
input.Select(family => new
{
    name = family.children[0].familyName,
    grade = family.children[0].grade + 3
});
```

- SQL

```
SELECT VALUE {"name":f.children[0].familyName,
              "grade": f.children[0].grade + 3 }
FROM Families f
```

SelectMany operator

The syntax is `input.SelectMany(x => f(x))`, where `f` is a scalar expression that returns a container type.

- LINQ lambda expression

```
input.SelectMany(family => family.children);
```

- SQL

```
SELECT VALUE child
FROM child IN Families.children
```

Where operator

The syntax is `input.Where(x => f(x))`, where `f` is a scalar expression, which returns a Boolean value.

Where operator, example 1:

- LINQ lambda expression

```
input.Where(family=> family.parents[0].familyName == "Wakefield");
```

- SQL

```
SELECT *
FROM Families f
WHERE f.parents[0].familyName = "Wakefield"
```

Where operator, example 2:

- LINQ lambda expression

```
input.Where(
    family => family.parents[0].familyName == "Wakefield" &&
    family.children[0].grade < 3);
```

- SQL

```
SELECT *
FROM Families f
WHERE f.parents[0].familyName = "Wakefield"
AND f.children[0].grade < 3
```

Composite SQL queries

You can compose the preceding operators to form more powerful queries. Since Cosmos DB supports nested containers, you can concatenate or nest the composition.

Concatenation

The syntax is `input(.|.SelectMany())(.Select()|.Where())*`. A concatenated query can start with an optional `SelectMany` query, followed by multiple `Select` or `Where` operators.

Concatenation, example 1:

- LINQ lambda expression

```
input.Select(family => family.parents[0])
    .Where(parent => parent.familyName == "Wakefield");
```

- SQL

```
SELECT *
FROM Families f
WHERE f.parents[0].familyName = "Wakefield"
```

Concatenation, example 2:

- LINQ lambda expression

```
input.Where(family => family.children[0].grade > 3)
    .Select(family => family.parents[0].familyName);
```

- SQL

```
SELECT VALUE f.parents[0].familyName
FROM Families f
WHERE f.children[0].grade > 3
```

Concatenation, example 3:

- LINQ lambda expression

```
input.Select(family => new { grade=family.children[0].grade}).
    Where(anon=> anon.grade < 3);
```

- SQL

```
SELECT *
FROM Families f
WHERE ({grade: f.children[0].grade}.grade > 3)
```

Concatenation, example 4:

- LINQ lambda expression

```
input.SelectMany(family => family.parents)
    .Where(parent => parents.familyName == "Wakefield");
```

- SQL

```
SELECT *
FROM p IN Families.parents
WHERE p.familyName = "Wakefield"
```

Nesting

The syntax is `input.SelectMany(x=>x.Q())` where `Q` is a `Select`, `SelectMany`, or `Where` operator.

A nested query applies the inner query to each element of the outer container. One important feature is that the inner query can refer to the fields of the elements in the outer container, like a self-join.

Nesting, example 1:

- LINQ lambda expression

```
input.SelectMany(family=>
    family.parents.Select(p => p.familyName));
```

- SQL

```
SELECT VALUE p.familyName
FROM Families f
JOIN p IN f.parents
```

Nesting, example 2:

- LINQ lambda expression

```
input.SelectMany(family =>
    family.children.Where(child => child.familyName == "Jeff"));
```

- SQL

```
SELECT *
FROM Families f
JOIN c IN f.children
WHERE c.familyName = "Jeff"
```

Nesting, example 3:

- LINQ lambda expression

```
input.SelectMany(family => family.children.Where(
    child => child.familyName == family.parents[0].familyName));
```

- SQL

```
SELECT *
FROM Families f
JOIN c IN f.children
WHERE c.familyName = f.parents[0].familyName
```

Next steps

- Azure Cosmos DB .NET samples
- Model document data

Indexing in Azure Cosmos DB - Overview

12/4/2020 • 4 minutes to read • Edit Online

APPLIES TO: SQL API

Azure Cosmos DB is a schema-agnostic database that allows you to iterate on your application without having to deal with schema or index management. By default, Azure Cosmos DB automatically indexes every property for all items in your container without having to define any schema or configure secondary indexes.

The goal of this article is to explain how Azure Cosmos DB indexes data and how it uses indexes to improve query performance. It is recommended to go through this section before exploring how to customize indexing policies.

From items to trees

Every time an item is stored in a container, its content is projected as a JSON document, then converted into a tree representation. What that means is that every property of that item gets represented as a node in a tree. A pseudo root node is created as a parent to all the first-level properties of the item. The leaf nodes contain the actual scalar values carried by an item.

As an example, consider this item:

```
{
    "locations": [
        { "country": "Germany", "city": "Berlin" },
        { "country": "France", "city": "Paris" }
    ],
    "headquarters": { "country": "Belgium", "employees": 250 },
    "exports": [
        { "city": "Moscow" },
        { "city": "Athens" }
    ]
}
```

It would be represented by the following tree:

Note how arrays are encoded in the tree: every entry in an array gets an intermediate node labeled with the index of that entry within the array (0, 1 etc.).

From trees to property paths

The reason why Azure Cosmos DB transforms items into trees is because it allows properties to be referenced by their paths within those trees. To get the path for a property, we can traverse the tree from the root node to that property, and concatenate the labels of each traversed node.

Here are the paths for each property from the example item described above:

- /locations/0/country: "Germany"
- /locations/0/city: "Berlin"
- /locations/1/country: "France"
- /locations/1/city: "Paris"
- /headquarters/country: "Belgium"
- /headquarters/employees: 250
- /exports/0/city: "Moscow"
- /exports/1/city: "Athens"

When an item is written, Azure Cosmos DB effectively indexes each property's path and its corresponding value.

Index kinds

Azure Cosmos DB currently supports three kinds of indexes.

Range Index

Range index is based on an ordered tree-like structure. The range index kind is used for:

- Equality queries:

  ```
  SELECT * FROM container c WHERE c.property = 'value'
  ```

  ```
  SELECT * FROM c WHERE c.property IN ("value1", "value2", "value3")
  ```

  Equality match on an array element

  ```
    SELECT * FROM c WHERE ARRAY_CONTAINS(c.tags, "tag1")
  ```

- Range queries:

  ```
  SELECT * FROM container c WHERE c.property > 'value'
  ```

  (works for `>`, `<`, `>=`, `<=`, `!=`)

- Checking for the presence of a property:

  ```
  SELECT * FROM c WHERE IS_DEFINED(c.property)
  ```

- String system functions:

  ```
  SELECT * FROM c WHERE CONTAINS(c.property, "value")
  ```

```
SELECT * FROM c WHERE STRINGEQUALS(c.property, "value")
```

- `ORDER BY` queries:

```
SELECT * FROM container c ORDER BY c.property
```

- `JOIN` queries:

```
SELECT child FROM container c JOIN child IN c.properties WHERE child = 'value'
```

Range indexes can be used on scalar values (string or number).

Spatial index

Spatial indices enable efficient queries on geospatial objects such as - points, lines, polygons, and multipolygon. These queries use ST_DISTANCE, ST_WITHIN, ST_INTERSECTS keywords. The following are some examples that use spatial index kind:

- Geospatial distance queries:

```
SELECT * FROM container c WHERE ST_DISTANCE(c.property, { "type": "Point", "coordinates": [0.0, 10.0]
}) < 40
```

- Geospatial within queries:

```
SELECT * FROM container c WHERE ST_WITHIN(c.property, {"type": "Point", "coordinates": [0.0, 10.0] })
```

- Geospatial intersect queries:

```
SELECT * FROM c WHERE ST_INTERSECTS(c.property, { 'type':'Polygon', 'coordinates': [[ [31.8, -5],
[32, -5], [31.8, -5] ]]  })
```

Spatial indexes can be used on correctly formatted GeoJSON objects. Points, LineStrings, Polygons, and MultiPolygons are currently supported.

Composite indexes

Composite indices increase the efficiency when you are performing operations on multiple fields. The composite index kind is used for:

- `ORDER BY` queries on multiple properties:

```
SELECT * FROM container c ORDER BY c.property1, c.property2
```

- Queries with a filter and `ORDER BY`. These queries can utilize a composite index if the filter property is added to the `ORDER BY` clause.

```
SELECT * FROM container c WHERE c.property1 = 'value' ORDER BY c.property1, c.property2
```

- Queries with a filter on two or more properties where at least one property is an equality filter

```
SELECT * FROM container c WHERE c.property1 = 'value' AND c.property2 > 'value'
```

As long as one filter predicate uses one of the index kind, the query engine will evaluate that first before scanning the rest. For example, if you have a SQL query such as `SELECT * FROM c WHERE c.firstName = "Andrew" and CONTAINS(c.lastName, "Liu")`

- The above query will first filter for entries where firstName = "Andrew" by using the index. It then pass all of the firstName = "Andrew" entries through a subsequent pipeline to evaluate the CONTAINS filter predicate.
- You can speed up queries and avoid full container scans when using functions that don't use the index (e.g. CONTAINS) by adding additional filter predicates that do use the index. The order of filter clauses isn't important. The query engine is will figure out which predicates are more selective and run the query accordingly.

Querying with indexes

The paths extracted when indexing data make it easy to lookup the index when processing a query. By matching the `WHERE` clause of a query with the list of indexed paths, it is possible to identify the items that match the query predicate very quickly.

For example, consider the following query: `SELECT location FROM location IN company.locations WHERE location.country = 'France'`. The query predicate (filtering on items, where any location has "France" as its country/region) would match the path highlighted in red below:

> **NOTE**
> An `ORDER BY` clause that orders by a single property *always* needs a range index and will fail if the path it references doesn't have one. Similarly, an `ORDER BY` query which orders by multiple properties *always* needs a composite index.

Next steps

Read more about indexing in the following articles:

- Indexing policy
- How to manage indexing policy

Time to Live (TTL) in Azure Cosmos DB

12/4/2020 • 3 minutes to read • Edit Online

APPLIES TO: SQL API

With **Time to Live** or TTL, Azure Cosmos DB provides the ability to delete items automatically from a container after a certain time period. By default, you can set time to live at the container level and override the value on a per-item basis. After you set the TTL at a container or at an item level, Azure Cosmos DB will automatically remove these items after the time period, since the time they were last modified. Time to live value is configured in seconds. When you configure TTL, the system will automatically delete the expired items based on the TTL value, without needing a delete operation that is explicitly issued by the client application. The maximum value for TTL is 2147483647.

Deletion of expired items is a background task that consumes left-over Request Units, that is Request Units that haven't been consumed by user requests. Even after the TTL has expired, if the container is overloaded with requests and if there aren't enough RU's available, the data deletion is delayed. Data is deleted once there are enough RUs available to perform the delete operation. Though the data deletion is delayed, data is not returned by any queries (by any API) after the TTL has expired.

> This content is related to Azure Cosmos DB transactional store TTL. If you are looking for analytical store TTL, that enables NoETL HTAP scenarios through Azure Synapse Link, please click here.

Time to live for containers and items

The time to live value is set in seconds, and it is interpreted as a delta from the time that an item was last modified. You can set time to live on a container or an item within the container:

1. **Time to Live on a container** (set using `DefaultTimeToLive`):
   - If missing (or set to null), items are not expired automatically.
   - If present and the value is set to "-1", it is equal to infinity, and items don't expire by default.
   - If present and the value is set to some *non-zero* number *"n"* – items will expire *"n"* seconds after their last modified time.
2. **Time to Live on an item** (set using `ttl`):
   - This Property is applicable only if `DefaultTimeToLive` is present and it is not set to null for the parent container.
   - If present, it overrides the `DefaultTimeToLive` value of the parent container.

Time to Live configurations

- If TTL is set to *"n"* on a container, then the items in that container will expire after *n* seconds. If there are items in the same container that have their own time to live, set to -1 (indicating they do not expire) or if some items have overridden the time to live setting with a different number, these items expire based on their own configured TTL value.
- If TTL is not set on a container, then the time to live on an item in this container has no effect.
- If TTL on a container is set to -1, an item in this container that has the time to live set to n, will expire after

n seconds, and remaining items will not expire.

Examples

This section shows some examples with different time to live values assigned to container and items:

Example 1

TTL on container is set to null (DefaultTimeToLive = null)

TTL ON ITEM	RESULT
ttl = null	TTL is disabled. The item will never expire (default).
ttl = -1	TTL is disabled. The item will never expire.
ttl = 2000	TTL is disabled. The item will never expire.

Example 2

TTL on container is set to -1 (DefaultTimeToLive = -1)

TTL ON ITEM	RESULT
ttl = null	TTL is enabled. The item will never expire (default).
ttl = -1	TTL is enabled. The item will never expire.
ttl = 2000	TTL is enabled. The item will expire after 2000 seconds.

Example 3

TTL on container is set to 1000 (DefaultTimeToLive = 1000)

TTL ON ITEM	RESULT
ttl = null	TTL is enabled. The item will expire after 1000 seconds (default).
ttl = -1	TTL is enabled. The item will never expire.
ttl = 2000	TTL is enabled. The item will expire after 2000 seconds.

Next steps

Learn how to configure Time to Live in the following articles:

- How to configure Time to Live

Unique key constraints in Azure Cosmos DB

12/4/2020 • 3 minutes to read • Edit Online

APPLIES TO: SQL API

Unique keys add a layer of data integrity to an Azure Cosmos container. You create a unique key policy when you create an Azure Cosmos container. With unique keys, you make sure that one or more values within a logical partition is unique. You also can guarantee uniqueness per partition key.

After you create a container with a unique key policy, the creation of a new or an update of an existing item resulting in a duplicate within a logical partition is prevented, as specified by the unique key constraint. The partition key combined with the unique key guarantees the uniqueness of an item within the scope of the container.

For example, consider an Azure Cosmos container with email address as the unique key constraint and `CompanyID` as the partition key. When you configure the user's email address with a unique key, each item has a unique email address within a given `CompanyID`. Two items can't be created with duplicate email addresses and with the same partition key value. In Azure Cosmos DB's SQL (Core) API, items are stored as JSON values. These JSON values are case sensitive. When you choose a property as a unique key, you can insert case sensitive values for that property. For example, If you have a unique key defined on the name property, "Gaby" is different from "gaby" and you can insert both into the container.

To create items with the same email address, but not the same first name, last name, and email address, add more paths to the unique key policy. Instead of creating a unique key based on the email address only, you also can create a unique key with a combination of the first name, last name, and email address. This key is known as a composite unique key. In this case, each unique combination of the three values within a given `CompanyID` is allowed.

For example, the container can contain items with the following values, where each item honors the unique key constraint.

COMPANYID	FIRST NAME	LAST NAME	EMAIL ADDRESS
Contoso	Gaby	Duperre	gaby@contoso.com
Contoso	Gaby	Duperre	gaby@fabrikam.com
Fabrikam	Gaby	Duperre	gaby@fabrikam.com
Fabrikam	Ivan	Duperre	gaby@fabrikam.com
Fabrkam		Duperre	gaby@fabraikam.com
Fabrkam			gaby@fabraikam.com

If you attempt to insert another item with the combinations listed in the previous table, you receive an error. The error indicates that the unique key constraint wasn't met. You receive either `Resource with specified ID or name already exists` or `Resource with specified ID, name, or unique index already exists` as a return message.

Define a unique key

You can define unique keys only when you create an Azure Cosmos container. A unique key is scoped to a logical partition. In the previous example, if you partition the container based on the ZIP code, you end up with duplicated items in each logical partition. Consider the following properties when you create unique keys:

- You can't update an existing container to use a different unique key. In other words, after a container is created with a unique key policy, the policy can't be changed.
- To set a unique key for an existing container, create a new container with the unique key constraint. Use the appropriate data migration tool to move the data from the existing container to the new container. For SQL containers, use the Data Migration tool to move data. For MongoDB containers, use mongoimport.exe or mongorestore.exe to move data.
- A unique key policy can have a maximum of 16 path values. For example, the values can be `/firstName`, `/lastName`, and `/address/zipCode`. Each unique key policy can have a maximum of 10 unique key constraints or combinations. The combined paths for each unique index constraint must not exceed 60 bytes. In the previous example, first name, last name, and email address together are one constraint. This constraint uses 3 out of the 16 possible paths.
- When a container has a unique key policy, Request Unit (RU) charges to create, update, and delete an item are slightly higher.
- Sparse unique keys are not supported. If some unique path values are missing, they're treated as null values, which take part in the uniqueness constraint. For this reason, there can be only a single item with a null value to satisfy this constraint.
- Unique key names are case-sensitive. For example, consider a container with the unique key constraint set to `/address/zipcode`. If your data has a field named `ZipCode`, Azure Cosmos DB inserts "null" as the unique key because `zipcode` isn't the same as `ZipCode`. Because of this case sensitivity, all other records with ZipCode can't be inserted because the duplicate "null" violates the unique key constraint.

Next steps

- Learn more about logical partitions
- Explore how to define unique keys when creating a container

Transactions and optimistic concurrency control

12/4/2020 • 5 minutes to read • Edit Online

APPLIES TO: SQL API

Database transactions provide a safe and predictable programming model to deal with concurrent changes to the data. Traditional relational databases, like SQL Server, allow you to write the business logic using stored-procedures and/or triggers, send it to the server for execution directly within the database engine. With traditional relational databases, you are required to deal with two different programming languages the (non-transactional) application programming language such as JavaScript, Python, C#, Java, etc. and the transactional programming language (such as T-SQL) that is natively executed by the database.

The database engine in Azure Cosmos DB supports full ACID (Atomicity, Consistency, Isolation, Durability) compliant transactions with snapshot isolation. All the database operations within the scope of a container's logical partition are transactionally executed within the database engine that is hosted by the replica of the partition. These operations include both write (updating one or more items within the logical partition) and read operations. The following table illustrates different operations and transaction types:

OPERATION	OPERATION TYPE	SINGLE OR MULTI ITEM TRANSACTION
Insert (without a pre/post trigger)	Write	Single item transaction
Insert (with a pre/post trigger)	Write and Read	Multi-item transaction
Replace (without a pre/post trigger)	Write	Single item transaction
Replace (with a pre/post trigger)	Write and Read	Multi-item transaction
Upsert (without a pre/post trigger)	Write	Single item transaction
Upsert (with a pre/post trigger)	Write and Read	Multi-item transaction
Delete (without a pre/post trigger)	Write	Single item transaction
Delete (with a pre/post trigger)	Write and Read	Multi-item transaction
Execute stored procedure	Write and Read	Multi-item transaction
System initiated execution of a merge procedure	Write	Multi-item transaction
System initiated execution of deleting items based on expiration (TTL) of an item	Write	Multi-item transaction
Read	Read	Single-item transaction
Change Feed	Read	Multi-item transaction
Paginated Read	Read	Multi-item transaction

OPERATION	OPERATION TYPE	SINGLE OR MULTI ITEM TRANSACTION
Paginated Query	Read	Multi-item transaction
Execute UDF as part of the paginated query	Read	Multi-item transaction

Multi-item transactions

Azure Cosmos DB allows you to write stored procedures, pre/post triggers, user-defined-functions (UDFs) and merge procedures in JavaScript. Azure Cosmos DB natively supports JavaScript execution inside its database engine. You can register stored procedures, pre/post triggers, user-defined-functions (UDFs) and merge procedures on a container and later execute them transactionally within the Azure Cosmos database engine. Writing application logic in JavaScript allows natural expression of control flow, variable scoping, assignment, and integration of exception handling primitives within the database transactions directly in the JavaScript language.

The JavaScript-based stored procedures, triggers, UDFs, and merge procedures are wrapped within an ambient ACID transaction with snapshot isolation across all items within the logical partition. During the course of its execution, if the JavaScript program throws an exception, the entire transaction is aborted and rolled-back. The resulting programming model is simple yet powerful. JavaScript developers get a durable programming model while still using their familiar language constructs and library primitives.

The ability to execute JavaScript directly within the database engine provides performance and transactional execution of database operations against the items of a container. Furthermore, since Azure Cosmos database engine natively supports JSON and JavaScript, there is no impedance mismatch between the type systems of an application and the database.

Optimistic concurrency control

Optimistic concurrency control allows you to prevent lost updates and deletes. Concurrent, conflicting operations are subjected to the regular pessimistic locking of the database engine hosted by the logical partition that owns the item. When two concurrent operations attempt to update the latest version of an item within a logical partition, one of them will win and the other will fail. However, if one or two operations attempting to concurrently update the same item had previously read an older value of the item, the database doesn't know if the previously read value by either or both the conflicting operations was indeed the latest value of the item. Fortunately, this situation can be detected with the **Optimistic Concurrency Control (OCC)** before letting the two operations enter the transaction boundary inside the database engine. OCC protects your data from accidentally overwriting changes that were made by others. It also prevents others from accidentally overwriting your own changes.

The concurrent updates of an item are subjected to the OCC by Azure Cosmos DB's communication protocol layer. Azure Cosmos database ensures that the client-side version of the item that you are updating (or deleting) is the same as the version of the item in the Azure Cosmos container. This ensures that your writes are protected from being overwritten accidentally by the writes of others and vice versa. In a multi-user environment, the optimistic concurrency control protects you from accidentally deleting or updating wrong version of an item. As such, items are protected against the infamous "lost update" or "lost delete" problems.

Every item stored in an Azure Cosmos container has a system defined `_etag` property. The value of the `_etag` is automatically generated and updated by the server every time the item is updated. `_etag` can be used with the client supplied `if-match` request header to allow the server to decide whether an item can be conditionally updated. The value of the `if-match` header matches the value of the `_etag` at the server, the item is then updated. If the value of the `if-match` request header is no longer current, the server rejects the operation with an "HTTP 412 Precondition failure" response message. The client then can re-fetch the item to acquire the current version of the item on the server or override the version of item in the server with its own `_etag` value for the item. In

addition, `_etag` can be used with the `if-none-match` header to determine whether a refetch of a resource is needed.

The item's `_etag` value changes every time the item is updated. For replace item operations, `if-match` must be explicitly expressed as a part of the request options. For an example, see the sample code in GitHub. `_etag` values are implicitly checked for all written items touched by the stored procedure. If any conflict is detected, the stored procedure will roll back the transaction and throw an exception. With this method, either all or no writes within the stored procedure are applied atomically. This is a signal to the application to reapply updates and retry the original client request.

Next steps

Learn more about database transactions and optimistic concurrency control in the following articles:

- Working with Azure Cosmos databases, containers and items
- Consistency levels
- Conflict types and resolution policies
- Using TransactionalBatch
- Stored procedures, triggers, and user-defined functions

Azure Cosmos DB Attachments

11/2/2020 • 3 minutes to read • Edit Online

APPLIES TO: SQL API

Azure Cosmos DB attachments are special items that contain references to an associated metadata with an external blob or media file.

Azure Cosmos DB supports two types of attachments:

- **Unmanaged Attachments** are a wrapper around an URI reference to a blob that is stored in an external service (for example, Azure Storage, OneDrive, etc.). This approach is similar to storing a URI property in a standard Azure Cosmos DB item.
- **Managed Attachments** are blobs managed and stored internally by Azure Cosmos DB and exposed via a system-generated mediaLink.

> **NOTE**
>
> Attachment is a legacy feature. Their support is scoped to offer continued functionality if you are already using this feature.
>
> Instead of using attachments, we recommend you to use Azure Blob Storage as a purpose-built blob storage service to store blob data . You can continue to store metadata related to blobs, along with reference URI links, in Azure Cosmos DB as item properties. Storing this data in Azure Cosmos DB provides the ability to query metadata and links to blobs stored in Azure Blob Storage.
>
> Microsoft is committed to provide a minimum 36-month notice prior to fully deprecating attachments – which will be announced at a further date.

Known limitations

Azure Cosmos DB's managed attachments are distinct from its support for standard items – for which it offers unlimited scalability, global distribution, and integration with other Azure services.

- Attachments aren't supported in all versions of the Azure Cosmos DB's SDKs.
- Managed attachments are limited to 2 GB of storage per database account.
- Managed attachments aren't compatible with Azure Cosmos DB's global distribution, and they aren't replicated across regions.

Migrating Attachments to Azure Blob Storage

We recommend migrating Azure Cosmos DB attachments to Azure Blob Storage by following these steps:

1. Copy attachments data from your source Azure Cosmos DB container to your target Azure Blob Storage container.
2. Validate the uploaded blob data in the target Azure Blob Storage container.
3. If applicable, add URI references to the blobs contained in Azure Blob Storage as string properties within your Azure Cosmos DB dataset.
4. Refactor your application code to read and write blobs from the new Azure Blob Storage container.

The following dotnet code sample shows how to copy attachments from Azure Cosmos DB to Azure Blob storage as part of a migration flow by using Azure Cosmos DB's .NET SDK v2 and Azure Blob Storage .NET SDK v12. Make

sure to replace the `<placeholder values>` for the source Azure Cosmos DB account and target Azure Blob storage container.

```
using System;
using System.IO;
using System.Threading.Tasks;
using Microsoft.Azure.Documents;
using Microsoft.Azure.Documents.Client;
using Azure.Storage.Blobs;
using Azure.Storage.Blobs.Models;

namespace attachments
{
    class Program
    {
        private static string cosmosAccount = "<Your_Azure_Cosmos_account_URI>";
        private static string cosmosKey = "<Your_Azure_Cosmos_account_PRIMARY_KEY>";
        private static string cosmosDatabaseName = "<Your_Azure_Cosmos_database>";
        private static string cosmosCollectionName = "<Your_Azure_Cosmos_collection>";
        private static string storageConnectionString = "<Your_Azure_Storage_connection_string>";
        private static string storageContainerName = "<Your_Azure_Storage_container_name>";
        private static DocumentClient cosmosClient = new DocumentClient(new Uri(cosmosAccount), cosmosKey);
        private static BlobServiceClient storageClient = new BlobServiceClient(storageConnectionString);
        private static BlobContainerClient storageContainerClient =
storageClient.GetBlobContainerClient(storageContainerName);

        static void Main(string[] args)
        {
            CopyAttachmentsToBlobsAsync().Wait();
        }

        private async static Task CopyAttachmentsToBlobsAsync()
        {
            Console.WriteLine("Copying Azure Cosmos DB Attachments to Azure Blob Storage ...");

            int totalCount = 0;
            string docContinuation = null;

            // Iterate through each item (document in v2) in the Azure Cosmos DB container (collection in v2)
to look for attachments.
            do
            {
                FeedResponse<dynamic> response = await cosmosClient.ReadDocumentFeedAsync(
                    UriFactory.CreateDocumentCollectionUri(cosmosDatabaseName, cosmosCollectionName),
                    new FeedOptions
                    {
                        MaxItemCount = -1,
                        RequestContinuation = docContinuation
                    });
                docContinuation = response.ResponseContinuation;

                foreach (Document document in response)
                {
                    string attachmentContinuation = null;
                    PartitionKey docPartitionKey = new PartitionKey(document.Id);

                    // Iterate through each attachment within the item (if any).
                    do
                    {
                        FeedResponse<Attachment> attachments = await cosmosClient.ReadAttachmentFeedAsync(
                            document.SelfLink,
                            new FeedOptions
                            {
                                PartitionKey = docPartitionKey,
                                RequestContinuation = attachmentContinuation
                            }
                        );
```

```
                        };
                        attachmentContinuation = attachments.ResponseContinuation;

                        foreach (var attachment in attachments)
                        {
                            // Download the attachment in to local memory.
                            MediaResponse content = await cosmosClient.ReadMediaAsync(attachment.MediaLink);

                            byte[] buffer = new byte[content.ContentLength];
                            await content.Media.ReadAsync(buffer, 0, buffer.Length);

                            // Upload the locally buffered attachment to blob storage
                            string blobId = String.Concat(document.Id, "-", attachment.Id);

                            Azure.Response<BlobContentInfo> uploadedBob = await
storageContainerClient.GetBlobClient(blobId).UploadAsync(
                                new MemoryStream(buffer, writable: false),
                                true
                            );

                            Console.WriteLine("Copied attachment ... Item Id: {0} , Attachment Id: {1}, Blob
Id: {2}", document.Id, attachment.Id, blobId);
                            totalCount++;

                            // Clean up attachment from Azure Cosmos DB.
                            // Warning: please verify you've succesfully migrated attachments to blog storage
prior to cleaning up Azure Cosmos DB.
                            // await cosmosClient.DeleteAttachmentAsync(
                            //     attachment.SelfLink,
                            //     new RequestOptions { PartitionKey = docPartitionKey }
                            // );

                            // Console.WriteLine("Cleaned up attachment ... Document Id: {0} , Attachment Id:
{1}", document.Id, attachment.Id);
                        }

                    } while (!string.IsNullOrEmpty(attachmentContinuation));
                }
            }
            while (!string.IsNullOrEmpty(docContinuation));

            Console.WriteLine("Finished copying {0} attachments to blob storage", totalCount);
        }
    }
}
```

Next steps

- Get started with Azure Blob storage
- Get references for using attachments via Azure Cosmos DB's .NET SDK v2
- Get references for using attachments via Azure Cosmos DB's Java SDK v2
- Get references for using attachments via Azure Cosmos DB's REST API

Change feed design patterns in Azure Cosmos DB

12/4/2020 • 7 minutes to read • Edit Online

APPLIES TO: SQL API

The Azure Cosmos DB change feed enables efficient processing of large datasets with a high volume of writes. Change feed also offers an alternative to querying an entire dataset to identify what has changed. This document focuses on common change feed design patterns, design tradeoffs, and change feed limitations.

Azure Cosmos DB is well-suited for IoT, gaming, retail, and operational logging applications. A common design pattern in these applications is to use changes to the data to trigger additional actions. Examples of additional actions include:

- Triggering a notification or a call to an API, when an item is inserted or updated.
- Real-time stream processing for IoT or real-time analytics processing on operational data.
- Data movement such as synchronizing with a cache, a search engine, a data warehouse, or cold storage.

The change feed in Azure Cosmos DB enables you to build efficient and scalable solutions for each of these patterns, as shown in the following image:

Event computing and notifications

The Azure Cosmos DB change feed can simplify scenarios that need to trigger a notification or send a call to an API based on a certain event. You can use the Change Feed Process Library to automatically poll your container for changes and call an external API each time there is a write or update.

You can also selectively trigger a notification or send a call to an API based on specific criteria. For example, if you are reading from the change feed using Azure Functions, you can put logic into the function to only send a notification if a specific criteria has been met. While the Azure Function code would execute during each write and update, the notification would only be sent if specific criteria had been met.

Real-time stream processing

The Azure Cosmos DB change feed can be used for real-time stream processing for IoT or real-time analytics processing on operational data. For example, you might receive and store event data from devices, sensors, infrastructure and applications, and process these events in real time, using Spark. The following image shows how you can implement a lambda architecture using the Azure Cosmos DB via change feed:

In many cases, stream processing implementations first receive a high volume of incoming data into a temporary message queue such as Azure Event Hub or Apache Kafka. The change feed is a great alternative due to Azure Cosmos DB's ability to support a sustained high rate of data ingestion with guaranteed low read and write latency. The advantages of the Azure Cosmos DB change feed over a message queue include:

Data persistence

Data written to Azure Cosmos DB will show up in the change feed and be retained until deleted. Message queues typically have a maximum retention period. For example, Azure Event Hub offers a maximum data retention of 90 days.

Querying ability

In addition to reading from a Cosmos container's change feed, you can also run SQL queries on the data stored in Azure Cosmos DB. The change feed isn't a duplication of data already in the container but rather just a different mechanism of reading the data. Therefore, if you read data from the change feed, it will always be consistent with queries of the same Azure Cosmos DB container.

High availability

Azure Cosmos DB offers up to 99.999% read and write availability. Unlike many message queues, Azure Cosmos DB data can be easily globally distributed and configured with an RTO (Recovery Time Objective) of zero.

After processing items in the change feed, you can build a materialized view and persist aggregated values back in Azure Cosmos DB. If you're using Azure Cosmos DB to build a game, you can, for example, use change feed to implement real-time leaderboards based on scores from completed games.

Data movement

You can also read from the change feed for real-time data movement.

For example, the change feed helps you perform the following tasks efficiently:

- Update a cache, search index, or data warehouse with data stored in Azure Cosmos DB.
- Perform zero down-time migrations to another Azure Cosmos account or another Azure Cosmos container with a different logical partition key.

- Implement an application-level data tiering and archival. For example, you can store "hot data" in Azure Cosmos DB and age out "cold data" to other storage systems such as Azure Blob Storage.

When you have to denormalize data across partitions and containers, you can read from your container's change feed as a source for this data replication. Real-time data replication with the change feed can only guarantee eventual consistency. You can monitor how far the Change Feed Processor lags behind in processing changes in your Cosmos container.

Event sourcing

The event sourcing pattern involves using an append-only store to record the full series of actions on that data. Azure Cosmos DB's change feed is a great choice as a central data store in event sourcing architectures where all data ingestion is modeled as writes (no updates or deletes). In this case, each write to Azure Cosmos DB is an "event" and you'll have a full record of past events in the change feed. Typical uses of the events published by the central event store are for maintaining materialized views or for integration with external systems. Because there is no time limit for retention in the change feed, you can replay all past events by reading from the beginning of your Cosmos container's change feed.

You can have multiple change feed consumers subscribe to the same container's change feed. Aside from the lease container's provisioned throughput, there is no cost to utilize the change feed. The change feed is available in every container regardless of whether it is utilized.

Azure Cosmos DB is a great central append-only persistent data store in the event sourcing pattern because of its strengths in horizontal scalability and high availability. In addition, the change Feed Processor library offers an "at least once" guarantee, ensuring that you won't miss processing any events.

Current limitations

The change feed has important limitations that you should understand. While items in a Cosmos container will always remain in the change feed, the change feed is not a full operation log. There are important areas to consider when designing an application that utilizes the change feed.

Intermediate updates

Only the most recent change for a given item is included in the change feed. When processing changes, you will read the latest available item version. If there are multiple updates to the same item in a short period of time, it is possible to miss processing intermediate updates. If you would like to track updates and be able to replay past updates to an item, we recommend modeling these updates as a series of writes instead.

Deletes

The change feed does not capture deletes. If you delete an item from your container, it is also removed from the change feed. The most common method of handling this is adding a soft marker on the items that are being deleted. You can add a property called "deleted" and set it to "true" at the time of deletion. This document update will show up in the change feed. You can set a TTL on this item so that it can be automatically deleted later.

Guaranteed order

There is guaranteed order in the change feed within a partition key value but not across partition key values. You should select a partition key that gives you a meaningful order guarantee.

For example, consider a retail application using the event sourcing design pattern. In this application, different user actions are each "events" which are modeled as writes to Azure Cosmos DB. Imagine if some example events occurred in the following sequence:

1. Customer adds Item A to their shopping cart
2. Customer adds Item B to their shopping cart
3. Customer removes Item A from their shopping cart

4. Customer checks out and shopping cart contents are shipped

A materialized view of current shopping cart contents is maintained for each customer. This application must ensure that these events are processed in the order in which they occur. If, for example, the cart checkout were to be processed before Item A's removal, it is likely that the customer would have had Item A shipped, as opposed to the desired Item B. In order to guarantee that these four events are processed in order of their occurrence, they should fall within the same partition key value. If you select **username** (each customer has a unique username) as the partition key, you can guarantee that these events show up in the change feed in the same order in which they are written to Azure Cosmos DB.

Examples

Here are some real-world change feed code examples that extend beyond the scope of the samples provided in Microsoft docs:

- Introduction to the change feed
- IoT use case centered around the change feed
- Retail use case centered around the change feed

Next steps

- Change feed overview
- Options to read change feed
- Using change feed with Azure Functions

Reading Azure Cosmos DB change feed

12/4/2020 • 6 minutes to read • Edit Online

APPLIES TO: SQL API

You can work with the Azure Cosmos DB change feed using either a push model or a pull model. With a push model, the change feed processor pushes work to a client that has business logic for processing this work. However, the complexity in checking for work and storing state for the last processed work is handled within the change feed processor.

With a pull model, the client has to pull the work from the server. The client, in this case, not only has business logic for processing work but also storing state for the last processed work, handling load balancing across multiple clients processing work in parallel, and handling errors.

When reading from the Azure Cosmos DB change feed, we usually recommend using a push model because you won't need to worry about:

- Polling the change feed for future changes.
- Storing state for the last processed change. When reading from the change feed, this is automatically stored in a lease container.
- Load balancing across multiple clients consuming changes. For example, if one client can't keep up with processing changes and another has available capacity.
- Handling errors. For example, automatically retrying failed changes that weren't correctly processed after an unhandled exception in code or a transient network issue.

The majority of scenarios that use the Azure Cosmos DB change feed will use one of the push model options. However, there are some scenarios where you might want the additional low level control of the pull model. These include:

- Reading changes from a particular partition key
- Controlling the pace at which your client receives changes for processing
- Doing a one-time read of the existing data in the change feed (for example, to do a data migration)

Reading change feed with a push model

Using a push model is the easiest way to read from the change feed. There are two ways you can read from the change feed with a push model: Azure Functions Cosmos DB triggers and the change feed processor library. Azure Functions uses the change feed processor behind the scenes, so these are both very similar ways to read the change feed. Think of Azure Functions as simply a hosting platform for the change feed processor, not an entirely different way of reading the change feed.

Azure Functions

Azure Functions is the simplest option if you are just getting started using the change feed. Due to its simplicity, it is also the recommended option for most change feed use cases. When you create an Azure Functions trigger for Azure Cosmos DB, you select the container to connect, and the Azure Function gets triggered whenever there is a change in the container. Because Azure Functions uses the change feed processor behind the scenes, it automatically parallelizes change processing across your container's partitions.

Developing with Azure Functions is an easy experience and can be faster than deploying the change feed processor on your own. Triggers can be created using the Azure Functions portal or programmatically using SDKs. Visual Studio and VS Code provide support to write Azure Functions, and you can even use the Azure Functions

CLI for cross-platform development. You can write and debug the code on your desktop, and then deploy the function with one click. See Serverless database computing using Azure Functions and Using change feed with Azure Functions articles to learn more.

Change feed processor library

The change feed processor gives you more control of the change feed and still hides most complexity. The change feed processor library follows the observer pattern, where your processing function is called by the library. The change feed processor library will automatically check for changes and, if changes are found, "push" these to the client. If you have a high throughput change feed, you can instantiate multiple clients to read the change feed. The change feed processor library will automatically divide the load among the different clients. You won't have to implement any logic for load balancing across multiple clients or any logic to maintain the lease state.

The change feed processor library guarantees an "at-least-once" delivery of all of the changes. In other words, if you use the change feed processor library, your processing function will be called successfully for every item in the change feed. If there is an unhandled exception in the business logic in your processing function, the failed changes will be retried until they are processed successfully. To prevent your change feed processor from getting "stuck" continuously retrying the same changes, add logic in your processing function to write documents, upon exception, to a dead-letter queue. Learn more about error handling.

In Azure Functions, the recommendation for handling errors is the same. You should still add logic in your delegate code to write documents, upon exception, to a dead-letter queue. However, if there is an unhandled exception in your Azure Function, the change that generated the exception won't be automatically retried. If there is an unhandled exception in the business logic, the Azure Function will move on to processing the next change. The Azure Function won't retry the same failed change.

Like Azure Functions, developing with the change feed processor library is easy. However, you are responsible for deploying one or more hosts for the change feed processor. A host is an application instance that uses the change feed processor to listen for changes. While Azure Functions has capabilities for automatic scaling, you are responsible for scaling your hosts. To learn more, see using the change feed processor. The change feed processor library is part of the Azure Cosmos DB SDK V3.

Reading change feed with a pull model

The change feed pull model allows you to consume the change feed at your own pace. Changes must be requested by the client and there is no automatic polling for changes. If you want to permanently "bookmark" the last processed change (similar to the push model's lease container), you'll need to save a continuation token.

Using the change feed pull model, you get more low level control of the change feed. When reading the change feed with the pull model, you have three options:

- Read changes for an entire container
- Read changes for a specific FeedRange
- Read changes for a specific partition key value

You can parallelize the processing of changes across multiple clients, just as you can with the change feed processor. However, the pull model does not automatically handle load-balancing across clients. When you use the pull model to parallelize processing of the change feed, you'll first obtain a list of FeedRanges. A FeedRange spans a range of partition key values. You'll need to have an orchestrator process that obtains FeedRanges and distributes them among your machines. You can then use these FeedRanges to have multiple machines read the change feed in parallel.

There is no built-in "at-least-once" delivery guarantee with the pull model. The pull model gives you low level control to decide how you would like to handle errors.

> **NOTE**
> The change feed pull model is currently in preview in the Azure Cosmos DB .NET SDK only. The preview is not yet available for other SDK versions.

Change feed in APIs for Cassandra and MongoDB

Change feed functionality is surfaced as change streams in MongoDB API and Query with predicate in Cassandra API. To learn more about the implementation details for MongoDB API, see the Change streams in the Azure Cosmos DB API for MongoDB.

Native Apache Cassandra provides change data capture (CDC), a mechanism to flag specific tables for archival as well as rejecting writes to those tables once a configurable size-on-disk for the CDC log is reached. The change feed feature in Azure Cosmos DB API for Cassandra enhances the ability to query the changes with predicate via CQL. To learn more about the implementation details, see Change feed in the Azure Cosmos DB API for Cassandra.

Next steps

You can now continue to learn more about change feed in the following articles:

- Overview of change feed
- Using change feed with Azure Functions
- Using change feed processor library

Change feed processor in Azure Cosmos DB

12/4/2020 • 8 minutes to read • Edit Online

APPLIES TO: SQL API

The change feed processor is part of the Azure Cosmos DB SDK V3. It simplifies the process of reading the change feed and distribute the event processing across multiple consumers effectively.

The main benefit of change feed processor library is its fault-tolerant behavior that assures an "at-least-once" delivery of all the events in the change feed.

Components of the change feed processor

There are four main components of implementing the change feed processor:

1. **The monitored container:** The monitored container has the data from which the change feed is generated. Any inserts and updates to the monitored container are reflected in the change feed of the container.
2. **The lease container:** The lease container acts as a state storage and coordinates processing the change feed across multiple workers. The lease container can be stored in the same account as the monitored container or in a separate account.
3. **The host:** A host is an application instance that uses the change feed processor to listen for changes. Multiple instances with the same lease configuration can run in parallel, but each instance should have a different **instance name**.
4. **The delegate:** The delegate is the code that defines what you, the developer, want to do with each batch of changes that the change feed processor reads.

To further understand how these four elements of change feed processor work together, let's look at an example in the following diagram. The monitored container stores documents and uses 'City' as the partition key. We see that the partition key values are distributed in ranges that contain items. There are two host instances and the change feed processor is assigning different ranges of partition key values to each instance to maximize compute distribution. Each range is being read in parallel and its progress is maintained separately from other ranges in the lease container.

Implementing the change feed processor

The point of entry is always the monitored container, from a `Container` instance you call `GetChangeFeedProcessorBuilder`:

```
/// <summary>
/// Start the Change Feed Processor to listen for changes and process them with the HandleChangesAsync
implementation.
/// </summary>
private static async Task<ChangeFeedProcessor> StartChangeFeedProcessorAsync(
    CosmosClient cosmosClient,
    IConfiguration configuration)
{
    string databaseName = configuration["SourceDatabaseName"];
    string sourceContainerName = configuration["SourceContainerName"];
    string leaseContainerName = configuration["LeasesContainerName"];

    Container leaseContainer = cosmosClient.GetContainer(databaseName, leaseContainerName);
    ChangeFeedProcessor changeFeedProcessor = cosmosClient.GetContainer(databaseName, sourceContainerName)
        .GetChangeFeedProcessorBuilder<ToDoItem>(processorName: "changeFeedSample", HandleChangesAsync)
            .WithInstanceName("consoleHost")
            .WithLeaseContainer(leaseContainer)
            .Build();

    Console.WriteLine("Starting Change Feed Processor...");
    await changeFeedProcessor.StartAsync();
    Console.WriteLine("Change Feed Processor started.");
    return changeFeedProcessor;
}
```

Where the first parameter is a distinct name that describes the goal of this processor and the second name is the delegate implementation that will handle changes.

An example of a delegate would be:

```
/// <summary>
/// The delegate receives batches of changes as they are generated in the change feed and can process
them.
/// </summary>
static async Task HandleChangesAsync(IReadOnlyCollection<ToDoItem> changes, CancellationToken
cancellationToken)
{
    Console.WriteLine("Started handling changes...");
    foreach (ToDoItem item in changes)
    {
        Console.WriteLine($"Detected operation for item with id {item.id}, created at
{item.creationTime}.");
        // Simulate some asynchronous operation
        await Task.Delay(10);
    }

    Console.WriteLine("Finished handling changes.");
}
```

Finally you define a name for this processor instance with `WithInstanceName` and which is the container to maintain the lease state with `WithLeaseContainer`.

Calling `Build` will give you the processor instance that you can start by calling `StartAsync`.

Processing life cycle

The normal life cycle of a host instance is:

1. Read the change feed.
2. If there are no changes, sleep for a predefined amount of time (customizable with `WithPollInterval` in the Builder) and go to #1.
3. If there are changes, send them to the **delegate**.
4. When the delegate finishes processing the changes **successfully**, update the lease store with the latest processed point in time and go to #1.

Error handling

The change feed processor is resilient to user code errors. That means that if your delegate implementation has an unhandled exception (step #4), the thread processing that particular batch of changes will be stopped, and a new thread will be created. The new thread will check which was the latest point in time the lease store has for that range of partition key values, and restart from there, effectively sending the same batch of changes to the delegate. This behavior will continue until your delegate processes the changes correctly and it's the reason the change feed processor has an "at least once" guarantee, because if the delegate code throws an exception, it will retry that batch.

To prevent your change feed processor from getting "stuck" continuously retrying the same batch of changes, you should add logic in your delegate code to write documents, upon exception, to a dead-letter queue. This design ensures that you can keep track of unprocessed changes while still being able to continue to process future changes. The dead-letter queue might be another Cosmos container. The exact data store does not matter, simply that the unprocessed changes are persisted.

In addition, you can use the change feed estimator to monitor the progress of your change feed processor instances as they read the change feed. You can use this estimation to understand if your change feed processor is "stuck" or lagging behind due to available resources like CPU, memory, and network bandwidth.

Deployment unit

A single change feed processor deployment unit consists of one or more instances with the same `processorName` and lease container configuration. You can have many deployment units where each one has a different business flow for the changes and each deployment unit consisting of one or more instances.

For example, you might have one deployment unit that triggers an external API anytime there is a change in your container. Another deployment unit might move data, in real time, each time there is a change. When a change happens in your monitored container, all your deployment units will get notified.

Dynamic scaling

As mentioned before, within a deployment unit you can have one or more instances. To take advantage of the compute distribution within the deployment unit, the only key requirements are:

1. All instances should have the same lease container configuration.
2. All instances should have the same `processorName`.
3. Each instance needs to have a different instance name (`WithInstanceName`).

If these three conditions apply, then the change feed processor will, using an equal distribution algorithm, distribute all the leases in the lease container across all running instances of that deployment unit and parallelize compute. One lease can only be owned by one instance at a given time, so the maximum number of instances equals to the number of leases.

The number of instances can grow and shrink, and the change feed processor will dynamically adjust the load by redistributing accordingly.

Moreover, the change feed processor can dynamically adjust to containers scale due to throughput or storage increases. When your container grows, the change feed processor transparently handles these scenarios by dynamically increasing the leases and distributing the new leases among existing instances.

Change feed and provisioned throughput

Change feed read operations on the monitored container will consume RUs.

Operations on the lease container consume RUs. The higher the number of instances using the same lease container, the higher the potential RU consumption will be. Remember to monitor your RU consumption on the leases container if you decide to scale and increment the number of instances.

Starting time

By default, when a change feed processor starts the first time, it will initialize the leases container, and start its processing life cycle. Any changes that happened in the monitored container before the change feed processor was initialized for the first time won't be detected.

Reading from a previous date and time

It's possible to initialize the change feed processor to read changes starting at a **specific date and time**, by passing an instance of a `DateTime` to the `WithStartTime` builder extension:

```
Container leaseContainer = client.GetContainer(databaseId, Program.leasesContainer);
Container monitoredContainer = client.GetContainer(databaseId, Program.monitoredContainer);
ChangeFeedProcessor changeFeedProcessor = monitoredContainer
    .GetChangeFeedProcessorBuilder<ToDoItem>("changeFeedTime", Program.HandleChangesAsync)
        .WithInstanceName("consoleHost")
        .WithLeaseContainer(leaseContainer)
        .WithStartTime(particularPointInTime)
        .Build();
```

The change feed processor will be initialized for that specific date and time and start reading the changes that happened after.

Reading from the beginning

In other scenarios like data migrations or analyzing the entire history of a container, we need to read the change feed from **the beginning of that container's lifetime**. To do that, we can use `WithStartTime` on the builder extension, but passing `DateTime.MinValue.ToUniversalTime()`, which would generate the UTC representation of the minimum `DateTime` value, like so:

```
Container leaseContainer = client.GetContainer(databaseId, Program.leasesContainer);
Container monitoredContainer = client.GetContainer(databaseId, Program.monitoredContainer);
ChangeFeedProcessor changeFeedProcessor = monitoredContainer
    .GetChangeFeedProcessorBuilder<ToDoItem>("changeFeedBeginning", Program.HandleChangesAsync)
        .WithInstanceName("consoleHost")
        .WithLeaseContainer(leaseContainer)
        .WithStartTime(DateTime.MinValue.ToUniversalTime())
        .Build();
```

The change feed processor will be initialized and start reading changes from the beginning of the lifetime of the container.

> **NOTE**
> These customization options only work to setup the starting point in time of the change feed processor. Once the leases container is initialized for the first time, changing them has no effect.

Where to host the change feed processor

The change feed processor can be hosted in any platform that supports long running processes or tasks:

- A continuous running Azure WebJob.
- A process in an Azure Virtual Machine.
- A background job in Azure Kubernetes Service.
- An ASP.NET hosted service.

While change feed processor can run in short lived environments, because the lease container maintains the state, the startup cycle of these environments will add delay to receiving the notifications (due to the overhead of starting the processor every time the environment is started).

Additional resources

- Azure Cosmos DB SDK
- Complete sample application on GitHub
- Additional usage samples on GitHub
- Cosmos DB workshop labs for change feed processor

Next steps

You can now proceed to learn more about change feed processor in the following articles:

- Overview of change feed
- Change feed pull model
- How to migrate from the change feed processor library
- Using the change feed estimator

- Change feed processor start time

Serverless event-based architectures with Azure Cosmos DB and Azure Functions

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: SQL API

Azure Functions provides the simplest way to connect to the change feed. You can create small reactive Azure Functions that will be automatically triggered on each new event in your Azure Cosmos container's change feed.

With the Azure Functions trigger for Cosmos DB, you can leverage the Change Feed Processor's scaling and reliable event detection functionality without the need to maintain any worker infrastructure. Just focus on your Azure Function's logic without worrying about the rest of the event-sourcing pipeline. You can even mix the Trigger with any other Azure Functions bindings.

> **NOTE**
>
> Currently, the Azure Functions trigger for Cosmos DB is supported for use with the Core (SQL) API only.

Requirements

To implement a serverless event-based flow, you need:

- **The monitored container**: The monitored container is the Azure Cosmos container being monitored, and it stores the data from which the change feed is generated. Any inserts, updates to the monitored container are reflected in the change feed of the container.
- **The lease container**: The lease container maintains state across multiple and dynamic serverless Azure Function instances and enables dynamic scaling. This lease container can be manually or automatically created by the Azure Functions trigger for Cosmos DB. To automatically create the lease container, set the *CreateLeaseCollectionIfNotExists* flag in the configuration. Partitioned lease containers are required to have a `/id` partition key definition.

Create your Azure Functions trigger for Cosmos DB

Creating your Azure Function with an Azure Functions trigger for Cosmos DB is now supported across all Azure Functions IDE and CLI integrations:

- Visual Studio Extension for Visual Studio users.
- Visual Studio Code Extension for Visual Studio Code users.
- And finally Core CLI tooling for a cross-platform IDE agnostic experience.

Run your trigger locally

You can run your Azure Function locally with the Azure Cosmos DB Emulator to create and develop your serverless event-based flows without an Azure Subscription or incurring any costs.

If you want to test live scenarios in the cloud, you can Try Cosmos DB for free without any credit card or Azure subscription required.

Next steps

You can now continue to learn more about change feed in the following articles:

- Overview of change feed
- Ways to read change feed
- Using change feed processor library
- How to work with change feed processor library
- Serverless database computing using Azure Cosmos DB and Azure Functions

Change feed pull model in Azure Cosmos DB

12/4/2020 • 5 minutes to read • Edit Online

APPLIES TO: SQL API

With the change feed pull model, you can consume the Azure Cosmos DB change feed at your own pace. As you can already do with the change feed processor, you can use the change feed pull model to parallelize the processing of changes across multiple change feed consumers.

> **NOTE**
> The change feed pull model is currently in preview in the Azure Cosmos DB .NET SDK only. The preview is not yet available for other SDK versions.

Comparing with change feed processor

Many scenarios can process the change feed using either the change feed processor or the pull model. The pull model's continuation tokens and the change feed processor's lease container are both "bookmarks" for the last processed item (or batch of items) in the change feed.

However, you can't convert continuation tokens to a lease container (or vice versa).

> **NOTE**
> In most cases when you need to read from the change feed, the simplest option is to use the change feed processor.

You should consider using the pull model in these scenarios:

- Read changes from a particular partition key
- Control the pace at which your client receives changes for processing
- Perform a one-time read of the existing data in the change feed (for example, to do a data migration)

Here's some key differences between the change feed processor and pull model:

FEATURE	CHANGE FEED PROCESSOR	PULL MODEL
Keeping track of current point in processing change feed	Lease (stored in an Azure Cosmos DB container)	Continuation token (stored in memory or manually persisted)
Ability to replay past changes	Yes, with push model	Yes, with pull model
Polling for future changes	Automatically checks for changes based on user-specified `WithPollInterval`	Manual
Behavior where there are no new changes	Automatically wait `WithPollInterval` and recheck	Must catch exception and manually recheck
Process changes from entire container	Yes, and automatically parallelized across multiple threads/machine consuming from the same container	Yes, and manually parallelized using FeedTokens

FEATURE	CHANGE FEED PROCESSOR	PULL MODEL
Process changes from just a single partition key	Not supported	Yes
Support level	Generally available	Preview

> **NOTE**
> Unlike when reading using the change feed processor, you must explicitly handle cases where there no are no new changes.

Consuming an entire container's changes

You can create a `FeedIterator` to process the change feed using the pull model. When you initially create a `FeedIterator`, you must specify a required `ChangeFeedStartFrom` value which consists of both the starting position for reading changes as well as the desired `FeedRange`. The `FeedRange` is a range of partition key values and specifies the items that will be read from the change feed using that specific `FeedIterator`.

You can optionally specify `ChangeFeedRequestOptions` to set a `PageSizeHint`. The `PageSizeHint` is the maximum number of items that will be returned in a single page.

The `FeedIterator` comes in two flavors. In addition to the examples below that return entity objects, you can also obtain the response with `Stream` support. Streams allow you to read data without having it first deserialized, saving on client resources.

Here's an example for obtaining a `FeedIterator` that returns entity objects, in this case a `User` object:

```
FeedIterator<User> InteratorWithPOCOS = container.GetChangeFeedIterator<User>
(ChangeFeedStartFrom.Beginning());
```

Here's an example for obtaining a `FeedIterator` that returns a `Stream`:

```
FeedIterator iteratorWithStreams = container.GetChangeFeedStreamIterator<User>
(ChangeFeedStartFrom.Beginning());
```

If you don't supply a `FeedRange` to a `FeedIterator`, you can process an entire container's change feed at your own pace. Here's an example which starts reading all changes starting at the current time:

```
FeedIterator iteratorForTheEntireContainer = container.GetChangeFeedStreamIterator<User>
(ChangeFeedStartFrom.Now());

while (iteratorForTheEntireContainer.HasMoreResults)
{
    try {
        FeedResponse<User> users = await iteratorForTheEntireContainer.ReadNextAsync();

        foreach (User user in users)
            {
                Console.WriteLine($"Detected change for user with id {user.id}");
            }
    }
    catch {
        Console.WriteLine($"No new changes");
        Thread.Sleep(5000);
    }
}
```

Because the change feed is effectively an infinite list of items encompassing all future writes and updates, the value of `HasMoreResults` is always true. When you try to read the change feed and there are no new changes available, you'll receive an exception. In the above example, the exception is handled by waiting 5 seconds before rechecking for changes.

Consuming a partition key's changes

In some cases, you may only want to process a specific partition key's changes. You can obtain a `FeedIterator` for a specific partition key and process the changes the same way that you can for an entire container.

```
FeedIterator<User> iteratorForPartitionKey = container.GetChangeFeedIterator<User>
(ChangeFeedStartFrom.Beginning(FeedRange.FromPartitionKey(new PartitionKey("PartitionKeyValue"))));

while (iteratorForThePartitionKey.HasMoreResults)
{
    try {
        FeedResponse<User> users = await iteratorForThePartitionKey.ReadNextAsync();

        foreach (User user in users)
            {
                Console.WriteLine($"Detected change for user with id {user.id}");
            }
    }
    catch {
        Console.WriteLine($"No new changes");
        Thread.Sleep(5000);
    }
}
```

Using FeedRange for parallelization

In the change feed processor, work is automatically spread across multiple consumers. In the change feed pull model, you can use the `FeedRange` to parallelize the processing of the change feed. A `FeedRange` represents a range of partition key values.

Here's an example showing how to obtain a list of ranges for your container:

```
IReadOnlyList<FeedRange> ranges = await container.GetFeedRangesAsync();
```

When you obtain of list of FeedRanges for your container, you'll get one `FeedRange` per physical partition.

Using a `FeedRange`, you can then create a `FeedIterator` to parallelize the processing of the change feed across multiple machines or threads. Unlike the previous example that showed how to obtain a `FeedIterator` for the entire container or a single partition key, you can use FeedRanges to obtain multiple FeedIterators which can process the change feed in parallel.

In the case where you want to use FeedRanges, you need to have an orchestrator process that obtains FeedRanges and distributes them to those machines. This distribution could be:

- Using `FeedRange.ToJsonString` and distributing this string value. The consumers can use this value with `FeedRange.FromJsonString`
- If the distribution is in-process, passing the `FeedRange` object reference.

Here's a sample that shows how to read from the beginning of the container's change feed using two hypothetical separate machines that are reading in parallel:

Machine 1:

```
FeedIterator<User> iteratorA = container.GetChangeFeedIterator<User>
(ChangeFeedStartFrom.Beginning(ranges[0]));
while (iteratorA.HasMoreResults)
{
    try {
        FeedResponse<User> users = await iteratorA.ReadNextAsync();

        foreach (User user in users)
            {
                Console.WriteLine($"Detected change for user with id {user.id}");
            }
    }
    catch {
        Console.WriteLine($"No new changes");
        Thread.Sleep(5000);
    }
}
```

Machine 2:

```
FeedIterator<User> iteratorB = container.GetChangeFeedIterator<User>
(ChangeFeedStartFrom.Beginning(ranges[1]));
while (iteratorB.HasMoreResults)
{
    try {
        FeedResponse<User> users = await iteratorA.ReadNextAsync();

        foreach (User user in users)
            {
                Console.WriteLine($"Detected change for user with id {user.id}");
            }
    }
    catch {
        Console.WriteLine($"No new changes");
        Thread.Sleep(5000);
    }
}
```

Saving continuation tokens

You can save the position of your `FeedIterator` by creating a continuation token. A continuation token is a string value that keeps of track of your FeedIterator's last processed changes. This allows the `FeedIterator` to resume at this point later. The following code will read through the change feed since container creation. After no more

changes are available, it will persist a continuation token so that change feed consumption can be later resumed.

```
FeedIterator<User> iterator = container.GetChangeFeedIterator<User>(ChangeFeedStartFrom.Beginning());

string continuation = null;

while (iterator.HasMoreResults)
{
   try {
        FeedResponse<User> users = await iterator.ReadNextAsync();
        continuation = users.ContinuationToken;

        foreach (User user in users)
            {
                Console.WriteLine($"Detected change for user with id {user.id}");
            }
   }
    catch {
        Console.WriteLine($"No new changes");
        Thread.Sleep(5000);
    }
}

// Some time later
FeedIterator<User> iteratorThatResumesFromLastPoint = container.GetChangeFeedIterator<User>
(ChangeFeedStartFrom.ContinuationToken(continuation));
```

As long as the Cosmos container still exists, a FeedIterator's continuation token never expires.

Next steps

- Overview of change feed
- Using the change feed processor
- Trigger Azure Functions

Stored procedures, triggers, and user-defined functions

12/4/2020 • 8 minutes to read • Edit Online

APPLIES TO: SQL API

Azure Cosmos DB provides language-integrated, transactional execution of JavaScript. When using the SQL API in Azure Cosmos DB, you can write **stored procedures**, **triggers**, and **user-defined functions (UDFs)** in the JavaScript language. You can write your logic in JavaScript that executed inside the database engine. You can create and execute triggers, stored procedures, and UDFs by using Azure portal, the JavaScript language integrated query API in Azure Cosmos DB or the Cosmos DB SQL API client SDKs.

Benefits of using server-side programming

Writing stored procedures, triggers, and user-defined functions (UDFs) in JavaScript allows you to build rich applications and they have the following advantages:

- **Procedural logic:** JavaScript as a high-level programming language that provides rich and familiar interface to express business logic. You can perform a sequence of complex operations on the data.
- **Atomic transactions:** Azure Cosmos DB database operations that are performed within a single stored procedure or a trigger are atomic. This atomic functionality lets an application combine related operations into a single batch, so that either all of the operations succeed or none of them succeed.
- **Performance:** The JSON data is intrinsically mapped to the JavaScript language type system. This mapping allows for a number of optimizations like lazy materialization of JSON documents in the buffer pool and making them available on-demand to the executing code. There are other performance benefits associated with shipping business logic to the database, which includes:
  - *Batching:* You can group operations like inserts and submit them in bulk. The network traffic latency costs and the store overhead to create separate transactions are reduced significantly.
  - *Pre-compilation:* Stored procedures, triggers, and UDFs are implicitly pre-compiled to the byte code format in order to avoid compilation cost at the time of each script invocation. Due to pre-compilation, the invocation of stored procedures is fast and has a low footprint.
  - *Sequencing:* Sometimes operations need a triggering mechanism that may perform one or additional updates to the data. In addition to Atomicity, there are also performance benefits when executing on the server side.
- **Encapsulation:** Stored procedures can be used to group logic in one place. Encapsulation adds an abstraction layer on top of the data, which enables you to evolve your applications independently from the data. This layer of abstraction is helpful when the data is schema-less and you don't have to manage adding additional logic directly into your application. The abstraction lets your keep the data secure by streamlining the access from the scripts.

> **TIP**
> Stored procedures are best suited for operations that are write-heavy and require a transaction across a partition key value. When deciding whether to use stored procedures, optimize around encapsulating the maximum amount of writes possible. Generally speaking, stored procedures are not the most efficient means for doing large numbers of read or query operations, so using stored procedures to batch large numbers of reads to return to the client will not yield the desired benefit. For best performance, these read-heavy operations should be done on the client-side, using the Cosmos SDK.

Transactions

Transaction in a typical database can be defined as a sequence of operations performed as a single logical unit of work. Each transaction provides **ACID property guarantees**. ACID is a well-known acronym that stands for: **A**tomicity, **C**onsistency, **I**solation, and **D**urability.

- Atomicity guarantees that all the operations done inside a transaction are treated as a single unit, and either all of them are committed or none of them are.
- Consistency makes sure that the data is always in a valid state across transactions.
- Isolation guarantees that no two transactions interfere with each other – many commercial systems provide multiple isolation levels that can be used based on the application needs.
- Durability ensures that any change that is committed in a database will always be present.

In Azure Cosmos DB, JavaScript runtime is hosted inside the database engine. Hence, requests made within the stored procedures and the triggers execute in the same scope as the database session. This feature enables Azure Cosmos DB to guarantee ACID properties for all operations that are part of a stored procedure or a trigger. For examples, see how to implement transactions article.

Scope of a transaction

Stored procedures are associated with an Azure Cosmos container and stored procedure execution is scoped to a logical partition key. Stored procedures must include a logical partition key value during execution that defines the logical partition for the scope of the transaction. For more information, see Azure Cosmos DB partitioning article.

Commit and rollback

Transactions are natively integrated into the Azure Cosmos DB JavaScript programming model. Within a JavaScript function, all the operations are automatically wrapped under a single transaction. If the JavaScript logic in a stored procedure completes without any exceptions, all the operations within the transaction are committed to the database. Statements like `BEGIN TRANSACTION` and `COMMIT TRANSACTION` (familiar to relational databases) are implicit in Azure Cosmos DB. If there are any exceptions from the script, the Azure Cosmos DB JavaScript runtime will roll back the entire transaction. As such, throwing an exception is effectively equivalent to a `ROLLBACK TRANSACTION` in Azure Cosmos DB.

Data consistency

Stored procedures and triggers are always executed on the primary replica of an Azure Cosmos container. This feature ensures that reads from stored procedures offer strong consistency. Queries using user-defined functions can be executed on the primary or any secondary replica. Stored procedures and triggers are intended to support transactional writes – meanwhile read-only logic is best implemented as application-side logic and queries using the Azure Cosmos DB SQL API SDKs, will help you saturate the database throughput.

> **TIP**
> The queries executed within a stored procedure or trigger may not see changes to items made by the same script transaction. This statement applies both to SQL queries, such as `getContent().getCollection.queryDocuments()`, as well as integrated language queries, such as `getContext().getCollection().filter()`.

Bounded execution

All Azure Cosmos DB operations must complete within the specified timeout duration. This constraint applies to JavaScript functions - stored procedures, triggers, and user-defined functions. If an operation does not complete within that time limit, the transaction is rolled back.

You can either ensure that your JavaScript functions finish within the time limit or implement a continuation-based model to batch/resume execution. In order to simplify development of stored procedures and triggers to handle time limits, all functions under the Azure Cosmos container (for example, create, read, update, and delete of items) return a boolean value that represents whether that operation will complete. If this value is false, it is an indication that the procedure must wrap up execution because the script is consuming more time or provisioned throughput than the configured value. Operations queued prior to the first unaccepted store operation are guaranteed to complete if the stored procedure completes in time and does not queue any more requests. Thus, operations should be queued one at a time by using JavaScript's callback convention to manage the script's control flow. Because scripts are executed in a server-side environment, they are strictly governed. Scripts that repeatedly violate execution boundaries may be marked inactive and can't be executed, and they should be recreated to honor the execution boundaries.

JavaScript functions are also subject to provisioned throughput capacity. JavaScript functions could potentially end up using a large number of request units within a short time and may be rate-limited if the provisioned throughput capacity limit is reached. It is important to note that scripts consume additional throughput in addition to the throughput spent executing database operations, although these database operations are slightly less expensive than executing the same operations from the client.

Triggers

Azure Cosmos DB supports two types of triggers:

Pre-triggers

Azure Cosmos DB provides triggers that can be invoked by performing an operation on an Azure Cosmos item. For example, you can specify a pre-trigger when you are creating an item. In this case, the pre-trigger will run before the item is created. Pre-triggers cannot have any input parameters. If necessary, the request object can be used to update the document body from original request. When triggers are registered, users can specify the operations that it can run with. If a trigger was created with `TriggerOperation.Create`, this means using the trigger in a replace operation will not be permitted. For examples, see How to write triggers article.

Post-triggers

Similar to pre-triggers, post-triggers, are also associated with an operation on an Azure Cosmos item and they don't require any input parameters. They run *after* the operation has completed and have access to the response message that is sent to the client. For examples, see How to write triggers article.

> **NOTE**
> Registered triggers don't run automatically when their corresponding operations (create / delete / replace / update) happen. They have to be explicitly called when executing these operations. To learn more, see how to run triggers article.

User-defined functions

User-defined functions (UDFs) are used to extend the SQL API query language syntax and implement custom business logic easily. They can be called only within queries. UDFs do not have access to the context object and are meant to be used as compute only JavaScript. Therefore, UDFs can be run on secondary replicas. For examples, see How to write user-defined functions article.

JavaScript language-integrated query API

In addition to issuing queries using SQL API query syntax, the server-side SDK allows you to perform queries by using a JavaScript interface without any knowledge of SQL. The JavaScript query API allows you to programmatically build queries by passing predicate functions into sequence of function calls. Queries are parsed by the JavaScript runtime and are executed efficiently within Azure Cosmos DB. To learn about JavaScript query API support, see Working with JavaScript language integrated query API article. For examples, see How to write stored procedures and triggers using Javascript Query API article.

Next steps

Learn how to write and use stored procedures, triggers, and user-defined functions in Azure Cosmos DB with the following articles:

- How to write stored procedures, triggers, and user-defined functions
- How to use stored procedures, triggers, and user-defined functions
- Working with JavaScript language integrated query API

JavaScript query API in Azure Cosmos DB

12/4/2020 • 3 minutes to read • Edit Online

APPLIES TO: SQL API

In addition to issuing queries using the SQL API in Azure Cosmos DB, the Cosmos DB server-side SDK provides a JavaScript interface for performing optimized queries in Cosmos DB Stored Procedures and Triggers. You don't have to be aware of the SQL language to use this JavaScript interface. The JavaScript query API allows you to programmatically build queries by passing predicate functions into sequence of function calls, with a syntax familiar to ECMAScript5's array built-ins and popular JavaScript libraries like Lodash. Queries are parsed by the JavaScript runtime and efficiently executed using Azure Cosmos DB indices.

Supported JavaScript functions

FUNCTION	DESCRIPTION
`chain() ... .value([callback] [, options])`	Starts a chained call that must be terminated with value().
`filter(predicateFunction [, options] [, callback])`	Filters the input using a predicate function that returns true/false in order to filter in/out input documents into the resulting set. This function behaves similar to a WHERE clause in SQL.
`flatten([isShallow] [, options] [, callback])`	Combines and flattens arrays from each input item into a single array. This function behaves similar to SelectMany in LINQ.
`map(transformationFunction [, options] [, callback])`	Applies a projection given a transformation function that maps each input item to a JavaScript object or value. This function behaves similar to a SELECT clause in SQL.
`pluck([propertyName] [, options] [, callback])`	This function is a shortcut for a map that extracts the value of a single property from each input item.
`sortBy([predicate] [, options] [, callback])`	Produces a new set of documents by sorting the documents in the input document stream in ascending order by using the given predicate. This function behaves similar to an ORDER BY clause in SQL.
`sortByDescending([predicate] [, options] [, callback])`	Produces a new set of documents by sorting the documents in the input document stream in descending order using the given predicate. This function behaves similar to an ORDER BY x DESC clause in SQL.
`unwind(collectionSelector, [resultSelector], [options], [callback])`	Performs a self-join with inner array and adds results from both sides as tuples to the result projection. For instance, joining a person document with person.pets would produce [person, pet] tuples. This is similar to SelectMany in .NET LINK.

When included inside predicate and/or selector functions, the following JavaScript constructs get automatically optimized to run directly on Azure Cosmos DB indices:

- Simple operators: `=` `+` `-` `*` `/` `%` `|` `^` `&` `==` `!=` `===` `!===` `<` `>` `<=` `>=` `||` `&&` `<<` `>>` `>>>!` `~`

- Literals, including the object literal: {}
- var, return

The following JavaScript constructs do not get optimized for Azure Cosmos DB indices:

- Control flow (for example, if, for, while)
- Function calls

For more information, see the Cosmos DB Server Side JavaScript Documentation.

SQL to JavaScript cheat sheet

The following table presents various SQL queries and the corresponding JavaScript queries. As with SQL queries, properties (for example, item.id) are case-sensitive.

> **NOTE**
>
> `__` (double-underscore) is an alias to `getContext().getCollection()` when using the JavaScript query API.

SQL	JAVASCRIPT QUERY API	DESCRIPTION
SELECT * FROM docs	__.map(function(doc) { return doc; });	Results in all documents (paginated with continuation token) as is.

SELECT
  docs.id,
  docs.message AS msg,
  docs.actions
FROM docs
	__.map(function(doc) {
	  return {
	  id: doc.id,
	  msg: doc.message,
	  actions:doc.actions
	  };
	});
		Projects the id, message (aliased to msg), and action from all documents.

SELECT *
FROM docs
WHERE
  docs.id="X998_Y998"
	__.filter(function(doc) {
	  return doc.id ==="X998_Y998";
	});
		Queries for documents with the predicate: id = "X998_Y998".

SELECT *
FROM docs
WHERE
  ARRAY_CONTAINS(docs.Tags, 123)
	__.filter(function(x) {
	  return x.Tags && x.Tags.indexOf(123) > -1;
	});
		Queries for documents that have a Tags property and Tags is an array containing the value 123.

SELECT
  docs.id,
  docs.message AS msg
FROM docs
WHERE
  docs.id="X998_Y998"
	__.chain()
	  .filter(function(doc) {
	  return doc.id ==="X998_Y998";
	  })
	  .map(function(doc) {
	  return {
	  id: doc.id,
	  msg: doc.message
	  };
	  })
	.value();
		Queries for documents with a predicate, id = "X998_Y998", and then projects the id and message (aliased to msg).

SQL	JAVASCRIPT QUERY API	DESCRIPTION

SELECT VALUE tag
FROM docs
JOIN tag IN docs.Tags
ORDER BY docs._ts
	__.chain()
	.filter(function(doc) {
	return doc.Tags &&
	Array.isArray(doc.Tags);
	})
	.sortBy(function(doc) {
	return doc._ts;
	})
	.pluck("Tags")
	.flatten()
	.value()
		Filters for documents that have an array property, Tags, and sorts the resulting documents by the _ts timestamp system property, and then projects + flattens the Tags array.

Next steps

Learn more concepts and how-to write and use stored procedures, triggers, and user-defined functions in Azure Cosmos DB:

- How to write stored procedures and triggers using Javascript Query API
- Working with Azure Cosmos DB stored procedures, triggers and user-defined functions
- How to use stored procedures, triggers, user-defined functions in Azure Cosmos DB
- Azure Cosmos DB JavaScript server-side API reference
- JavaScript ES6 (ECMA 2015)

Secure access to data in Azure Cosmos DB

12/4/2020 • 8 minutes to read • Edit Online

APPLIES TO: SQL API

This article provides an overview of securing access to data stored in Microsoft Azure Cosmos DB.

Azure Cosmos DB uses two types of keys to authenticate users and provide access to its data and resources.

KEY TYPE	RESOURCES
Primary keys	Used for administrative resources: database accounts, databases, users, and permissions
Resource tokens	Used for application resources: containers, documents, attachments, stored procedures, triggers, and UDFs

Primary keys

Primary keys provide access to all the administrative resources for the database account. Each account consists of two primary keys: a primary key and secondary key. The purpose of dual keys is so that you can regenerate, or roll keys, providing continuous access to your account and data. To learn more about primary keys, see the Database security article.

Key rotation

The process of rotating your primary key is simple.

1. Navigate to the Azure portal to retrieve your secondary key.
2. Replace your primary key with your secondary key in your application. Make sure that all the Cosmos DB clients across all the deployments are promptly restarted and will start using the updated key.
3. Rotate the primary key in the Azure portal.
4. Validate the new primary key works against all resource. Key rotation process can take anywhere from less than a minute to hours depending on the size of the Cosmos DB account.
5. Replace the secondary key with the new primary key.

Code sample to use a primary key

The following code sample illustrates how to use a Cosmos DB account endpoint and primary key to instantiate a DocumentClient and create a database:

```
//Read the Azure Cosmos DB endpointUrl and authorization keys from config.
//These values are available from the Azure portal on the Azure Cosmos DB account blade under "Keys".
//Keep these values in a safe and secure location. Together they provide Administrative access to your Azure
Cosmos DB account.

private static readonly string endpointUrl = ConfigurationManager.AppSettings["EndPointUrl"];
private static readonly string authorizationKey = ConfigurationManager.AppSettings["AuthorizationKey"];

CosmosClient client = new CosmosClient(endpointUrl, authorizationKey);
```

The following code sample illustrates how to use the Azure Cosmos DB account endpoint and primary key to instantiate a `CosmosClient` object:

```
HOST = config.settings["host"]
MASTER_KEY = config.settings["master_key"]

DATABASE_ID = config.settings["database_id"]
CONTAINER_ID = config.settings["container_id"]
PARTITION_KEY = PartitionKey(path="/username")


# User that you want to give access to
USERNAME, USERNAME_2 = "user", "user2"

CONTAINER_ALL_PERMISSION = "CONTAINER_ALL_PERMISSION"
PARTITION_READ_PERMISSION = "PARTITION_READ_PERMISSION"
DOCUMENT_ALL_PERMISSION = "DOCUMENT_ALL_PERMISSION"


def create_user_if_not_exists(db, username):
    try:
        user = db.create_user(body={"id": username})
    except exceptions.CosmosResourceExistsError:
        user = db.get_user_client(username)

    return user


def create_permission_if_not_exists(user, permission_definition):
    try:
        permission = user.create_permission(permission_definition)
    except exceptions.CosmosResourceExistsError:
        permission = user.get_permission(permission_definition["id"])

    return permission


def token_client_upsert(container, username, item_id):
    try:
        container.upsert_item(
            {
                "id": item_id,
                "username": username,
                "msg": "This is a message for " + username,
            }
        )
    except exceptions.CosmosHttpResponseError:
        print("Error in upserting item with id '{0}'.".format(item_id))


def token_client_read_all(container):
    try:
        items = list(container.read_all_items())
        for i in items:
            print(i)
    except exceptions.CosmosResourceNotFoundError:
```

```
    except exceptions.CosmosResourceNotFoundError:
        print("Cannot read items--container '{0}' not found.".format(container.id))
    except exceptions.CosmosHttpResponseError:
        print("Error in reading items in container '{0}'.".format(container.id))


def token_client_read_item(container, username, item_id):
    try:
        item = container.read_item(item=item_id, partition_key=username)
        print(item)
    except exceptions.CosmosResourceNotFoundError:
        print("Cannot read--item with id '{0}' not found.".format(item_id))
    except exceptions.CosmosHttpResponseError:
        print("Error in reading item with id '{0}'.".format(item_id))


def token_client_delete(container, username, item_id):
    try:
        container.delete_item(item=item_id, partition_key=username)
    except exceptions.CosmosResourceNotFoundError:
        print("Cannot delete--item with id '{0}' not found.".format(item_id))
    except exceptions.CosmosHttpResponseError:
        print("Error in deleting item with id '{0}'.".format(item_id))


def token_client_query(container, username):
    try:
        for item in container.query_items(
            query="SELECT * FROM my_container c WHERE c.username=@username",
            parameters=[{"name": "@username", "value": username}],
            partition_key=username,
        ):
            print(json.dumps(item, indent=True))
    except exceptions.CosmosHttpResponseError:
        print("Error in querying item(s)")


def run_sample():
    client = cosmos_client.CosmosClient(HOST, {"masterKey": MASTER_KEY})
```

Resource tokens

Resource tokens provide access to the application resources within a database. Resource tokens:

- Provide access to specific containers, partition keys, documents, attachments, stored procedures, triggers, and UDFs.
- Are created when a user is granted permissions to a specific resource.
- Are recreated when a permission resource is acted upon on by POST, GET, or PUT call.
- Use a hash resource token specifically constructed for the user, resource, and permission.
- Are time bound with a customizable validity period. The default valid time span is one hour. Token lifetime, however, may be explicitly specified, up to a maximum of five hours.
- Provide a safe alternative to giving out the primary key.
- Enable clients to read, write, and delete resources in the Cosmos DB account according to the permissions they've been granted.

You can use a resource token (by creating Cosmos DB users and permissions) when you want to provide access to resources in your Cosmos DB account to a client that cannot be trusted with the primary key.

Cosmos DB resource tokens provide a safe alternative that enables clients to read, write, and delete resources in your Cosmos DB account according to the permissions you've granted, and without need for either a primary or read only key.

Here is a typical design pattern whereby resource tokens may be requested, generated, and delivered to clients:

1. A mid-tier service is set up to serve a mobile application to share user photos.
2. The mid-tier service possesses the primary key of the Cosmos DB account.
3. The photo app is installed on end-user mobile devices.
4. On login, the photo app establishes the identity of the user with the mid-tier service. This mechanism of identity establishment is purely up to the application.
5. Once the identity is established, the mid-tier service requests permissions based on the identity.
6. The mid-tier service sends a resource token back to the phone app.
7. The phone app can continue to use the resource token to directly access Cosmos DB resources with the permissions defined by the resource token and for the interval allowed by the resource token.
8. When the resource token expires, subsequent requests receive a 401 unauthorized exception. At this point, the phone app re-establishes the identity and requests a new resource token.

Resource token generation and management are handled by the native Cosmos DB client libraries; however, if you use REST you must construct the request/authentication headers. For more information on creating authentication headers for REST, see Access Control on Cosmos DB Resources or the source code for our .NET SDK or Node.js SDK.

For an example of a middle tier service used to generate or broker resource tokens, see the ResourceTokenBroker app.

Users

Azure Cosmos DB users are associated with a Cosmos database. Each database can contain zero or more Cosmos DB users. The following code sample shows how to create a Cosmos DB user using the Azure Cosmos DB .NET SDK v3.

```
//Create a user.
Database database = benchmark.client.GetDatabase("SalesDatabase");

User user = await database.CreateUserAsync("User 1");
```

> **NOTE**
>
> Each Cosmos DB user has a ReadAsync() method that can be used to retrieve the list of permissions associated with the user.

Permissions

A permission resource is associated with a user and assigned at the container as well as partition key level. Each user may contain zero or more permissions. A permission resource provides access to a security token that the user needs when trying to access a specific container or data in a specific partition key. There are two available access levels that may be provided by a permission resource:

- All: The user has full permission on the resource.
- Read: The user can only read the contents of the resource but cannot perform write, update, or delete operations on the resource.

> **NOTE**
> In order to run stored procedures the user must have the All permission on the container in which the stored procedure will be run.

If you enable the diagnostic logs on data-plane requests, the following two properties corresponding to the permission are logged:

- **resourceTokenPermissionId** - This property indicates the resource token permission Id that you have specified.
- **resourceTokenPermissionMode** - This property indicates the permission mode that you have set when creating the resource token. The permission mode can have values such as "all" or "read".

Code sample to create permission

The following code sample shows how to create a permission resource, read the resource token of the permission resource, and associate the permissions with the user created above.

```
// Create a permission on a container and specific partition key value
Container container = client.GetContainer("SalesDatabase", "OrdersContainer");
user.CreatePermissionAsync(
    new PermissionProperties(
        id: "permissionUser1Orders",
        permissionMode: PermissionMode.All,
        container: benchmark.container,
        resourcePartitionKey: new PartitionKey("012345")));
```

Code sample to read permission for user

The following code snippet shows how to retrieve the permission associated with the user created above and instantiate a new CosmosClient on behalf of the user, scoped to a single partition key.

```
//Read a permission, create user client session.
PermissionProperties permissionProperties = await user.GetPermission("permissionUser1Orders")

CosmosClient client = new CosmosClient(accountEndpoint: "MyEndpoint", authKeyOrResourceToken:
permissionProperties.Token);
```

Add users and assign roles

To add Azure Cosmos DB account reader access to your user account, have a subscription owner perform the following steps in the Azure portal.

1. Open the Azure portal, and select your Azure Cosmos DB account.
2. Click the **Access control (IAM)** tab, and then click **+ Add role assignment**.

3. In the **Add role assignment** pane, in the **Role** box, select **Cosmos DB Account Reader Role**.
4. In the **Assign access to box**, select **Azure AD user, group, or application**.
5. Select the user, group, or application in your directory to which you wish to grant access. You can search the directory by display name, email address, or object identifiers. The selected user, group, or application appears in the selected members list.
6. Click **Save**.

The entity can now read Azure Cosmos DB resources.

Delete or export user data

As a database service, Azure Cosmos DB enables you to search, select, modify and delete any data located in your database or containers. It is however your responsibility to use the provided APIs and define logic required to find and erase any personal data if needed. Each multi-model API (SQL, MongoDB, Gremlin, Cassandra, Table) provides different language SDKs that contain methods to search and delete data based on custom predicates. You can also enable the time to live (TTL) feature to delete data automatically after a specified period, without incurring any additional cost.

> **NOTE**
>
> For information about viewing or deleting personal data, see Azure Data Subject Requests for the GDPR. For more information about GDPR, see the GDPR section of the Service Trust portal.

Next steps

- To learn more about Cosmos database security, see Cosmos DB Database security.
- To learn how to construct Azure Cosmos DB authorization tokens, see Access Control on Azure Cosmos DB Resources.
- User management samples with users and permissions, .NET SDK v3 user management samples

Advanced Threat Protection for Azure Cosmos DB (Preview)

12/4/2020 • 3 minutes to read • Edit Online

APPLIES TO: SQL API

Advanced Threat Protection for Azure Cosmos DB provides an additional layer of security intelligence that detects unusual and potentially harmful attempts to access or exploit Azure Cosmos DB accounts. This layer of protection allows you to address threats, even without being a security expert, and integrate them with central security monitoring systems.

Security alerts are triggered when anomalies in activity occur. These security alerts are integrated with Azure Security Center, and are also sent via email to subscription administrators, with details of the suspicious activity and recommendations on how to investigate and remediate the threats.

> **NOTE**
>
> - Advanced Threat Protection for Azure Cosmos DB is currently available only for the SQL API.
> - Advanced Threat Protection for Azure Cosmos DB is currently not available in Azure government and sovereign cloud regions.

For a full investigation experience of the security alerts, we recommended enabling diagnostic logging in Azure Cosmos DB, which logs operations on the database itself, including CRUD operations on all documents, containers, and databases.

Threat types

Advanced Threat Protection for Azure Cosmos DB detects anomalous activities indicating unusual and potentially harmful attempts to access or exploit databases. It can currently trigger the following alerts:

- **Access from unusual locations**: This alert is triggered when there is a change in the access pattern to an Azure Cosmos account, where someone has connected to the Azure Cosmos DB endpoint from an unusual geographical location. In some cases, the alert detects a legitimate action, meaning a new application or developer's maintenance operation. In other cases, the alert detects a malicious action from a former employee, external attacker, etc.
- **Unusual data extraction**: This alert is triggered when a client is extracting an unusual amount of data from an Azure Cosmos DB account. This can be the symptom of some data exfiltration performed to transfer all the data stored in the account to an external data store.

Configure Advanced Threat Protection

You can configure advanced threat protection in any of several ways, described in the following sections.

- Portal
- REST API
- PowerShell
- ARM template
- Azure Policy

1. Launch the Azure portal at https://portal.azure.com.

2. From the Azure Cosmos DB account, from the **Settings** menu, select **Advanced security**.

3. In the **Advanced security** configuration blade:

   - Click the **Advanced Threat Protection** option to set it to **ON**.
   - Click **Save** to save the new or updated Advanced Threat Protection policy.

Azure Cosmos DB SQL SDK connectivity modes

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: SQL API

How a client connects to Azure Cosmos DB has important performance implications, especially for observed client-side latency. Azure Cosmos DB offers a simple, open RESTful programming model over HTTPS called gateway mode. Additionally, it offers an efficient TCP protocol, which is also RESTful in its communication model and uses TLS for initial authentication and encrypting traffic, called direct mode.

Available connectivity modes

The two available connectivity modes are:

- Gateway mode

  Gateway mode is supported on all SDK platforms. If your application runs within a corporate network with strict firewall restrictions, gateway mode is the best choice because it uses the standard HTTPS port and a single DNS endpoint. The performance tradeoff, however, is that gateway mode involves an additional network hop every time data is read from or written to Azure Cosmos DB. We also recommend gateway connection mode when you run applications in environments that have a limited number of socket connections.

  When you use the SDK in Azure Functions, particularly in the Consumption plan, be aware of the current limits on connections.

- Direct mode

  Direct mode supports connectivity through TCP protocol and offers better performance because there are fewer network hops. The application connects directly to the backend replicas. Direct mode is currently only supported on .NET and Java SDK platforms.

These connectivity modes essentially condition the route that data plane requests - document reads and writes - take from your client machine to partitions in the Azure Cosmos DB back-end. Direct mode is the preferred option

for best performance - it allows your client to open TCP connections directly to partitions in the Azure Cosmos DB back-end and send requests *directly* with no intermediary. By contrast, in Gateway mode, requests made by your client are routed to a so-called "Gateway" server in the Azure Cosmos DB front end, which in turn fans out your requests to the appropriate partition(s) in the Azure Cosmos DB back-end.

Service port ranges

When you use direct mode, in addition to the gateway ports, you need to ensure the port range between 10000 and 20000 is open because Azure Cosmos DB uses dynamic TCP ports. When using direct mode on private endpoints, the full range of TCP ports - from 0 to 65535 should be open. If these ports aren't open and you try to use the TCP protocol, you might receive a 503 Service Unavailable error.

The following table shows a summary of the connectivity modes available for various APIs and the service ports used for each API:

CONNECTION MODE	SUPPORTED PROTOCOL	SUPPORTED SDKS	API/SERVICE PORT
Gateway	HTTPS	All SDKs	SQL (443), MongoDB (10250, 10255, 10256), Table (443), Cassandra (10350), Graph (443) The port 10250 maps to a default Azure Cosmos DB API for MongoDB instance without geo-replication. Whereas the ports 10255 and 10256 map to the instance that has geo-replication.
Direct	TCP	.NET SDK Java SDK	When using public/service endpoints: ports in the 10000 through 20000 range When using private endpoints: ports in the 0 through 65535 range

Next steps

For specific SDK platform performance optimizations:

- .NET V2 SDK performance tips
- .NET V3 SDK performance tips
- Java V4 SDK performance tips

Manage an Azure Cosmos account

12/4/2020 • 7 minutes to read • Edit Online

APPLIES TO: SQL API

This article describes how to manage various tasks on an Azure Cosmos account using the Azure portal, Azure PowerShell, Azure CLI, and Azure Resource Manager templates.

Create an account

Azure portal

1. From the Azure portal menu or the **Home page**, select **Create a resource**.
2. On the **New** page, search for and select **Azure Cosmos DB**.
3. On the **Azure Cosmos DB** page, select **Create**.
4. On the **Create Azure Cosmos DB Account** page, enter the basic settings for the new Azure Cosmos account.

SETTING	VALUE	DESCRIPTION
Subscription	Subscription name	Select the Azure subscription that you want to use for this Azure Cosmos account.
Resource Group	Resource group name	Select a resource group, or select **Create new**, then enter a unique name for the new resource group.
Account Name	A unique name	Enter a name to identify your Azure Cosmos account. Because *documents.azure.com* is appended to the name that you provide to create your URI, use a unique name. The name can only contain lowercase letters, numbers, and the hyphen (-) character. It must be between 3-44 characters in length.
API	The type of account to create	Select **Core (SQL)** to create a document database and query by using SQL syntax. The API determines the type of account to create. Azure Cosmos DB provides five APIs: Core (SQL) and MongoDB for document data, Gremlin for graph data, Azure Table, and Cassandra. Currently, you must create a separate account for each API.

SETTING	VALUE	DESCRIPTION
Capacity mode	Provisioned throughput or Serverless	Select **Provisioned throughput** to create an account in provisioned throughput mode. Select **Serverless** to create an account in serverless mode.
Apply Free Tier Discount	Apply or Do not apply	With Azure Cosmos DB free tier, you will get the first 400 RU/s and 5 GB of storage for free in an account. Learn more about free tier.
Location	The region closest to your users	Select a geographic location to host your Azure Cosmos DB account. Use the location that is closest to your users to give them the fastest access to the data.
Account Type	Production or Non-Production	Select **Production** if the account will be used for a production workload. Select **Non-Production** if the account will be used for non-production, e.g. development, testing, QA, or staging. This is an Azure resource tag setting that tunes the Portal experience but does not affect the underlying Azure Cosmos DB account. You can change this value anytime.

NOTE

You can have up to one free tier Azure Cosmos DB account per Azure subscription and must opt-in when creating the account. If you do not see the option to apply the free tier discount, this means another account in the subscription has already been enabled with free tier.

NOTE

The following options are not available if you select **Serverless** as the **Capacity mode**:

- Apply Free Tier Discount
- Geo-redundancy
- Multi-region Writes

5. Select **Review + create**. You can skip the **Network** and **Tags** sections.

6. Review the account settings, and then select **Create**. It takes a few minutes to create the account. Wait for the portal page to display **Your deployment is complete**.

7. Select **Go to resource** to go to the Azure Cosmos DB account page.

Azure CLI

Please see Create an Azure Cosmos DB account with Azure CLI

Azure PowerShell

Please see Create an Azure Cosmos DB account with PowerShell

Azure Resource Manager template

Please see Create Azure Cosmos DB account with Azure Resource Manager templates

Add/remove regions from your database account

Azure portal

1. Sign in to Azure portal.
2. Go to your Azure Cosmos account, and open the **Replicate data globally** menu.
3. To add regions, select the hexagons on the map with the **+** label that corresponds to your desired region(s). Alternatively, to add a region, select the **+ Add region** option and choose a region from the drop-down menu.
4. To remove regions, clear one or more regions from the map by selecting the blue hexagons with check marks. Or select the "wastebasket" ( ) icon next to the region on the right side.
5. To save your changes, select **OK**.

In a single-region write mode, you cannot remove the write region. You must fail over to a different region before you can delete the current write region.

In a multi-region write mode, you can add or remove any region, if you have at least one region.

Azure CLI

Please see Add or remove regions with Azure CLI

Azure PowerShell

Please see Add or remove regions with PowerShell

Configure multiple write-regions

Azure portal

Open the **Replicate Data Globally** tab and select **Enable** to enable multi-region writes. After you enable multi-region writes, all the read regions that you currently have on the account will become read and write regions.

Azure CLI

Please see Enable multiple-write regions with Azure CLI

Azure PowerShell

Please see Enable multiple-write regions with PowerShell

Resource Manager template

An account can be migrated from single write region to multiple write regions by deploying the Resource Manager template used to create the account and setting `enableMultipleWriteLocations: true`. The following Azure Resource Manager template is a bare minimum template that will deploy an Azure Cosmos account for SQL API with two regions and multiple write locations enabled.

```
{
    "$schema": "https://schema.management.azure.com/schemas/2015-01-01/deploymentTemplate.json#",
    "contentVersion": "1.0.0.0",
    "parameters": {
        "name": {
            "type": "String"
        },
        "location": {
            "type": "String",
            "defaultValue": "[resourceGroup().location]"
        },
        "primaryRegion":{
            "type":"string",
            "metadata": {
                "description": "The primary replica region for the Cosmos DB account."
            }
        },
        "secondaryRegion":{
            "type":"string",
            "metadata": {
              "description": "The secondary replica region for the Cosmos DB account."
          }
        }
    },
    "resources": [
        {
            "type": "Microsoft.DocumentDb/databaseAccounts",
            "kind": "GlobalDocumentDB",
            "name": "[parameters('name')]",
            "apiVersion": "2019-08-01",
            "location": "[parameters('location')]",
            "tags": {},
            "properties": {
                "databaseAccountOfferType": "Standard",
                "consistencyPolicy": { "defaultConsistencyLevel": "Session" },
                "locations":
                [
                    {
                        "locationName": "[parameters('primaryRegion')]",
                        "failoverPriority": 0,
                        "isZoneRedundant": false
                    },
                    {
                        "locationName": "[parameters('secondaryRegion')]",
                        "failoverPriority": 1,
                        "isZoneRedundant": false
                    }
                ],
                "enableMultipleWriteLocations": true
            }
        }
    ]
}
```

Enable automatic failover for your Azure Cosmos account

The Automatic failover option allows Azure Cosmos DB to failover to the region with the highest failover priority with no user action should a region become unavailable. When automatic failover is enabled, region priority can be modified. Account must have two or more regions to enable automatic failover.

Azure portal

1. From your Azure Cosmos account, open the **Replicate data globally** pane.

2. At the top of the pane, select **Automatic Failover**.

3. On the **Automatic Failover** pane, make sure that **Enable Automatic Failover** is set to **ON**.

4. Select **Save**.

Azure CLI

Please see Enable automatic failover with Azure CLI

Azure PowerShell

Please see Enable automatic failover with PowerShell

Set failover priorities for your Azure Cosmos account

After a Cosmos account is configured for automatic failover, the failover priority for regions can be changed.

> **IMPORTANT**
>
> You cannot modify the write region (failover priority of zero) when the account is configured for automatic failover. To change the write region, you must disable automatic failover and do a manual failover.

Azure portal

1. From your Azure Cosmos account, open the **Replicate data globally** pane.
2. At the top of the pane, select **Automatic Failover**.

3. On the **Automatic Failover** pane, make sure that **Enable Automatic Failover** is set to **ON**.
4. To modify the failover priority, drag the read regions via the three dots on the left side of the row that appear when you hover over them.
5. Select **Save**.

Azure CLI

Please see Set failover priority with Azure CLI

Azure PowerShell

Please see Set failover priority with PowerShell

Perform manual failover on an Azure Cosmos account

> **IMPORTANT**
>
> The Azure Cosmos account must be configured for manual failover for this operation to succeed.

The process for performing a manual failover involves changing the account's write region (failover priority = 0) to another region configured for the account.

> **NOTE**
>
> Accounts with multiple write regions cannot be manually failed over. For applications using the Azure Cosmos SDK, the SDK will detect when a region becomes unavailable, then redirect automatically to the next closest region if using multi-homing API in the SDK.

Azure portal

1. Go to your Azure Cosmos account, and open the **Replicate data globally** menu.
2. At the top of the menu, select **Manual Failover**.

3. On the **Manual Failover** menu, select your new write region. Select the check box to indicate that you understand this option changes your write region.
4. To trigger the failover, select **OK**.

Azure CLI

Please see Trigger manual failover with Azure CLI

Azure PowerShell

Please see Trigger manual failover with PowerShell

Next steps

For more information and examples on how to manage the Azure Cosmos account as well as database and containers, read the following articles:

- Manage Azure Cosmos DB using Azure PowerShell
- Manage Azure Cosmos DB using Azure CLI

Manage Azure Cosmos DB Core (SQL) API resources using PowerShell

12/4/2020 • 15 minutes to read • Edit Online

APPLIES TO: SQL API

The following guide describes how to use PowerShell to script and automate management of Azure Cosmos DB Core (SQL) API resources, including the Cosmos account, database, container, and throughput. For PowerShell cmdlets for other APIs see PowerShell Samples for Cassandra, PowerShell Samples for MongoDB API, PowerShell Samples for Gremlin, PowerShell Samples for Table

> **NOTE**
> Samples in this article use Az.CosmosDB management cmdlets. See the Az.CosmosDB API reference page for the latest changes.

For cross-platform management of Azure Cosmos DB, you can use the `Az` and `Az.CosmosDB` cmdlets with cross-platform PowerShell, as well as the Azure CLI, the REST API, or the Azure portal.

> **NOTE**
> This article has been updated to use the new Azure PowerShell Az module. You can still use the AzureRM module, which will continue to receive bug fixes until at least December 2020. To learn more about the new Az module and AzureRM compatibility, see Introducing the new Azure PowerShell Az module. For Az module installation instructions, see Install Azure PowerShell.

Getting Started

Follow the instructions in How to install and configure Azure PowerShell to install and sign in to your Azure account in PowerShell.

> **IMPORTANT**
> Azure Cosmos DB resources cannot be renamed as this violates how Azure Resource Manager works with resource URIs.

Azure Cosmos accounts

The following sections demonstrate how to manage the Azure Cosmos account, including:

- Create an Azure Cosmos account
- Update an Azure Cosmos account
- List all Azure Cosmos accounts in a subscription
- Get an Azure Cosmos account
- Delete an Azure Cosmos account
- Update tags for an Azure Cosmos account
- List keys for an Azure Cosmos account
- Regenerate keys for an Azure Cosmos account

- List connection strings for an Azure Cosmos account
- Modify failover priority for an Azure Cosmos account
- Trigger a manual failover for an Azure Cosmos account
- List resource locks on an Azure Cosmos DB account

Create an Azure Cosmos account

This command creates an Azure Cosmos DB database account with multiple regions, automatic failover and bounded-staleness consistency policy.

```
$resourceGroupName = "myResourceGroup"
$accountName = "mycosmosaccount"
$apiKind = "Sql"
$consistencyLevel = "BoundedStaleness"
$maxStalenessInterval = 300
$maxStalenessPrefix = 100000
$locations = @()
$locations += New-AzCosmosDBLocationObject -LocationName "East US" -FailoverPriority 0 -IsZoneRedundant 0
$locations += New-AzCosmosDBLocationObject -LocationName "West US" -FailoverPriority 1 -IsZoneRedundant 0

New-AzCosmosDBAccount `
    -ResourceGroupName $resourceGroupName `
    -LocationObject $locations `
    -Name $accountName `
    -ApiKind $apiKind `
    -EnableAutomaticFailover:$true `
    -DefaultConsistencyLevel $consistencyLevel `
    -MaxStalenessIntervalInSeconds $maxStalenessInterval `
    -MaxStalenessPrefix $maxStalenessPrefix
```

- `$resourceGroupName` The Azure resource group into which to deploy the Cosmos account. It must already exist.
- `$locations` The regions for the database account, the region with `FailoverPriority 0` is the write region.
- `$accountName` The name for the Azure Cosmos account. Must be unique, lowercase, include only alphanumeric and '-' characters, and between 3 and 31 characters in length.
- `$apiKind` The type of Cosmos account to create. For more information, see APIs in Cosmos DB.
- `$consistencyPolicy`, `$maxStalenessInterval`, and `$maxStalenessPrefix` The default consistency level and settings of the Azure Cosmos account. For more information, see Consistency Levels in Azure Cosmos DB.

Azure Cosmos accounts can be configured with IP Firewall, Virtual Network service endpoints, and private endpoints. For information on how to configure the IP Firewall for Azure Cosmos DB, see Configure IP Firewall. For information on how to enable service endpoints for Azure Cosmos DB, see Configure access from virtual Networks. For information on how to enable private endpoints for Azure Cosmos DB, see Configure access from private endpoints.

List all Azure Cosmos accounts in a Resource Group

This command lists all Azure Cosmos accounts in a Resource Group.

```
$resourceGroupName = "myResourceGroup"

Get-AzCosmosDBAccount -ResourceGroupName $resourceGroupName
```

Get the properties of an Azure Cosmos account

This command allows you to get the properties of an existing Azure Cosmos account.

```
$resourceGroupName = "myResourceGroup"
$accountName = "mycosmosaccount"

Get-AzCosmosDBAccount -ResourceGroupName $resourceGroupName -Name $accountName
```

Update an Azure Cosmos account

This command allows you to update your Azure Cosmos DB database account properties. Properties that can be updated include the following:

- Adding or removing regions
- Changing default consistency policy
- Changing IP Range Filter
- Changing Virtual Network configurations
- Enabling multi-region writes

> **NOTE**
> You cannot simultaneously add or remove regions (`locations`) and change other properties for an Azure Cosmos account. Modifying regions must be performed as a separate operation from any other change to the account.

> **NOTE**
> This command allows you to add and remove regions but does not allow you to modify failover priorities or trigger a manual failover. See Modify failover priority and Trigger manual failover.

```
# Create account with two regions
$resourceGroupName = "myResourceGroup"
$accountName = "mycosmosaccount"
$apiKind = "Sql"
$consistencyLevel = "Session"
$enableAutomaticFailover = $true
$locations = @()
$locations += New-AzCosmosDBLocationObject -LocationName "East US" -FailoverPriority 0 -IsZoneRedundant 0
$locations += New-AzCosmosDBLocationObject -LocationName "West US" -FailoverPriority 1 -IsZoneRedundant 0

# Create the Cosmos DB account
New-AzCosmosDBAccount `
    -ResourceGroupName $resourceGroupName `
    -LocationObject $locations `
    -Name $accountName `
    -ApiKind $apiKind `
    -EnableAutomaticFailover:$enableAutomaticFailover `
    -DefaultConsistencyLevel $consistencyLevel

# Add a region to the account
$locationObject2 = @()
$locationObject2 += New-AzCosmosDBLocationObject -LocationName "East US" -FailoverPriority 0 -IsZoneRedundant
0
$locationObject2 += New-AzCosmosDBLocationObject -LocationName "West US" -FailoverPriority 1 -IsZoneRedundant
0
$locationObject2 += New-AzCosmosDBLocationObject -LocationName "South Central US" -FailoverPriority 2 -
IsZoneRedundant 0

Update-AzCosmosDBAccountRegion `
    -ResourceGroupName $resourceGroupName `
    -Name $accountName `
    -LocationObject $locationObject2

Write-Host "Update-AzCosmosDBAccountRegion returns before the region update is complete."
Write-Host "Check account in Azure portal or using Get-AzCosmosDBAccount for region status."
Write-Host "When region was added, press any key to continue."
$HOST.UI.RawUI.ReadKey("NoEcho,IncludeKeyDown") | OUT-NULL
$HOST.UI.RawUI.Flushinputbuffer()

# Remove West US region from the account
$locationObject3 = @()
$locationObject3 += New-AzCosmosDBLocationObject -LocationName "East US" -FailoverPriority 0 -IsZoneRedundant
0
$locationObject3 += New-AzCosmosDBLocationObject -LocationName "South Central US" -FailoverPriority 1 -
IsZoneRedundant 0

Update-AzCosmosDBAccountRegion `
    -ResourceGroupName $resourceGroupName `
    -Name $accountName `
    -LocationObject $locationObject3

Write-Host "Update-AzCosmosDBAccountRegion returns before the region update is complete."
Write-Host "Check account in Azure portal or using Get-AzCosmosDBAccount for region status."
```

Enable multiple write regions for an Azure Cosmos account

```
$resourceGroupName = "myResourceGroup"
$accountName = "mycosmosaccount"
$enableAutomaticFailover = $false
$enableMultiMaster = $true

# First disable automatic failover - cannot have both automatic
# failover and multi-region writes on an account
Update-AzCosmosDBAccount `
    -ResourceGroupName $resourceGroupName `
    -Name $accountName `
    -EnableAutomaticFailover:$enableAutomaticFailover

# Now enable multi-region writes
Update-AzCosmosDBAccount `
    -ResourceGroupName $resourceGroupName `
    -Name $accountName `
    -EnableMultipleWriteLocations:$enableMultiMaster
```

Delete an Azure Cosmos account

This command deletes an existing Azure Cosmos account.

```
$resourceGroupName = "myResourceGroup"
$accountName = "mycosmosaccount"

Remove-AzCosmosDBAccount `
    -ResourceGroupName $resourceGroupName `
    -Name $accountName `
    -PassThru:$true
```

Update Tags of an Azure Cosmos account

This command sets the Azure resource tags for an Azure Cosmos account. Tags can be set both at account creation using `New-AzCosmosDBAccount` as well as on account update using `Update-AzCosmosDBAccount`.

```
$resourceGroupName = "myResourceGroup"
$accountName = "mycosmosaccount"
$tags = @{dept = "Finance"; environment = "Production";}

Update-AzCosmosDBAccount `
    -ResourceGroupName $resourceGroupName `
    -Name $accountName `
    -Tag $tags
```

List Account Keys

When you create an Azure Cosmos account, the service generates two primary access keys that can be used for authentication when the Azure Cosmos account is accessed. Read-only keys for authenticating read-only operations are also generated. By providing two access keys, Azure Cosmos DB enables you to regenerate and rotate one key at a time with no interruption to your Azure Cosmos account. Cosmos DB accounts have two read-write keys (primary and secondary) and two read-only keys (primary and secondary).

```
$resourceGroupName = "myResourceGroup"
$accountName = "mycosmosaccount"

Get-AzCosmosDBAccountKey `
    -ResourceGroupName $resourceGroupName `
    -Name $accountName `
    -Type "Keys"
```

List Connection Strings

The following command retrieves connection strings to connect apps to the Cosmos DB account.

```
$resourceGroupName = "myResourceGroup"
$accountName = "mycosmosaccount"

Get-AzCosmosDBAccountKey `
    -ResourceGroupName $resourceGroupName `
    -Name $accountName `
    -Type "ConnectionStrings"
```

Regenerate Account Keys

Access keys to an Azure Cosmos account should be periodically regenerated to help keep connections secure. A primary and secondary access keys are assigned to the account. This allows clients to maintain access while one key at a time is regenerated. There are four types of keys for an Azure Cosmos account (Primary, Secondary, PrimaryReadonly, and SecondaryReadonly)

```
$resourceGroupName = "myResourceGroup" # Resource Group must already exist
$accountName = "mycosmosaccount" # Must be all lower case
$keyKind = "primary" # Other key kinds: secondary, primaryReadOnly, secondaryReadOnly

New-AzCosmosDBAccountKey `
    -ResourceGroupName $resourceGroupName `
    -Name $accountName `
    -KeyKind $keyKind
```

Enable automatic failover

The following command sets a Cosmos DB account to fail over automatically to its secondary region should the primary region become unavailable.

```
$resourceGroupName = "myResourceGroup"
$accountName = "mycosmosaccount"
$enableAutomaticFailover = $true
$enableMultiMaster = $false

# First disable multi-region writes - cannot have both automatic
# failover and multi-region writes on an account
Update-AzCosmosDBAccount `
    -ResourceGroupName $resourceGroupName `
    -Name $accountName `
    -EnableMultipleWriteLocations:$enableMultiMaster

# Now enable automatic failover
Update-AzCosmosDBAccount `
    -ResourceGroupName $resourceGroupName `
    -Name $accountName `
    -EnableAutomaticFailover:$enableAutomaticFailover
```

Modify Failover Priority

For accounts configured with Automatic Failover, you can change the order in which Cosmos will promote secondary replicas to primary should the primary become unavailable.

For the example below, assume the current failover priority is `West US 2 = 0`, `East US 2 = 1`, `South Central US = 2`. The command will change this to `West US 2 = 0`, `South Central US = 1`, `East US 2 = 2`.

Caution

Changing the location for `failoverPriority=0` will trigger a manual failover for an Azure Cosmos account. Any other priority changes will not trigger a failover.

```
$resourceGroupName = "myResourceGroup"
$accountName = "mycosmosaccount"
$locations = @("West US 2", "South Central US", "East US 2") # Regions ordered by UPDATED failover priority

Update-AzCosmosDBAccountFailoverPriority `
    -ResourceGroupName $resourceGroupName `
    -Name $accountName `
    -FailoverPolicy $locations
```

Trigger Manual Failover

For accounts configured with Manual Failover, you can fail over and promote any secondary replica to primary by modifying to `failoverPriority=0`. This operation can be used to initiate a disaster recovery drill to test disaster recovery planning.

For the example below, assume the account has a current failover priority of `West US 2 = 0` and `East US 2 = 1` and flip the regions.

Caution

Changing `locationName` for `failoverPriority=0` will trigger a manual failover for an Azure Cosmos account. Any other priority change will not trigger a failover.

```
$resourceGroupName = "myResourceGroup"
$accountName = "mycosmosaccount"
$locations = @("East US 2", "West US 2") # Regions ordered by UPDATED failover priority

Update-AzCosmosDBAccountFailoverPriority `
    -ResourceGroupName $resourceGroupName `
    -Name $accountName `
    -FailoverPolicy $locations
```

List resource locks on an Azure Cosmos DB account

Resource locks can be placed on Azure Cosmos DB resources including databases and collections. The example below shows how to list all Azure resource locks on an Azure Cosmos DB account.

```
$resourceGroupName = "myResourceGroup"
$resourceTypeAccount = "Microsoft.DocumentDB/databaseAccounts"
$accountName = "mycosmosaccount"

Get-AzResourceLock `
    -ResourceGroupName $resourceGroupName `
    -ResourceType $resourceTypeAccount `
    -ResourceName $accountName
```

Azure Cosmos DB Database

The following sections demonstrate how to manage the Azure Cosmos DB database, including:

- Create an Azure Cosmos DB database
- Create an Azure Cosmos DB database with shared throughput
- Get the throughput of an Azure Cosmos DB database
- Migrate database throughput to autoscale
- List all Azure Cosmos DB databases in an account
- Get a single Azure Cosmos DB database
- Delete an Azure Cosmos DB database
- Create a resource lock on an Azure Cosmos DB database to prevent delete

- Remove a resource lock on an Azure Cosmos DB database

Create an Azure Cosmos DB database

```
$resourceGroupName = "myResourceGroup"
$accountName = "mycosmosaccount"
$databaseName = "myDatabase"

New-AzCosmosDBSqlDatabase `
    -ResourceGroupName $resourceGroupName `
    -AccountName $accountName `
    -Name $databaseName
```

Create an Azure Cosmos DB database with shared throughput

```
$resourceGroupName = "myResourceGroup"
$accountName = "mycosmosaccount"
$databaseName = "myDatabase"
$databaseRUs = 400

New-AzCosmosDBSqlDatabase `
    -ResourceGroupName $resourceGroupName `
    -AccountName $accountName `
    -Name $databaseName `
    -Throughput $databaseRUs
```

Get the throughput of an Azure Cosmos DB database

```
$resourceGroupName = "myResourceGroup"
$accountName = "mycosmosaccount"
$databaseName = "myDatabase"

Get-AzCosmosDBSqlDatabaseThroughput `
    -ResourceGroupName $resourceGroupName `
    -AccountName $accountName `
    -Name $databaseName
```

Migrate database throughput to autoscale

```
$resourceGroupName = "myResourceGroup"
$accountName = "mycosmosaccount"
$databaseName = "myDatabase"

Invoke-AzCosmosDBSqlDatabaseThroughputMigration `
    -ResourceGroupName $resourceGroupName `
    -AccountName $accountName `
    -Name $databaseName `
    -ThroughputType Autoscale
```

Get all Azure Cosmos DB databases in an account

```
$resourceGroupName = "myResourceGroup"
$accountName = "mycosmosaccount"

Get-AzCosmosDBSqlDatabase `
    -ResourceGroupName $resourceGroupName `
    -AccountName $accountName
```

Get a single Azure Cosmos DB database

```
$resourceGroupName = "myResourceGroup"
$accountName = "mycosmosaccount"
$databaseName = "myDatabase"

Get-AzCosmosDBSqlDatabase `
    -ResourceGroupName $resourceGroupName `
    -AccountName $accountName `
    -Name $databaseName
```

Delete an Azure Cosmos DB database

```
$resourceGroupName = "myResourceGroup"
$accountName = "mycosmosaccount"
$databaseName = "myDatabase"

Remove-AzCosmosDBSqlDatabase `
    -ResourceGroupName $resourceGroupName `
    -AccountName $accountName `
    -Name $databaseName
```

Create a resource lock on an Azure Cosmos DB database to prevent delete

```
$resourceGroupName = "myResourceGroup"
$resourceType = "Microsoft.DocumentDB/databaseAccounts/sqlDatabases"
$accountName = "mycosmosaccount"
$databaseName = "myDatabase"
$resourceName = "$accountName/$databaseName"
$lockName = "myResourceLock"
$lockLevel = "CanNotDelete"

New-AzResourceLock `
    -ResourceGroupName $resourceGroupName `
    -ResourceType $resourceType `
    -ResourceName $resourceName `
    -LockName $lockName `
    -LockLevel $lockLevel
```

Remove a resource lock on an Azure Cosmos DB database

```
$resourceGroupName = "myResourceGroup"
$resourceType = "Microsoft.DocumentDB/databaseAccounts/sqlDatabases"
$accountName = "mycosmosaccount"
$databaseName = "myDatabase"
$resourceName = "$accountName/$databaseName"
$lockName = "myResourceLock"

Remove-AzResourceLock `
    -ResourceGroupName $resourceGroupName `
    -ResourceType $resourceType `
    -ResourceName $resourceName `
    -LockName $lockName
```

Azure Cosmos DB Container

The following sections demonstrate how to manage the Azure Cosmos DB container, including:

- Create an Azure Cosmos DB container
- Create an Azure Cosmos DB container with autoscale
- Create an Azure Cosmos DB container with a large partition key

- Get the throughput of an Azure Cosmos DB container
- Migrate container throughput to autoscale
- Create an Azure Cosmos DB container with custom indexing
- Create an Azure Cosmos DB container with indexing turned off
- Create an Azure Cosmos DB container with unique key and TTL
- Create an Azure Cosmos DB container with conflict resolution
- List all Azure Cosmos DB containers in a database
- Get a single Azure Cosmos DB container in a database
- Delete an Azure Cosmos DB container
- Create a resource lock on an Azure Cosmos DB container to prevent delete
- Remove a resource lock on an Azure Cosmos DB container

Create an Azure Cosmos DB container

```
# Create an Azure Cosmos DB container with default indexes and throughput at 400 RU
$resourceGroupName = "myResourceGroup"
$accountName = "mycosmosaccount"
$databaseName = "myDatabase"
$containerName = "myContainer"
$partitionKeyPath = "/myPartitionKey"
$throughput = 400 #minimum = 400

New-AzCosmosDBSqlContainer `
    -ResourceGroupName $resourceGroupName `
    -AccountName $accountName `
    -DatabaseName $databaseName `
    -Name $containerName `
    -PartitionKeyKind Hash `
    -PartitionKeyPath $partitionKeyPath `
    -Throughput $throughput
```

Create an Azure Cosmos DB container with autoscale

```
# Create an Azure Cosmos DB container with default indexes and autoscale throughput at 4000 RU
$resourceGroupName = "myResourceGroup"
$accountName = "mycosmosaccount"
$databaseName = "myDatabase"
$containerName = "myContainer"
$partitionKeyPath = "/myPartitionKey"
$autoscaleMaxThroughput = 4000 #minimum = 4000

New-AzCosmosDBSqlContainer `
    -ResourceGroupName $resourceGroupName `
    -AccountName $accountName `
    -DatabaseName $databaseName `
    -Name $containerName `
    -PartitionKeyKind Hash `
    -PartitionKeyPath $partitionKeyPath `
    -AutoscaleMaxThroughput $autoscaleMaxThroughput
```

Create an Azure Cosmos DB container with a large partition key size

```
# Create an Azure Cosmos DB container with a large partition key value (version = 2)
$resourceGroupName = "myResourceGroup"
$accountName = "mycosmosaccount"
$databaseName = "myDatabase"
$containerName = "myContainer"
$partitionKeyPath = "/myPartitionKey"

New-AzCosmosDBSqlContainer `
    -ResourceGroupName $resourceGroupName `
    -AccountName $accountName `
    -DatabaseName $databaseName `
    -Name $containerName `
    -PartitionKeyKind Hash `
    -PartitionKeyPath $partitionKeyPath `
    -PartitionKeyVersion 2
```

Get the throughput of an Azure Cosmos DB container

```
$resourceGroupName = "myResourceGroup"
$accountName = "mycosmosaccount"
$databaseName = "myDatabase"
$containerName = "myContainer"

Get-AzCosmosDBSqlContainerThroughput `
    -ResourceGroupName $resourceGroupName `
    -AccountName $accountName `
    -DatabaseName $databaseName `
    -Name $containerName
```

Migrate container throughput to autoscale

```
$resourceGroupName = "myResourceGroup"
$accountName = "mycosmosaccount"
$databaseName = "myDatabase"
$containerName = "myContainer"

Invoke-AzCosmosDBSqlContainerThroughputMigration `
    -ResourceGroupName $resourceGroupName `
    -AccountName $accountName `
    -DatabaseName $databaseName `
    -Name $containerName `
    -ThroughputType Autoscale
```

Create an Azure Cosmos DB container with custom index policy

```
# Create a container with a custom indexing policy
$resourceGroupName = "myResourceGroup"
$accountName = "mycosmosaccount"
$databaseName = "myDatabase"
$containerName = "myContainer"
$partitionKeyPath = "/myPartitionKey"
$indexPathIncluded = "/*"
$indexPathExcluded = "/myExcludedPath/*"

$includedPathIndex = New-AzCosmosDBSqlIncludedPathIndex -DataType String -Kind Range
$includedPath = New-AzCosmosDBSqlIncludedPath -Path $indexPathIncluded -Index $includedPathIndex

$indexingPolicy = New-AzCosmosDBSqlIndexingPolicy `
    -IncludedPath $includedPath `
    -ExcludedPath $indexPathExcluded `
    -IndexingMode Consistent `
    -Automatic $true

New-AzCosmosDBSqlContainer `
    -ResourceGroupName $resourceGroupName `
    -AccountName $accountName `
    -DatabaseName $databaseName `
    -Name $containerName `
    -PartitionKeyKind Hash `
    -PartitionKeyPath $partitionKeyPath `
    -IndexingPolicy $indexingPolicy
```

Create an Azure Cosmos DB container with indexing turned off

```
# Create an Azure Cosmos DB container with no indexing
$resourceGroupName = "myResourceGroup"
$accountName = "mycosmosaccount"
$databaseName = "myDatabase"
$containerName = "myContainer"
$partitionKeyPath = "/myPartitionKey"

$indexingPolicy = New-AzCosmosDBSqlIndexingPolicy `
    -IndexingMode None

New-AzCosmosDBSqlContainer `
    -ResourceGroupName $resourceGroupName `
    -AccountName $accountName `
    -DatabaseName $databaseName `
    -Name $containerName `
    -PartitionKeyKind Hash `
    -PartitionKeyPath $partitionKeyPath `
    -IndexingPolicy $indexingPolicy
```

Create an Azure Cosmos DB container with unique key policy and TTL

```
# Create a container with a unique key policy and TTL of one day
$resourceGroupName = "myResourceGroup"
$accountName = "mycosmosaccount"
$databaseName = "myDatabase"
$containerName = "myContainer"
$partitionKeyPath = "/myPartitionKey"
$uniqueKeyPath = "/myUniqueKeyPath"
$ttlInSeconds = 86400 # Set this to -1 (or don't use it at all) to never expire

$uniqueKey = New-AzCosmosDBSqlUniqueKey `
    -Path $uniqueKeyPath

$uniqueKeyPolicy = New-AzCosmosDBSqlUniqueKeyPolicy `
    -UniqueKey $uniqueKey

New-AzCosmosDBSqlContainer `
    -ResourceGroupName $resourceGroupName `
    -AccountName $accountName `
    -DatabaseName $databaseName `
    -Name $containerName `
    -PartitionKeyKind Hash `
    -PartitionKeyPath $partitionKeyPath `
    -UniqueKeyPolicy $uniqueKeyPolicy `
    -TtlInSeconds $ttlInSeconds
```

Create an Azure Cosmos DB container with conflict resolution

To write all conflicts to the ConflictsFeed and handle separately, pass `-Type "Custom" -Path ""`.

```
# Create container with last-writer-wins conflict resolution policy
$resourceGroupName = "myResourceGroup"
$accountName = "mycosmosaccount"
$databaseName = "myDatabase"
$containerName = "myContainer"
$partitionKeyPath = "/myPartitionKey"
$conflictResolutionPath = "/myResolutionPath"

$conflictResolutionPolicy = New-AzCosmosDBSqlConflictResolutionPolicy `
    -Type LastWriterWins `
    -Path $conflictResolutionPath

New-AzCosmosDBSqlContainer `
    -ResourceGroupName $resourceGroupName `
    -AccountName $accountName `
    -DatabaseName $databaseName `
    -Name $containerName `
    -PartitionKeyKind Hash `
    -PartitionKeyPath $partitionKeyPath `
    -ConflictResolutionPolicy $conflictResolutionPolicy
```

To create a conflict resolution policy to use a stored procedure, call `New-AzCosmosDBSqlConflictResolutionPolicy` and pass parameters `-Type` and `-ConflictResolutionProcedure`.

```
# Create container with custom conflict resolution policy using a stored procedure
$resourceGroupName = "myResourceGroup"
$accountName = "mycosmosaccount"
$databaseName = "myDatabase"
$containerName = "myContainer"
$partitionKeyPath = "/myPartitionKey"
$conflictResolutionSprocName = "mysproc"

$conflictResolutionSproc = "/dbs/$databaseName/colls/$containerName/sprocs/$conflictResolutionSprocName"

$conflictResolutionPolicy = New-AzCosmosDBSqlConflictResolutionPolicy `
    -Type Custom `
    -ConflictResolutionProcedure $conflictResolutionSproc

New-AzCosmosDBSqlContainer `
    -ResourceGroupName $resourceGroupName `
    -AccountName $accountName `
    -DatabaseName $databaseName `
    -Name $containerName `
    -PartitionKeyKind Hash `
    -PartitionKeyPath $partitionKeyPath `
    -ConflictResolutionPolicy $conflictResolutionPolicy
```

List all Azure Cosmos DB containers in a database

```
# List all Azure Cosmos DB containers in a database
$resourceGroupName = "myResourceGroup"
$accountName = "mycosmosaccount"
$databaseName = "myDatabase"

Get-AzCosmosDBSqlContainer `
    -ResourceGroupName $resourceGroupName `
    -AccountName $accountName `
    -DatabaseName $databaseName
```

Get a single Azure Cosmos DB container in a database

```
# Get a single Azure Cosmos DB container in a database
$resourceGroupName = "myResourceGroup"
$accountName = "mycosmosaccount"
$databaseName = "myDatabase"
$containerName = "myContainer"

Get-AzCosmosDBSqlContainer `
    -ResourceGroupName $resourceGroupName `
    -AccountName $accountName `
    -DatabaseName $databaseName `
    -Name $containerName
```

Delete an Azure Cosmos DB container

```
# Delete an Azure Cosmos DB container
$resourceGroupName = "myResourceGroup"
$accountName = "mycosmosaccount"
$databaseName = "myDatabase"
$containerName = "myContainer"

Remove-AzCosmosDBSqlContainer `
    -ResourceGroupName $resourceGroupName `
    -AccountName $accountName `
    -DatabaseName $databaseName `
    -Name $containerName
```

Create a resource lock on an Azure Cosmos DB container to prevent delete

```
$resourceGroupName = "myResourceGroup"
$resourceType = "Microsoft.DocumentDB/databaseAccounts/sqlDatabases/containers"
$accountName = "mycosmosaccount"
$databaseName = "myDatabase"
$containerName = "myContainer"
$resourceName = "$accountName/$databaseName/$containerName"
$lockName = "myResourceLock"
$lockLevel = "CanNotDelete"

New-AzResourceLock `
    -ResourceGroupName $resourceGroupName `
    -ResourceType $resourceType `
    -ResourceName $resourceName `
    -LockName $lockName `
    -LockLevel $lockLevel
```

Remove a resource lock on an Azure Cosmos DB container

```
$resourceGroupName = "myResourceGroup"
$resourceType = "Microsoft.DocumentDB/databaseAccounts/sqlDatabases/containers"
$accountName = "mycosmosaccount"
$databaseName = "myDatabase"
$containerName = "myContainer"
$resourceName = "$accountName/$databaseName/$containerName"
$lockName = "myResourceLock"

Remove-AzResourceLock `
    -ResourceGroupName $resourceGroupName `
    -ResourceType $resourceType `
    -ResourceName $resourceName `
    -LockName $lockName
```

Next steps

- All PowerShell Samples
- How to manage Azure Cosmos account
- Create an Azure Cosmos DB container
- Configure time-to-live in Azure Cosmos DB

Manage Azure Cosmos Core (SQL) API resources using Azure CLI

12/4/2020 • 10 minutes to read • Edit Online

APPLIES TO: SQL API

The following guide describes common commands to automate management of your Azure Cosmos DB accounts, databases and containers using Azure CLI. Reference pages for all Azure Cosmos DB CLI commands are available in the Azure CLI Reference. You can also find more examples in Azure CLI samples for Azure Cosmos DB, including how to create and manage Cosmos DB accounts, databases and containers for MongoDB, Gremlin, Cassandra and Table API.

Prerequisites

- Use Azure Cloud Shell using the bash environment.

  Launch Cloud Shell

- If you prefer, install Azure CLI to run CLI reference commands.
  - If you're using a local install, sign in with Azure CLI by using the az login command. To finish the authentication process, follow the steps displayed in your terminal. See Sign in with Azure CLI for additional sign-in options.
  - When you're prompted, install Azure CLI extensions on first use. For more information about extensions, see Use extensions with Azure CLI.
  - Run az version to find the version and dependent libraries that are installed. To upgrade to the latest version, run az upgrade.
- This article requires version 2.12.1 or later of the Azure CLI. If using Azure Cloud Shell, the latest version is already installed.

For Azure CLI samples for other APIs see CLI Samples for Cassandra, CLI Samples for MongoDB API, CLI Samples for Gremlin, CLI Samples for Table

> **IMPORTANT**
>
> Azure Cosmos DB resources cannot be renamed as this violates how Azure Resource Manager works with resource URIs.

Azure Cosmos Accounts

The following sections demonstrate how to manage the Azure Cosmos account, including:

- Create an Azure Cosmos account
- Add or remove regions
- Enable multi-region writes
- Set regional failover priority
- Enable automatic failover
- Trigger manual failover
- List account keys

- List read-only account keys
- List connection strings
- Regenerate account key

Create an Azure Cosmos DB account

Create an Azure Cosmos DB account with SQL API, Session consistency in West US 2 and East US 2 regions:

> **IMPORTANT**
>
> The Azure Cosmos account name must be lowercase and less than 44 characters.

```
resourceGroupName='MyResourceGroup'
accountName='mycosmosaccount' #needs to be lower case and less than 44 characters

az cosmosdb create \
    -n $accountName \
    -g $resourceGroupName \
    --default-consistency-level Session \
    --locations regionName='West US 2' failoverPriority=0 isZoneRedundant=False \
    --locations regionName='East US 2' failoverPriority=1 isZoneRedundant=False
```

Add or remove regions

Create an Azure Cosmos account with two regions, add a region, and remove a region.

> **NOTE**
>
> You cannot simultaneously add or remove regions `locations` and change other properties for an Azure Cosmos account. Modifying regions must be performed as a separate operation than any other change to the account resource.

> **NOTE**
>
> This command allows you to add and remove regions but does not allow you to modify failover priorities or trigger a manual failover. See Set failover priority and Trigger manual failover.

```
resourceGroupName='myResourceGroup'
accountName='mycosmosaccount'

# Create an account with 2 regions
az cosmosdb create --name $accountName --resource-group $resourceGroupName \
    --locations regionName="West US 2" failoverPriority=0 isZoneRedundant=False \
    --locations regionName="East US 2" failoverPriority=1 isZoneRedundant=False

# Add a region
az cosmosdb update --name $accountName --resource-group $resourceGroupName \
    --locations regionName="West US 2" failoverPriority=0 isZoneRedundant=False \
    --locations regionName="East US 2" failoverPriority=1 isZoneRedundant=False \
    --locations regionName="South Central US" failoverPriority=2 isZoneRedundant=False

# Remove a region
az cosmosdb update --name $accountName --resource-group $resourceGroupName \
    --locations regionName="West US 2" failoverPriority=0 isZoneRedundant=False \
    --locations regionName="East US 2" failoverPriority=1 isZoneRedundant=False
```

Enable multiple write regions

Enable multi-region writes for a Cosmos account

```
# Update an Azure Cosmos account from single write region to multiple write regions
resourceGroupName='myResourceGroup'
accountName='mycosmosaccount'

# Get the account resource id for an existing account
accountId=$(az cosmosdb show -g $resourceGroupName -n $accountName --query id -o tsv)

az cosmosdb update --ids $accountId --enable-multiple-write-locations true
```

Set failover priority

Set the failover priority for an Azure Cosmos account configured for automatic failover

```
# Assume region order is initially 'West US 2'=0 'East US 2'=1 'South Central US'=2 for account
resourceGroupName='myResourceGroup'
accountName='mycosmosaccount'

# Get the account resource id for an existing account
accountId=$(az cosmosdb show -g $resourceGroupName -n $accountName --query id -o tsv)

# Make South Central US the next region to fail over to instead of East US 2
az cosmosdb failover-priority-change --ids $accountId \
    --failover-policies 'West US 2=0' 'South Central US=1' 'East US 2=2'
```

Enable automatic failover

```
# Enable automatic failover on an existing account
resourceGroupName='myResourceGroup'
accountName='mycosmosaccount'

# Get the account resource id for an existing account
accountId=$(az cosmosdb show -g $resourceGroupName -n $accountName --query id -o tsv)

az cosmosdb update --ids $accountId --enable-automatic-failover true
```

Trigger manual failover

Caution

Changing region with priority = 0 will trigger a manual failover for an Azure Cosmos account. Any other priority change will not trigger a failover.

```
# Assume region order is initially 'West US 2=0' 'East US 2=1' 'South Central US=2' for account
resourceGroupName='myResourceGroup'
accountName='mycosmosaccount'

# Get the account resource id for an existing account
accountId=$(az cosmosdb show -g $resourceGroupName -n $accountName --query id -o tsv)

# Trigger a manual failover to promote East US 2 as new write region
az cosmosdb failover-priority-change --ids $accountId \
    --failover-policies 'East US 2=0' 'South Central US=1' 'West US 2=2'
```

List all account keys

Get all keys for a Cosmos account.

```
# List all account keys
resourceGroupName='MyResourceGroup'
accountName='mycosmosaccount'

az cosmosdb keys list \
   -n $accountName \
   -g $resourceGroupName
```

List read-only account keys

Get read-only keys for a Cosmos account.

```
# List read-only account keys
resourceGroupName='MyResourceGroup'
accountName='mycosmosaccount'

az cosmosdb keys list \
    -n $accountName \
    -g $resourceGroupName \
    --type read-only-keys
```

List connection strings

Get the connection strings for a Cosmos account.

```
# List connection strings
resourceGroupName='MyResourceGroup'
accountName='mycosmosaccount'

az cosmosdb keys list \
    -n $accountName \
    -g $resourceGroupName \
    --type connection-strings
```

Regenerate account key

Regenerate a new key for a Cosmos account.

```
# Regenerate secondary account keys
# key-kind values: primary, primaryReadonly, secondary, secondaryReadonly
az cosmosdb keys regenerate \
    -n $accountName \
    -g $resourceGroupName \
    --key-kind secondary
```

Azure Cosmos DB database

The following sections demonstrate how to manage the Azure Cosmos DB database, including:

- Create a database
- Create a database with shared throughput
- Migrate a database to autoscale throughput
- Change database throughput
- Prevent a database from being deleted

Create a database

Create a Cosmos database.

```
resourceGroupName='MyResourceGroup'
accountName='mycosmosaccount'
databaseName='database1'

az cosmosdb sql database create \
    -a $accountName \
    -g $resourceGroupName \
    -n $databaseName
```

Create a database with shared throughput

Create a Cosmos database with shared throughput.

```
resourceGroupName='MyResourceGroup'
accountName='mycosmosaccount'
databaseName='database1'
throughput=400

az cosmosdb sql database create \
    -a $accountName \
    -g $resourceGroupName \
    -n $databaseName \
    --throughput $throughput
```

Migrate a database to autoscale throughput

```
resourceGroupName='MyResourceGroup'
accountName='mycosmosaccount'
databaseName='database1'

# Migrate to autoscale throughput
az cosmosdb sql database throughput migrate \
    -a $accountName \
    -g $resourceGroupName \
    -n $databaseName \
    -t 'autoscale'

# Read the new autoscale max throughput
az cosmosdb sql database throughput show \
    -g $resourceGroupName \
    -a $accountName \
    -n $databaseName \
    --query resource.autoscaleSettings.maxThroughput \
    -o tsv
```

Change database throughput

Increase the throughput of a Cosmos database by 1000 RU/s.

```
resourceGroupName='MyResourceGroup'
accountName='mycosmosaccount'
databaseName='database1'
newRU=1000

# Get minimum throughput to make sure newRU is not lower than minRU
minRU=$(az cosmosdb sql database throughput show \
    -g $resourceGroupName -a $accountName -n $databaseName \
    --query resource.minimumThroughput -o tsv)

if [ $minRU -gt $newRU ]; then
    newRU=$minRU
fi

az cosmosdb sql database throughput update \
    -a $accountName \
    -g $resourceGroupName \
    -n $databaseName \
    --throughput $newRU
```

Prevent a database from being deleted

Put an Azure resource delete lock on a database to prevent it from being deleted. This feature requires locking the Cosmos account from being changed by data plane SDKs. To learn more see, Preventing changes from SDKs. Azure resource locks can also prevent a resource from being changed by specifying a `ReadOnly` lock type. For a Cosmos database, it can be used to prevent throughput from being changed.

```
resourceGroupName='myResourceGroup'
accountName='mycosmosaccount'
databaseName='database1'

lockType='CanNotDelete' # CanNotDelete or ReadOnly
databaseParent="databaseAccounts/$accountName"
databaseLockName="$databaseName-Lock"

# Create a delete lock on database
az lock create --name $databaseLockName \
    --resource-group $resourceGroupName \
    --resource-type Microsoft.DocumentDB/sqlDatabases \
    --lock-type $lockType \
    --parent $databaseParent \
    --resource $databaseName

# Delete lock on database
lockid=$(az lock show --name $databaseLockName \
        --resource-group $resourceGroupName \
        --resource-type Microsoft.DocumentDB/sqlDatabases \
        --resource $databaseName \
        --parent $databaseParent \
        --output tsv --query id)
az lock delete --ids $lockid
```

Azure Cosmos DB container

The following sections demonstrate how to manage the Azure Cosmos DB container, including:

- Create a container
- Create a container with autoscale
- Create a container with TTL enabled
- Create a container with custom index policy
- Change container throughput

- Migrate a container to autoscale throughput
- Prevent a container from being deleted

Create a container

Create a Cosmos container with default index policy, partition key and RU/s of 400.

```
# Create a SQL API container
resourceGroupName='MyResourceGroup'
accountName='mycosmosaccount'
databaseName='database1'
containerName='container1'
partitionKey='/myPartitionKey'
throughput=400

az cosmosdb sql container create \
    -a $accountName -g $resourceGroupName \
    -d $databaseName -n $containerName \
    -p $partitionKey --throughput $throughput
```

Create a container with autoscale

Create a Cosmos container with default index policy, partition key and autoscale RU/s of 4000.

```
# Create a SQL API container
resourceGroupName='MyResourceGroup'
accountName='mycosmosaccount'
databaseName='database1'
containerName='container1'
partitionKey='/myPartitionKey'
maxThroughput=4000

az cosmosdb sql container create \
    -a $accountName -g $resourceGroupName \
    -d $databaseName -n $containerName \
    -p $partitionKey --max-throughput $maxThroughput
```

Create a container with TTL

Create a Cosmos container with TTL enabled.

```
# Create an Azure Cosmos container with TTL of one day
resourceGroupName='myResourceGroup'
accountName='mycosmosaccount'
databaseName='database1'
containerName='container1'

az cosmosdb sql container update \
    -g $resourceGroupName \
    -a $accountName \
    -d $databaseName \
    -n $containerName \
    --ttl=86400
```

Create a container with a custom index policy

Create a Cosmos container with a custom index policy, a spatial index, composite index, a partition key and RU/s of 400.

```
# Create a SQL API container
resourceGroupName='MyResourceGroup'
accountName='mycosmosaccount'
databaseName='database1'
containerName='container1'
partitionKey='/myPartitionKey'
throughput=400

# Generate a unique 10 character alphanumeric string to ensure unique resource names
uniqueId=$(env LC_CTYPE=C tr -dc 'a-z0-9' < /dev/urandom | fold -w 10 | head -n 1)

# Define the index policy for the container, include spatial and composite indexes
idxpolicy=$(cat << EOF
{
    "indexingMode": "consistent",
    "includedPaths": [
        {"path": "/*"}
    ],
    "excludedPaths": [
        { "path": "/headquarters/employees/?"}
    ],
    "spatialIndexes": [
        {"path": "/*", "types": ["Point"]}
    ],
    "compositeIndexes":[
        [
            { "path":"/name", "order":"ascending" },
            { "path":"/age", "order":"descending" }
        ]
    ]
}
EOF
)
# Persist index policy to json file
echo "$idxpolicy" > "idxpolicy-$uniqueId.json"


az cosmosdb sql container create \
    -a $accountName -g $resourceGroupName \
    -d $databaseName -n $containerName \
    -p $partitionKey --throughput $throughput \
    --idx @idxpolicy-$uniqueId.json

# Clean up temporary index policy file
rm -f "idxpolicy-$uniqueId.json"
```

Change container throughput

Increase the throughput of a Cosmos container by 1000 RU/s.

```
resourceGroupName='MyResourceGroup'
accountName='mycosmosaccount'
databaseName='database1'
containerName='container1'
newRU=1000

# Get minimum throughput to make sure newRU is not lower than minRU
minRU=$(az cosmosdb sql container throughput show \
    -g $resourceGroupName -a $accountName -d $databaseName \
    -n $containerName --query resource.minimumThroughput -o tsv)

if [ $minRU -gt $newRU ]; then
    newRU=$minRU
fi

az cosmosdb sql container throughput update \
    -a $accountName \
    -g $resourceGroupName \
    -d $databaseName \
    -n $containerName \
    --throughput $newRU
```

Migrate a container to autoscale throughput

```
resourceGroupName='MyResourceGroup'
accountName='mycosmosaccount'
databaseName='database1'
containerName='container1'

# Migrate to autoscale throughput
az cosmosdb sql container throughput migrate \
    -a $accountName \
    -g $resourceGroupName \
    -d $databaseName \
    -n $containerName \
    -t 'autoscale'

# Read the new autoscale max throughput
az cosmosdb sql container throughput show \
    -g $resourceGroupName \
    -a $accountName \
    -d $databaseName \
    -n $containerName \
    --query resource.autoscaleSettings.maxThroughput \
    -o tsv
```

Prevent a container from being deleted

Put an Azure resource delete lock on a container to prevent it from being deleted. This feature requires locking the Cosmos account from being changed by data plane SDKs. To learn more see, Preventing changes from SDKs. Azure resource locks can also prevent a resource from being changed by specifying a `ReadOnly` lock type. For a Cosmos container, this can be used to prevent throughput or any other property from being changed.

```
resourceGroupName='myResourceGroup'
accountName='mycosmosaccount'
databaseName='database1'
containerName='container1'

lockType='CanNotDelete' # CanNotDelete or ReadOnly
databaseParent="databaseAccounts/$accountName"
containerParent="databaseAccounts/$accountName/sqlDatabases/$databaseName"
containerLockName="$containerName-Lock"

# Create a delete lock on container
az lock create --name $containerLockName \
    --resource-group $resourceGroupName \
    --resource-type Microsoft.DocumentDB/containers \
    --lock-type $lockType \
    --parent $containerParent \
    --resource $containerName

# Delete lock on container
lockid=$(az lock show --name $containerLockName \
        --resource-group $resourceGroupName \
        --resource-type Microsoft.DocumentDB/containers \
        --resource-name $containerName \
        --parent $containerParent \
        --output tsv --query id)
az lock delete --ids $lockid
```

Next steps

For more information on the Azure CLI, see:

- Install Azure CLI
- Azure CLI Reference
- Additional Azure CLI samples for Azure Cosmos DB

Manage Azure Cosmos DB Core (SQL) API resources with Azure Resource Manager templates

12/4/2020 • 10 minutes to read • Edit Online

APPLIES TO: SQL API

In this article, you learn how to use Azure Resource Manager templates to help deploy and manage your Azure Cosmos DB accounts, databases, and containers.

This article only shows Azure Resource Manager template examples for Core (SQL) API accounts. You can also find template examples for Cassandra, Gremlin, MongoDB, and Table APIs.

IMPORTANT

- Account names are limited to 44 characters, all lowercase.
- To change the throughput values, redeploy the template with updated RU/s.
- When you add or remove locations to an Azure Cosmos account, you can't simultaneously modify other properties. These operations must be done separately.
- Azure Cosmos DB resources cannot be renamed as this violates how Azure Resource Manager works with resource URIs.

To create any of the Azure Cosmos DB resources below, copy the following example template into a new json file. You can optionally create a parameters json file to use when deploying multiple instances of the same resource with different names and values. There are many ways to deploy Azure Resource Manager templates including, Azure portal, Azure CLI, Azure PowerShell and GitHub.

Azure Cosmos account with autoscale throughput

This template creates an Azure Cosmos account in two regions with options for consistency and failover, with database and container configured for autoscale throughput that has most policy options enabled. This template is also available for one-click deploy from Azure Quickstart Templates Gallery.

Deploy to Azure

```
{
    "$schema": "https://schema.management.azure.com/schemas/2019-04-01/deploymentTemplate.json#",
    "contentVersion": "1.0.0.0",
    "parameters": {
        "accountName": {
            "type": "string",
            "defaultValue": "[concat('sql-', uniqueString(resourceGroup().id))]",
            "metadata": {
                "description": "Cosmos DB account name, max length 44 characters, lowercase"
            }
        },
        "location": {
            "type": "string",
            "defaultValue": "[resourceGroup().location]",
            "metadata": {
                "description": "Location for the Cosmos DB account."
            }
        },
        "primaryRegion":{
            "type":"string",
```

```
            "metadata": {
                "description": "The primary replica region for the Cosmos DB account."
            }
        },
        "secondaryRegion":{
            "type":"string",
            "metadata": {
              "description": "The secondary replica region for the Cosmos DB account."
          }
        },
        "defaultConsistencyLevel": {
            "type": "string",
            "defaultValue": "Session",
            "allowedValues": [
                "Eventual",
                "ConsistentPrefix",
                "Session",
                "BoundedStaleness",
                "Strong"
            ],
            "metadata": {
                "description": "The default consistency level of the Cosmos DB account."
            }
        },
        "maxStalenessPrefix": {
            "type": "int",
            "minValue": 10,
            "defaultValue": 100000,
            "maxValue": 2147483647,
            "metadata": {
                "description": "Max stale requests. Required for BoundedStaleness. Valid ranges, Single
Region: 10 to 1000000. Multi Region: 100000 to 1000000."
            }
        },
        "maxIntervalInSeconds": {
            "type": "int",
            "minValue": 5,
            "defaultValue": 300,
            "maxValue": 86400,
            "metadata": {
                "description": "Max lag time (minutes). Required for BoundedStaleness. Valid ranges, Single
Region: 5 to 84600. Multi Region: 300 to 86400."
            }
        },
        "automaticFailover": {
            "type": "bool",
            "defaultValue": true,
            "allowedValues": [
                true,
                false
            ],
            "metadata": {
                "description": "Enable automatic failover for regions"
            }
        },
        "databaseName": {
            "type": "string",
            "metadata": {
                "description": "The name for the database"
            }
        },
        "containerName": {
            "type": "string",
            "metadata": {
                "description": "The name for the container"
            }
        },
        "autoscaleMaxThroughput": {
            "type": "int",
```

```
            "defaultValue": 4000,
            "minValue": 4000,
            "maxValue": 1000000,
            "metadata": {
                "description": "Maximum throughput for the container"
            }
        }
    },
    "variables": {
        "accountName": "[toLower(parameters('accountName'))]",
        "consistencyPolicy": {
            "Eventual": {
                "defaultConsistencyLevel": "Eventual"
            },
            "ConsistentPrefix": {
                "defaultConsistencyLevel": "ConsistentPrefix"
            },
            "Session": {
                "defaultConsistencyLevel": "Session"
            },
            "BoundedStaleness": {
                "defaultConsistencyLevel": "BoundedStaleness",
                "maxStalenessPrefix": "[parameters('maxStalenessPrefix')]",
                "maxIntervalInSeconds": "[parameters('maxIntervalInSeconds')]"
            },
            "Strong": {
                "defaultConsistencyLevel": "Strong"
            }
        },
        "locations": [
            {
                "locationName": "[parameters('primaryRegion')]",
                "failoverPriority": 0,
                "isZoneRedundant": false
            },
            {
                "locationName": "[parameters('secondaryRegion')]",
                "failoverPriority": 1,
                "isZoneRedundant": false
            }
        ]
    },
    "resources": [
        {
            "type": "Microsoft.DocumentDB/databaseAccounts",
            "name": "[variables('accountName')]",
            "apiVersion": "2020-04-01",
            "kind": "GlobalDocumentDB",
            "location": "[parameters('location')]",
            "properties": {
                "consistencyPolicy": "[variables('consistencyPolicy')
[parameters('defaultConsistencyLevel')]]",
                "locations": "[variables('locations')]",
                "databaseAccountOfferType": "Standard",
                "enableAutomaticFailover": "[parameters('automaticFailover')]"
            }
        },
        {
            "type": "Microsoft.DocumentDB/databaseAccounts/sqlDatabases",
            "name": "[concat(variables('accountName'), '/', parameters('databaseName'))]",
            "apiVersion": "2020-04-01",
            "dependsOn": [
                "[resourceId('Microsoft.DocumentDB/databaseAccounts', variables('accountName'))]"
            ],
            "properties":{
                "resource":{
                    "id": "[parameters('databaseName')]"
                }
            }
```

```
        },
        {
            "type": "Microsoft.DocumentDB/databaseAccounts/sqlDatabases/containers",
            "name": "[concat(variables('accountName'), '/', parameters('databaseName'), '/',
parameters('containerName'))]",
            "apiVersion": "2020-04-01",
            "dependsOn": [
                "[resourceId('Microsoft.DocumentDB/databaseAccounts/sqlDatabases', variables('accountName'),
parameters('databaseName'))]"
            ],
            "properties":
            {
                "resource":{
                    "id":  "[parameters('containerName')]",
                    "partitionKey": {
                        "paths": [
                            "/myPartitionKey"
                        ],
                        "kind": "Hash"
                    },
                    "indexingPolicy": {
                        "indexingMode": "consistent",
                        "includedPaths": [
                            {
                                "path": "/*"
                            }
                        ],
                        "excludedPaths": [
                            {
                                "path": "/myPathToNotIndex/*"
                            }
                        ],
                        "compositeIndexes":[
                            [
                                {
                                    "path":"/name",
                                    "order":"ascending"
                                },
                                {
                                    "path":"/age",
                                    "order":"descending"
                                }
                            ]
                        ],
                        "spatialIndexes": [
                            {
                                "path": "/path/to/geojson/property/?",
                                "types": [
                                    "Point",
                                    "Polygon",
                                    "MultiPolygon",
                                    "LineString"
                                ]
                            }
                        ]
                    },
                    "defaultTtl": 86400,
                    "uniqueKeyPolicy": {
                        "uniqueKeys": [
                            {
                                "paths": [
                                    "/phoneNumber"
                                ]
                            }
                        ]
                    }
                },
                "options": {
                    "autoscaleSettings": {
```

```
                        "maxThroughput": "[parameters('autoscaleMaxThroughput')]"
                    }
                }
            }
        }
    ]
}
```

Azure Cosmos account with analytical store

This template creates an Azure Cosmos account in one region with a container with Analytical TTL enabled and options for manual or autoscale throughput. This template is also available for one-click deploy from Azure Quickstart Templates Gallery.

Deploy to Azure

```
{
    "$schema": "https://schema.management.azure.com/schemas/2019-04-01/deploymentTemplate.json#",
    "contentVersion": "1.0.0.0",
    "parameters": {
        "accountName": {
            "type": "string",
            "defaultValue": "[concat('cosmos', uniqueString(resourceGroup().id))]",
            "metadata": {
                "description": "Cosmos DB account name"
            }
        },
        "location": {
            "type": "string",
            "defaultValue": "[resourceGroup().location]",
            "metadata": {
                "description": "Location for the Cosmos DB account."
            }
        },
        "databaseName": {
            "type": "string",
            "defaultValue": "database1",
            "metadata": {
                "description": "The name for the database"
            }
        },
        "containerName": {
            "type": "string",
            "defaultValue": "container1",
            "metadata": {
                "description": "The name for the container"
            }
        },
        "partitionKeyPath": {
            "type": "string",
            "defaultValue": "/partitionKey",
            "metadata": {
                "description": "The partition key for the container"
            }
        },
        "throughputPolicy": {
            "type": "string",
            "defaultValue": "Autoscale",
            "allowedValues": [
                "Manual",
                "Autoscale"
            ],
            "metadata": {
                "description": "The throughput policy for the container"
```

```
            }
        },
        "manualProvisionedThroughput": {
            "type": "int",
            "defaultValue": 400,
            "minValue": 400,
            "maxValue": 1000000,
            "metadata": {
                "description": "Throughput value when using Manual Throughput Policy for the container"
            }
        },
        "autoscaleMaxThroughput": {
            "type": "int",
            "defaultValue": 4000,
            "minValue": 4000,
            "maxValue": 1000000,
            "metadata": {
                "description": "Maximum throughput when using Autoscale Throughput Policy for the container"
            }
        }
    },
    "variables": {
        "accountName": "[toLower(parameters('accountName'))]",
        "locations": [
            {
                "locationName": "[parameters('location')]",
                "failoverPriority": 0,
                "isZoneRedundant": false
            }
        ],
        "throughputPolicy": {
            "Manual": {
                "Throughput": "[parameters('manualProvisionedThroughput')]"
            },
            "Autoscale": {
                "autoscaleSettings": { "maxThroughput": "[parameters('autoscaleMaxThroughput')]" }
            }
        }
    },
    "resources": [
        {
            "type": "Microsoft.DocumentDB/databaseAccounts",
            "name": "[variables('accountName')]",
            "apiVersion": "2020-04-01",
            "location": "[parameters('location')]",
            "properties": {
                "consistencyPolicy": {
                    "defaultConsistencyLevel": "Session"
                },
                "databaseAccountOfferType": "Standard",
                "locations": "[variables('locations')]",
                "enableAnalyticalStorage": true
            }
        },
        {
            "type": "Microsoft.DocumentDB/databaseAccounts/sqlDatabases",
            "name": "[concat(variables('accountName'), '/', parameters('databaseName'))]",
            "apiVersion": "2020-04-01",
            "dependsOn": [
                "[resourceId('Microsoft.DocumentDB/databaseAccounts', variables('accountName'))]"
            ],
            "properties": {
                "resource": {
                    "id": "[parameters('databaseName')]"
                }
            }
        },
        {
            "type": "Microsoft.DocumentDB/databaseAccounts/sqlDatabases/containers",
```

```
            "name": "[concat(variables('accountName'), '/', parameters('databaseName'), '/',
parameters('containerName'))]",
            "apiVersion": "2020-04-01",
            "dependsOn": [
                "[resourceId('Microsoft.DocumentDB/databaseAccounts/sqlDatabases', variables('accountName'),
parameters('databaseName'))]"
            ],
            "properties": {
                "resource": {
                    "id": "[parameters('containerName')]",
                    "partitionKey": {
                        "paths": [
                            "[parameters('partitionKeyPath')]"
                        ],
                        "kind": "Hash"
                    },
                    "analyticalStorageTtl": -1
                },
                "options": "[variables('throughputPolicy')[parameters('throughputPolicy')]]"
            }
        }
    ]
}
```

Azure Cosmos account with standard provisioned throughput

This template creates an Azure Cosmos account in two regions with options for consistency and failover, with database and container configured for standard throughput that has most policy options enabled. This template is also available for one-click deploy from Azure Quickstart Templates Gallery.

Deploy to Azure

```
{
  "$schema": "https://schema.management.azure.com/schemas/2019-04-01/deploymentTemplate.json#",
  "contentVersion": "1.0.0.0",
  "parameters": {
    "accountName": {
      "type": "string",
      "defaultValue": "[concat('sql-', uniqueString(resourceGroup().id))]",
      "metadata": {
        "description": "Cosmos DB account name, max length 44 characters"
      }
    },
    "location": {
      "type": "string",
      "defaultValue": "[resourceGroup().location]",
      "metadata": {
        "description": "Location for the Cosmos DB account."
      }
    },
    "primaryRegion": {
      "type": "string",
      "metadata": {
        "description": "The primary replica region for the Cosmos DB account."
      }
    },
    "secondaryRegion": {
      "type": "string",
      "metadata": {
        "description": "The secondary replica region for the Cosmos DB account."
      }
    },
    "defaultConsistencyLevel": {
      "type": "string",
      "defaultValue": "Session",
```

```
      "defaultValue": "Session",
      "allowedValues": [ "Eventual", "ConsistentPrefix", "Session", "BoundedStaleness", "Strong" ],
      "metadata": {
        "description": "The default consistency level of the Cosmos DB account."
      }
    },
    "maxStalenessPrefix": {
      "type": "int",
      "defaultValue": 100000,
      "minValue": 10,
      "maxValue": 2147483647,
      "metadata": {
        "description": "Max stale requests. Required for BoundedStaleness. Valid ranges, Single Region: 10 to
1000000. Multi Region: 100000 to 1000000."
      }
    },
    "maxIntervalInSeconds": {
      "type": "int",
      "defaultValue": 300,
      "minValue": 5,
      "maxValue": 86400,
      "metadata": {
        "description": "Max lag time (minutes). Required for BoundedStaleness. Valid ranges, Single Region: 5
to 84600. Multi Region: 300 to 86400."
      }
    },
    "automaticFailover": {
      "type": "bool",
      "defaultValue": true,
      "allowedValues": [ true, false ],
      "metadata": {
        "description": "Enable automatic failover for regions"
      }
    },
    "databaseName": {
      "type": "string",
      "defaultValue": "myDatabase",
      "metadata": {
        "description": "The name for the database"
      }
    },
    "containerName": {
      "type": "string",
      "defaultValue": "myContainer",
      "metadata": {
        "description": "The name for the container"
      }
    },
    "throughput": {
      "type": "int",
      "defaultValue": 400,
      "minValue": 400,
      "maxValue": 1000000,
      "metadata": {
        "description": "The throughput for the container"
      }
    }
  },
  "variables": {
    "accountName": "[toLower(parameters('accountName'))]",
    "consistencyPolicy": {
      "Eventual": {
        "defaultConsistencyLevel": "Eventual"
      },
      "ConsistentPrefix": {
        "defaultConsistencyLevel": "ConsistentPrefix"
      },
      "Session": {
        "defaultConsistencyLevel": "Session"
      }
```

```
      },
      "BoundedStaleness": {
        "defaultConsistencyLevel": "BoundedStaleness",
        "maxStalenessPrefix": "[parameters('maxStalenessPrefix')]",
        "maxIntervalInSeconds": "[parameters('maxIntervalInSeconds')]"
      },
      "Strong": {
        "defaultConsistencyLevel": "Strong"
      }
    },
    "locations": [
      {
        "locationName": "[parameters('primaryRegion')]",
        "failoverPriority": 0,
        "isZoneRedundant": false
      },
      {
        "locationName": "[parameters('secondaryRegion')]",
        "failoverPriority": 1,
        "isZoneRedundant": false
      }
    ]
  },
  "resources": [
    {
      "type": "Microsoft.DocumentDB/databaseAccounts",
      "apiVersion": "2020-06-01-preview",
      "name": "[variables('accountName')]",
      "location": "[parameters('location')]",
      "kind": "GlobalDocumentDB",
      "properties": {
        "consistencyPolicy": "[variables('consistencyPolicy')[parameters('defaultConsistencyLevel')]]",
        "locations": "[variables('locations')]",
        "databaseAccountOfferType": "Standard",
        "enableAutomaticFailover": "[parameters('automaticFailover')]"
      }
    },
    {
      "type": "Microsoft.DocumentDB/databaseAccounts/sqlDatabases",
      "apiVersion": "2020-06-01-preview",
      "name": "[concat(variables('accountName'), '/', parameters('databaseName'))]",
      "dependsOn": [ "[resourceId('Microsoft.DocumentDB/databaseAccounts', variables('accountName'))]" ],
      "properties": {
        "resource": {
          "id": "[parameters('databaseName')]"
        }
      }
    },
    {
      "type": "Microsoft.DocumentDB/databaseAccounts/sqlDatabases/containers",
      "apiVersion": "2020-06-01-preview",
      "name": "[concat(variables('accountName'), '/', parameters('databaseName'), '/',
parameters('containerName'))]",
      "dependsOn": [ "[resourceId('Microsoft.DocumentDB/databaseAccounts/sqlDatabases',
variables('accountName'), parameters('databaseName'))]" ],
      "properties": {
        "resource": {
          "id": "[parameters('containerName')]",
          "partitionKey": {
            "paths": [
              "/myPartitionKey"
            ],
            "kind": "Hash"
          },
          "indexingPolicy": {
            "indexingMode": "consistent",
            "includedPaths": [
              {
                "path": "/*"
```

```
            }
          ],
          "excludedPaths": [
            {
              "path": "/myPathToNotIndex/*"
            }
          ],
          "compositeIndexes": [
            [
              {
                "path": "/name",
                "order": "ascending"
              },
              {
                "path": "/age",
                "order": "descending"
              }
            ]
          ],
          "spatialIndexes": [
            {
              "path": "/path/to/geojson/property/?",
              "types": [
                "Point",
                "Polygon",
                "MultiPolygon",
                "LineString"
              ]
            }
          ]
        },
        "defaultTtl": 86400,
        "uniqueKeyPolicy": {
          "uniqueKeys": [
            {
              "paths": [
                "/phoneNumber"
              ]
            }
          ]
        }
      },
      "options": { "throughput": "[parameters('throughput')]" }
    }
  }
]
}
```

Azure Cosmos DB container with server-side functionality

This template creates an Azure Cosmos account, database and container with with a stored procedure, trigger, and user-defined function. This template is also available for one-click deploy from Azure Quickstart Templates Gallery.

Deploy to Azure

```
{
    "$schema": "https://schema.management.azure.com/schemas/2019-04-01/deploymentTemplate.json#",
    "contentVersion": "1.0.0.0",
    "parameters": {
        "accountName": {
            "type": "string",
            "defaultValue": "[concat('cosmos-', uniqueString(resourceGroup().id))]",
            "metadata": {
                "description": "Cosmos DB account name"
            }
```

```
        },
        "location": {
            "type": "string",
            "defaultValue": "[resourceGroup().location]",
            "metadata": {
                "description": "Location for the Cosmos DB account."
            }
        },
        "primaryRegion":{
            "type":"string",
            "metadata": {
                "description": "The primary replica region for the Cosmos DB account."
            }
        },
        "secondaryRegion":{
            "type":"string",
            "metadata": {
              "description": "The secondary replica region for the Cosmos DB account."
          }
        },
        "defaultConsistencyLevel": {
            "type": "string",
            "defaultValue": "Session",
            "allowedValues": [ "Eventual", "ConsistentPrefix", "Session", "BoundedStaleness", "Strong" ],
            "metadata": {
                "description": "The default consistency level of the Cosmos DB account."
            }
        },
        "maxStalenessPrefix": {
            "type": "int",
            "minValue": 10,
            "defaultValue": 100000,
            "maxValue": 2147483647,
            "metadata": {
                "description": "Max stale requests. Required for BoundedStaleness. Valid ranges, Single
Region: 10 to 1000000. Multi Region: 100000 to 1000000."
            }
        },
        "maxIntervalInSeconds": {
            "type": "int",
            "minValue": 5,
            "defaultValue": 300,
            "maxValue": 86400,
            "metadata": {
                "description": "Max lag time (seconds). Required for BoundedStaleness. Valid ranges, Single
Region: 5 to 84600. Multi Region: 300 to 86400."
            }
        },
        "automaticFailover": {
            "type": "bool",
            "defaultValue": true,
            "allowedValues": [ true, false ],
            "metadata": {
                "description": "Enable automatic failover for regions"
            }
        },
        "databaseName": {
            "type": "string",
            "metadata": {
                "description": "The name for the Core (SQL) database"
            }
        },
        "containerName": {
            "type": "string",
            "defaultValue": "container1",
            "metadata": {
                "description": "The name for the Core (SQL) API container"
            }
        },
```

```
        "throughput": {
            "type": "int",
            "defaultValue": 400,
            "minValue": 400,
            "maxValue": 1000000,
            "metadata": {
                "description": "The throughput for the container"
            }
        }
    },
    "variables": {
        "accountName": "[toLower(parameters('accountName'))]",
        "consistencyPolicy": {
            "Eventual": {
                "defaultConsistencyLevel": "Eventual"
            },
            "ConsistentPrefix": {
                "defaultConsistencyLevel": "ConsistentPrefix"
            },
            "Session": {
                "defaultConsistencyLevel": "Session"
            },
            "BoundedStaleness": {
                "defaultConsistencyLevel": "BoundedStaleness",
                "maxStalenessPrefix": "[parameters('maxStalenessPrefix')]",
                "maxIntervalInSeconds": "[parameters('maxIntervalInSeconds')]"
            },
            "Strong": {
                "defaultConsistencyLevel": "Strong"
            }
        },
        "locations":
        [
            {
                "locationName": "[parameters('primaryRegion')]",
                "failoverPriority": 0,
                "isZoneRedundant": false
            },
            {
                "locationName": "[parameters('secondaryRegion')]",
                "failoverPriority": 1,
                "isZoneRedundant": false
            }
        ]
    },
    "resources":
    [
        {
            "type": "Microsoft.DocumentDB/databaseAccounts",
            "name": "[variables('accountName')]",
            "apiVersion": "2020-03-01",
            "location": "[parameters('location')]",
            "kind": "GlobalDocumentDB",
            "properties": {
                "consistencyPolicy": "[variables('consistencyPolicy')
[parameters('defaultConsistencyLevel')]]",
                "locations": "[variables('locations')]",
                "databaseAccountOfferType": "Standard",
                "enableAutomaticFailover": "[parameters('automaticFailover')]"
            }
        },
        {
            "type": "Microsoft.DocumentDB/databaseAccounts/sqlDatabases",
            "name": "[concat(variables('accountName'), '/', parameters('databaseName'))]",
            "apiVersion": "2020-03-01",
            "dependsOn": [ "[resourceId('Microsoft.DocumentDB/databaseAccounts', variables('accountName'))]"
],
            "properties":{
                "resource":{
```

```
                    "id": "[parameters('databaseName')]"
                }
            }
        },
        {
            "type": "Microsoft.DocumentDB/databaseAccounts/sqlDatabases/containers",
            "name": "[concat(variables('accountName'), '/', parameters('databaseName'), '/',
parameters('containerName'))]",
            "apiVersion": "2020-03-01",
            "dependsOn": [ "[resourceId('Microsoft.DocumentDB/databaseAccounts/sqlDatabases',
variables('accountName'), parameters('databaseName'))]" ],
            "properties":
            {
                "resource":{
                    "id":  "[parameters('containerName')]",
                    "partitionKey": {
                        "paths": ["/myPartitionKey"],
                        "kind": "Hash"
                    },
                    "indexingPolicy": {
                        "indexingMode": "consistent",
                        "includedPaths": [{ "path": "/*" } ],
                        "excludedPaths": [{ "path": "/myPathToNotIndex/*" } ]
                    }
                },
                "options": { "throughput": "[parameters('throughput')]" }
            },
            "resources":
            [
                {
                    "type": "Microsoft.DocumentDB/databaseAccounts/sqlDatabases/containers/storedProcedures",
                    "name": "[concat(variables('accountName'), '/', parameters('databaseName'), '/',
parameters('containerName'), '/myStoredProcedure')]",
                    "apiVersion": "2020-03-01",
                    "dependsOn": [ "
[resourceId('Microsoft.DocumentDB/databaseAccounts/sqlDatabases/containers', variables('accountName'),
parameters('databaseName'), parameters('containerName'))]" ],
                    "properties": {
                        "resource": {
                        "id": "myStoredProcedure",
                        "body": "function () { var context = getContext(); var response =
context.getResponse(); response.setBody('Hello, World'); }"
                        }
                    }
                },
                {
                    "type": "Microsoft.DocumentDB/databaseAccounts/sqlDatabases/containers/triggers",
                    "name": "[concat(variables('accountName'), '/', parameters('databaseName'), '/',
parameters('containerName'), '/myPreTrigger')]",
                    "apiVersion": "2020-03-01",
                    "dependsOn": [ "
[resourceId('Microsoft.DocumentDB/databaseAccounts/sqlDatabases/containers', variables('accountName'),
parameters('databaseName'), parameters('containerName'))]" ],
                    "properties": {
                        "resource": {
                        "id": "myPreTrigger",
                        "triggerType": "Pre",
                        "triggerOperation": "Create",
                        "body": "function validateToDoItemTimestamp(){var context=getContext();var
request=context.getRequest();var itemToCreate=request.getBody();if(!('timestamp'in itemToCreate)){var ts=new
Date();itemToCreate['timestamp']=ts.getTime();}request.setBody(itemToCreate);}"
                        }
                    }
                },
                {
                    "type":
"Microsoft.DocumentDB/databaseAccounts/sqlDatabases/containers/userDefinedFunctions",
                    "name": "[concat(variables('accountName'), '/', parameters('databaseName'), '/',
parameters('containerName'), '/myUserDefinedFunction')]",
```

```
                    "apiVersion": "2020-03-01",
                    "dependsOn": [ "
[resourceId('Microsoft.DocumentDB/databaseAccounts/sqlDatabases/containers', variables('accountName'),
parameters('databaseName'), parameters('containerName'))]" ],
                    "properties": {
                        "resource": {
                        "id": "myUserDefinedFunction",
                        "body": "function tax(income){if(income==undefined)throw'no
input';if(income<1000)return income*0.1;else if(income<10000)return income*0.2;else return income*0.4;}"
                        }
                    }
                }
            ]
        }
    ]
}
```

Free tier Azure Cosmos DB account

This template creates a free-tier Azure Cosmos account and a database with shared throughput that can be shared with up to 25 containers. This template is also available for one-click deploy from Azure Quickstart Templates Gallery.

Deploy to Azure

```
{
    "$schema": "https://schema.management.azure.com/schemas/2019-04-01/deploymentTemplate.json#",
    "contentVersion": "1.0.0.0",
    "parameters": {
        "accountName": {
            "type": "string",
            "defaultValue": "[concat('cosmos-', uniqueString(resourceGroup().id))]",
            "metadata": {
                "description": "Cosmos DB account name"
            }
        },
        "location": {
            "type": "string",
            "defaultValue": "[resourceGroup().location]",
            "metadata": {
                "description": "Location for the Cosmos DB account."
            }
        },
        "databaseName": {
            "type": "string",
            "metadata": {
                "description": "The name for the Core (SQL) database"
            }
        }
    },
    "variables": {
        "accountName": "[toLower(parameters('accountName'))]"
    },
    "resources": [
        {
            "type": "Microsoft.DocumentDB/databaseAccounts",
            "apiVersion": "2020-03-01",
            "name": "[parameters('accountName')]",
            "location": "[parameters('location')]",
            "properties": {
                "enableFreeTier": true,
                "databaseAccountOfferType": "Standard",
                "consistencyPolicy": {
                    "defaultConsistencyLevel": "Session"
                }
            }
        },
        {
            "type": "Microsoft.DocumentDB/databaseAccounts/sqlDatabases",
            "apiVersion": "2020-03-01",
            "name": "[concat(variables('accountName'), '/', parameters('databaseName'))]",
            "dependsOn": [ "[resourceId('Microsoft.DocumentDB/databaseAccounts', variables('accountName'))]"
],
            "properties": {
                "resource": {
                    "id": "[parameters('databaseName')]"
                },
                "options": {"throughput": 400}
            }
        }
    ]
}
```

Next steps

Here are some additional resources:

- Azure Resource Manager documentation
- Azure Cosmos DB resource provider schema

- Azure Cosmos DB Quickstart templates
- Troubleshoot common Azure Resource Manager deployment errors

Move an Azure Cosmos DB account to another region

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: SQL API

This article describes how to either:

- Move a region where data is replicated in Azure Cosmos DB.
- Migrate account (Azure Resource Manager) metadata and data from one region to another.

Move data from one region to another

Azure Cosmos DB supports data replication natively, so moving data from one region to another is simple. You can accomplish it by using the Azure portal, Azure PowerShell, or the Azure CLI. It involves the following steps:

1. Add a new region to the account.

   To add a new region to an Azure Cosmos DB account, see Add/remove regions to an Azure Cosmos DB account.

2. Perform a manual failover to the new region.

   When the region that's being removed is currently the write region for the account, you'll need to start a failover to the new region added in the previous step. This is a zero-downtime operation. If you're moving a read region in a multiple-region account, you can skip this step.

   To start a failover, see Perform manual failover on an Azure Cosmos account.

3. Remove the original region.

   To remove a region from an Azure Cosmos DB account, see Add/remove regions from your Azure Cosmos DB account.

Migrate Azure Cosmos DB account metadata

Azure Cosmos DB does not natively support migrating account metadata from one region to another. To migrate both the account metadata and customer data from one region to another, you must create a new account in the desired region and then copy the data manually.

A near-zero-downtime migration for the SQL API requires the use of the change feed or a tool that uses it. If you're migrating the MongoDB API, the Cassandra API, or another API, or to learn more about options for migrating data between accounts, see Options to migrate your on-premises or cloud data to Azure Cosmos DB.

The following steps demonstrate how to migrate an Azure Cosmos DB account for the SQL API and its data from one region to another:

1. Create a new Azure Cosmos DB account in the desired region.

   To create a new account via the Azure portal, PowerShell, or the Azure CLI, see Create an Azure Cosmos DB account.

2. Create a new database and container.

To create a new database and container, see Create an Azure Cosmos container.

3. Migrate data by using the Azure Cosmos DB Live Data Migrator tool.

   To migrate data with near zero downtime, see Azure Cosmos DB Live Data Migrator tool.

4. Update the application connection string.

   With the Live Data Migrator tool still running, update the connection information in the new deployment of your application. You can retrieve the endpoints and keys for your application from the Azure portal.

5. Redirect requests to the new application.

   After the new application is connected to Azure Cosmos DB, you can redirect client requests to your new deployment.

6. Delete any resources that you no longer need.

   With requests now fully redirected to the new instance, you can delete the old Azure Cosmos DB account and the Live Data Migrator tool.

Next steps

For more information and examples on how to manage the Azure Cosmos account as well as databases and containers, read the following articles:

- Manage an Azure Cosmos account
- Change feed in Azure Cosmos DB

Prevent Azure Cosmos DB resources from being deleted or changed

12/4/2020 • 3 minutes to read • Edit Online

APPLIES TO: SQL API

As an administrator, you may need to lock an Azure Cosmos account, database or container to prevent other users in your organization from accidentally deleting or modifying critical resources. You can set the lock level to CanNotDelete or ReadOnly.

- **CanNotDelete** means authorized users can still read and modify a resource, but they can't delete the resource.
- **ReadOnly** means authorized users can read a resource, but they can't delete or update the resource. Applying this lock is similar to restricting all authorized users to the permissions granted by the Reader role.

How locks are applied

When you apply a lock at a parent scope, all resources within that scope inherit the same lock. Even resources you add later inherit the lock from the parent. The most restrictive lock in the inheritance takes precedence.

Unlike Azure role-based access control, you use management locks to apply a restriction across all users and roles. To learn about Azure RBAC for Azure Cosmos DB see, Azure role-based access control in Azure Cosmos DB.

Resource Manager locks apply only to operations that happen in the management plane, which consists of operations sent to https://management.azure.com. The locks don't restrict how resources perform their own functions. Resource changes are restricted, but resource operations aren't restricted. For example, a ReadOnly lock on an Azure Cosmos container prevents you from deleting or modifying the container. It doesn't prevent you from creating, updating, or deleting data in the container. Data transactions are permitted because those operations aren't sent to https://management.azure.com.

Manage locks

> **WARNING**
>
> Resource locks do not work for changes made by users accessing Azure Cosmos DB using account keys unless the Azure Cosmos account is first locked by enabling the disableKeyBasedMetadataWriteAccess property. Care should be taken before enabling this property to ensure it does not break existing applications that make changes to resources using any SDK, Azure portal or 3rd party tools that connect via account keys and modify resources such as changing throughput, updating index policies, etc. To learn more and to go through a checklist to ensure your applications continue to function see, Preventing changes from the Azure Cosmos DB SDKs

PowerShell

```
$resourceGroupName = "myResourceGroup"
$accountName = "my-cosmos-account"
$lockName = "$accountName-Lock"

# First, update the account to prevent changes by anything that connects via account keys
Update-AzCosmosDBAccount -ResourceGroupName $resourceGroupName -Name $accountName -
DisableKeyBasedMetadataWriteAccess true

# Create a Delete Lock on an Azure Cosmos account resource and all child resources
New-AzResourceLock `
    -ApiVersion "2020-04-01" `
    -ResourceType "Microsoft.DocumentDB/databaseAccounts" `
    -ResourceGroupName $resourceGroupName `
    -ResourceName $accountName `
    -LockName $lockName `
    -LockLevel "CanNotDelete" # CanNotDelete or ReadOnly
```

Azure CLI

```
resourceGroupName='myResourceGroup'
accountName='my-cosmos-account'
$lockName="$accountName-Lock"

# First, update the account to prevent changes by anything that connects via account keys
az cosmosdb update  --name $accountName --resource-group $resourceGroupName  --disable-key-based-metadata-
write-access true

# Create a Delete Lock on an Azure Cosmos account resource
az lock create --name $lockName \
    --resource-group $resourceGroupName \
    --resource-type Microsoft.DocumentDB/databaseAccount \
    --lock-type 'CanNotDelete' # CanNotDelete or ReadOnly \
    --resource $accountName
```

Template

When applying a lock to an Azure Cosmos DB resource, use the following formats:

- name - `{resourceName}/Microsoft.Authorization/{lockName}`
- type - `{resourceProviderNamespace}/{resourceType}/providers/locks`

> **IMPORTANT**
>
> When modifying an existing Azure Cosmos account, make sure to include the other properties for your account and child resources when redploying with this property. Do not deploy this template as is or it will reset all of your account properties.

```
"resources": [
    {
        "type": "Microsoft.DocumentDB/databaseAccounts",
        "name": "[variables('accountName')]",
        "apiVersion": "2020-04-01",
        "kind": "GlobalDocumentDB",
        "location": "[parameters('location')]",
        "properties": {
            "consistencyPolicy": "[variables('consistencyPolicy')[parameters('defaultConsistencyLevel')]]",
            "locations": "[variables('locations')]",
            "databaseAccountOfferType": "Standard",
            "enableAutomaticFailover": "[parameters('automaticFailover')]",
            "disableKeyBasedMetadataWriteAccess": true
        }
    },
    {
        "type": "Microsoft.DocumentDB/databaseAccounts/providers/locks",
        "apiVersion": "2020-04-01",
        "name": "[concat(variables('accountName'), '/Microsoft.Authorization/siteLock')]",
        "dependsOn": [
        "[resourceId('Microsoft.DocumentDB/databaseAccounts', variables('accountName'))]"
        ],
        "properties": {
        "level": "CanNotDelete",
        "notes": "Cosmos account should not be deleted."
        }
    }
]
```

Next steps

- Overview of Azure Resource Manager Locks

Configure multi-region writes in your applications that use Azure Cosmos DB

12/4/2020 • 3 minutes to read • Edit Online

APPLIES TO: SQL API

Once an account has been created with multiple write regions enabled, you must make two changes in your application to the ConnectionPolicy for the DocumentClient to enable the multi-region writes and multi-homing capabilities in Azure Cosmos DB. Within the ConnectionPolicy, set UseMultipleWriteLocations to true and pass the name of the region where the application is deployed to SetCurrentLocation. This will populate the PreferredLocations property based on the geo-proximity from location passed in. If a new region is later added to the account, the application does not have to be updated or redeployed, it will automatically detect the closer region and will auto-home on to it should a regional event occur.

> **NOTE**
>
> Cosmos accounts initially configured with single write region can be configured to multiple write regions with zero down time. To learn more see, Configure multiple-write regions

Azure portal

To enable multi-region writes from Azure portal, use the following steps:

1. Sign-in to the Azure portal.
2. Navigate to your Azure Cosmos account and from the menu, open the **Replicate data globally** pane.
3. Under the **Multi-region writes** option, choose **enable**. It automatically adds the existing regions to read and write regions.
4. You can add additional regions by selecting the icons on the map or by selecting the **Add region** button. All the regions you add will have both read and writes enabled.
5. After you update the region list, select **save** to apply the changes.

.NET SDK v2

To enable multi-region writes in your application, set `UseMultipleWriteLocations` to `true`. Also, set `SetCurrentLocation` to the region in which the application is being deployed and where Azure Cosmos DB is replicated:

```
ConnectionPolicy policy = new ConnectionPolicy
    {
        ConnectionMode = ConnectionMode.Direct,
        ConnectionProtocol = Protocol.Tcp,
        UseMultipleWriteLocations = true
    };
policy.SetCurrentLocation("West US 2");
```

.NET SDK v3

To enable multi-region writes in your application, set `ApplicationRegion` to the region in which the application is being deployed and where Cosmos DB is replicated:

```
CosmosClient cosmosClient = new CosmosClient(
    "<connection-string-from-portal>",
    new CosmosClientOptions()
    {
        ApplicationRegion = Regions.WestUS2,
    });
```

Optionally, you can use the `CosmosClientBuilder` and `WithApplicationRegion` to achieve the same result:

```
CosmosClientBuilder cosmosClientBuilder = new CosmosClientBuilder("<connection-string-from-portal>")
            .WithApplicationRegion(Regions.WestUS2);
CosmosClient client = cosmosClientBuilder.Build();
```

Java V4 SDK

To enable multi-region writes in your application, call `.multipleWriteRegionsEnabled(true)` and `.preferredRegions(preferredRegions)` in the client builder, where `preferredRegions` is a `List` containing one element - that is the region in which the application is being deployed and where Cosmos DB is replicated:

- Async
- Sync

Java SDK V4 (Maven com.azure::azure-cosmos) Async API

```
ArrayList<String> preferredRegions = new ArrayList<String>();
preferredRegions.add(region);

CosmosAsyncClient client =
        new CosmosClientBuilder()
                .endpoint(HOST)
                .key(MASTER_KEY)
                .multipleWriteRegionsEnabled(true)
                .preferredRegions(preferredRegions)
                .buildAsyncClient();
```

Async Java V2 SDK

The Java V2 SDK used the Maven com.microsoft.azure::azure-cosmosdb. To enable multi-region writes in your application, set `policy.setUsingMultipleWriteLocations(true)` and set `policy.setPreferredLocations` to the region in which the application is being deployed and where Cosmos DB is replicated:

```
ConnectionPolicy policy = new ConnectionPolicy();
policy.setUsingMultipleWriteLocations(true);
policy.setPreferredLocations(Collections.singletonList(region));

AsyncDocumentClient client =
    new AsyncDocumentClient.Builder()
        .withMasterKeyOrResourceToken(this.accountKey)
        .withServiceEndpoint(this.accountEndpoint)
        .withConsistencyLevel(ConsistencyLevel.Eventual)
        .withConnectionPolicy(policy).build();
```

Node.js, JavaScript, and TypeScript SDKs

To enable multi-region writes in your application, set `connectionPolicy.UseMultipleWriteLocations` to `true`. Also, set `connectionPolicy.PreferredLocations` to the region in which the application is being deployed and where Cosmos DB is replicated:

```
const connectionPolicy: ConnectionPolicy = new ConnectionPolicy();
connectionPolicy.UseMultipleWriteLocations = true;
connectionPolicy.PreferredLocations = [region];

const client = new CosmosClient({
  endpoint: config.endpoint,
  auth: { masterKey: config.key },
  connectionPolicy,
  consistencyLevel: ConsistencyLevel.Eventual
});
```

Python SDK

To enable multi-region writes in your application, set `connection_policy.UseMultipleWriteLocations` to `true`. Also, set `connection_policy.PreferredLocations` to the region in which the application is being deployed and where Cosmos DB is replicated.

```
connection_policy = documents.ConnectionPolicy()
connection_policy.UseMultipleWriteLocations = True
connection_policy.PreferredLocations = [region]

client = cosmos_client.CosmosClient(self.account_endpoint, {
                                    'masterKey': self.account_key}, connection_policy,
documents.ConsistencyLevel.Session)
```

Next steps

Read the following articles:

- Use session tokens to manage consistency in Azure Cosmos DB
- Conflict types and resolution policies in Azure Cosmos DB
- High availability in Azure Cosmos DB
- Consistency levels in Azure Cosmos DB
- Choose the right consistency level in Azure Cosmos DB
- Consistency, availability, and performance tradeoffs in Azure Cosmos DB
- Availability and performance tradeoffs for various consistency levels
- Globally scaling provisioned throughput

- Global distribution: Under the hood

Manage consistency levels in Azure Cosmos DB

12/4/2020 • 5 minutes to read • Edit Online

APPLIES TO: SQL API

This article explains how to manage consistency levels in Azure Cosmos DB. You learn how to configure the default consistency level, override the default consistency, manually manage session tokens, and understand the Probabilistically Bounded Staleness (PBS) metric.

> **NOTE**
>
> This article has been updated to use the new Azure PowerShell Az module. You can still use the AzureRM module, which will continue to receive bug fixes until at least December 2020. To learn more about the new Az module and AzureRM compatibility, see Introducing the new Azure PowerShell Az module. For Az module installation instructions, see Install Azure PowerShell.

Configure the default consistency level

The default consistency level is the consistency level that clients use by default.

- Azure portal
- CLI
- PowerShell

To view or modify the default consistency level, sign in to the Azure portal. Find your Azure Cosmos account, and open the **Default consistency** pane. Select the level of consistency you want as the new default, and then select **Save**. The Azure portal also provides a visualization of different consistency levels with music notes.

Override the default consistency level

Clients can override the default consistency level that is set by the service. Consistency level can be set on a per request, which overrides the default consistency level set at the account level.

> **TIP**
> Consistency can only be **relaxed** at the request level. To move from weaker to stronger consistency, update the default consistency for the Cosmos account.

.NET SDK

- .NET SDK V2
- .NET SDK V3

```
// Override consistency at the client level
documentClient = new DocumentClient(new Uri(endpoint), authKey, connectionPolicy, ConsistencyLevel.Eventual);

// Override consistency at the request level via request options
RequestOptions requestOptions = new RequestOptions { ConsistencyLevel = ConsistencyLevel.Eventual };

var response = await client.CreateDocumentAsync(collectionUri, document, requestOptions);
```

Java V4 SDK

- Async
- Sync

Java SDK V4 (Maven com.azure::azure-cosmos) Async API

```
CosmosAsyncClient client =
        new CosmosClientBuilder()
                .endpoint(HOST)
                .key(MASTER_KEY)
                .consistencyLevel(ConsistencyLevel.EVENTUAL)
                .buildAsyncClient();
```

Java V2 SDKs

- Async
- Sync

Async Java V2 SDK (Maven com.microsoft.azure::azure-cosmosdb)

```
// Override consistency at the client level
ConnectionPolicy policy = new ConnectionPolicy();

AsyncDocumentClient client =
        new AsyncDocumentClient.Builder()
                .withMasterKey(this.accountKey)
                .withServiceEndpoint(this.accountEndpoint)
                .withConsistencyLevel(ConsistencyLevel.Eventual)
                .withConnectionPolicy(policy).build();
```

Node.js/JavaScript/TypeScript SDK

```
// Override consistency at the client level
const client = new CosmosClient({
  /* other config... */
  consistencyLevel: ConsistencyLevel.Eventual
});

// Override consistency at the request level via request options
const { body } = await item.read({ consistencyLevel: ConsistencyLevel.Eventual });
```

Python SDK

```
# Override consistency at the client level
connection_policy = documents.ConnectionPolicy()
client = cosmos_client.CosmosClient(self.account_endpoint, {
                                    'masterKey': self.account_key}, connection_policy,
documents.ConsistencyLevel.Eventual)
```

Utilize session tokens

One of the consistency levels in Azure Cosmos DB is *Session* consistency. This is the default level applied to Cosmos accounts by default. When working with *Session* consistency, the client will use a session token internally with each read/query request to ensure that the set consistency level is maintained.

To manage session tokens manually, get the session token from the response and set them per request. If you don't need to manage session tokens manually, you don't need to use these samples. The SDK keeps track of session tokens automatically. If you don't set the session token manually, by default, the SDK uses the most recent session token.

.NET SDK

- .NET SDK V2
- .NET SDK V3

```
var response = await client.ReadDocumentAsync(
                UriFactory.CreateDocumentUri(databaseName, collectionName, "SalesOrder1"));
string sessionToken = response.SessionToken;

RequestOptions options = new RequestOptions();
options.SessionToken = sessionToken;
var response = await client.ReadDocumentAsync(
                UriFactory.CreateDocumentUri(databaseName, collectionName, "SalesOrder1"), options);
```

Java V4 SDK

- Async
- Sync

Java SDK V4 (Maven com.azure::azure-cosmos) Async API

```
// Get session token from response
CosmosItemResponse<JsonNode> response = container.readItem(itemId, new PartitionKey(partitionKey),
JsonNode.class).block();
String sessionToken = response.getSessionToken();

// Resume the session by setting the session token on the RequestOptions
CosmosItemRequestOptions options = new CosmosItemRequestOptions();
options.setSessionToken(sessionToken);
CosmosItemResponse<JsonNode> response2 = container.readItem(itemId, new PartitionKey(partitionKey),
JsonNode.class).block();
```

Java V2 SDKs

- Async
- Sync

Async Java V2 SDK (Maven com.microsoft.azure::azure-cosmosdb)

```
// Get session token from response
RequestOptions options = new RequestOptions();
options.setPartitionKey(new PartitionKey(document.get("mypk")));
Observable<ResourceResponse<Document>> readObservable = client.readDocument(document.getSelfLink(), options);
readObservable.single()           // we know there will be one response
  .subscribe(
      documentResourceResponse -> {
          System.out.println(documentResourceResponse.getSessionToken());
      },
      error -> {
          System.err.println("an error happened: " + error.getMessage());
      });

// Resume the session by setting the session token on RequestOptions
RequestOptions options = new RequestOptions();
requestOptions.setSessionToken(sessionToken);
Observable<ResourceResponse<Document>> readObservable = client.readDocument(document.getSelfLink(), options);
```

Node.js/JavaScript/TypeScript SDK

```
// Get session token from response
const { headers, item } = await container.items.create({ id: "meaningful-id" });
const sessionToken = headers["x-ms-session-token"];

// Immediately or later, you can use that sessionToken from the header to resume that session.
const { body } = await item.read({ sessionToken });
```

Python SDK

```
// Get the session token from the last response headers
item = client.ReadItem(item_link)
session_token = client.last_response_headers["x-ms-session-token"]

// Resume the session by setting the session token on the options for the request
options = {
    "sessionToken": session_token
}
item = client.ReadItem(doc_link, options)
```

Monitor Probabilistically Bounded Staleness (PBS) metric

How eventual is eventual consistency? For the average case, can we offer staleness bounds with respect to version history and time. The **Probabilistically Bounded Staleness (PBS)** metric tries to quantify the probability of staleness and shows it as a metric. To view the PBS metric, go to your Azure Cosmos account in the Azure portal. Open the **Metrics** pane, and select the **Consistency** tab. Look at the graph named **Probability of strongly consistent reads based on your workload (see PBS)**.

Next steps

Learn more about how to manage data conflicts, or move on to the next key concept in Azure Cosmos DB. See the following articles:

- Consistency Levels in Azure Cosmos DB
- Partitioning and data distribution
- Manage conflicts between regions
- Partitioning and data distribution
- Consistency tradeoffs in modern distributed database systems design
- High availability
- Azure Cosmos DB SLA

Manage conflict resolution policies in Azure Cosmos DB

12/4/2020 • 7 minutes to read • Edit Online

APPLIES TO: SQL API

With multi-region writes, when multiple clients write to the same item, conflicts may occur. When a conflict occurs, you can resolve the conflict by using different conflict resolution policies. This article describes how to manage conflict resolution policies.

Create a last-writer-wins conflict resolution policy

These samples show how to set up a container with a last-writer-wins conflict resolution policy. The default path for last-writer-wins is the timestamp field or the `_ts` property. For SQL API, this may also be set to a user-defined path with a numeric type. In a conflict, the highest value wins. If the path isn't set or it's invalid, it defaults to `_ts`. Conflicts resolved with this policy do not show up in the conflict feed. This policy can be used by all APIs.

.NET SDK

- .NET SDK V2
- .NET SDK V3

```
DocumentCollection lwwCollection = await createClient.CreateDocumentCollectionIfNotExistsAsync(
  UriFactory.CreateDatabaseUri(this.databaseName), new DocumentCollection
  {
      Id = this.lwwCollectionName,
      ConflictResolutionPolicy = new ConflictResolutionPolicy
      {
          Mode = ConflictResolutionMode.LastWriterWins,
          ConflictResolutionPath = "/myCustomId",
      },
  });
```

Java V4 SDK

- Async
- Sync

Java SDK V4 (Maven com.azure::azure-cosmos) Async API

```
ConflictResolutionPolicy policy = ConflictResolutionPolicy.createLastWriterWinsPolicy("/myCustomId");

CosmosContainerProperties containerProperties = new CosmosContainerProperties(container_id, partition_key);
containerProperties.setConflictResolutionPolicy(policy);
/* ...other container config... */
database.createContainerIfNotExists(containerProperties).block();
```

Java V2 SDKs

- Async Java V2 SDK
- Sync Java V2 SDK

Async Java V2 SDK (Maven com.microsoft.azure::azure-cosmosdb)

```
DocumentCollection collection = new DocumentCollection();
collection.setId(id);
ConflictResolutionPolicy policy = ConflictResolutionPolicy.createLastWriterWinsPolicy("/myCustomId");
collection.setConflictResolutionPolicy(policy);
DocumentCollection createdCollection = client.createCollection(databaseUri, collection,
null).toBlocking().value();
```

Node.js/JavaScript/TypeScript SDK

```
const database = client.database(this.databaseName);
const { container: lwwContainer } = await database.containers.createIfNotExists(
  {
    id: this.lwwContainerName,
    conflictResolutionPolicy: {
      mode: "LastWriterWins",
      conflictResolutionPath: "/myCustomId"
    }
  }
);
```

Python SDK

```
udp_collection = {
    'id': self.udp_collection_name,
    'conflictResolutionPolicy': {
        'mode': 'LastWriterWins',
        'conflictResolutionPath': '/myCustomId'
    }
}
udp_collection = self.try_create_document_collection(
    create_client, database, udp_collection)
```

Create a custom conflict resolution policy using a stored procedure

These samples show how to set up a container with a custom conflict resolution policy with a stored procedure to resolve the conflict. These conflicts don't show up in the conflict feed unless there's an error in your stored procedure. After the policy is created with the container, you need to create the stored procedure. The .NET SDK sample below shows an example. This policy is supported on Core (SQL) Api only.

Sample custom conflict resolution stored procedure

Custom conflict resolution stored procedures must be implemented using the function signature shown below. The function name does not need to match the name used when registering the stored procedure with the container but it does simplify naming. Here is a description of the parameters that must be implemented for this stored procedure.

- **incomingItem**: The item being inserted or updated in the commit that is generating the conflicts. Is null for delete operations.
- **existingItem**: The currently committed item. This value is non-null in an update and null for an insert or deletes.
- **isTombstone**: Boolean indicating if the incomingItem is conflicting with a previously deleted item. When true, existingItem is also null.
- **conflictingItems**: Array of the committed version of all items in the container that are conflicting with incomingItem on ID or any other unique index properties.

> **IMPORTANT**
> Just as with any stored procedure, a custom conflict resolution procedure can access any data with the same partition key and can perform any insert, update or delete operation to resolve conflicts.

This sample stored procedure resolves conflicts by selecting the lowest value from the `/myCustomId` path.

```
function resolver(incomingItem, existingItem, isTombstone, conflictingItems) {
  var collection = getContext().getCollection();

  if (!incomingItem) {
      if (existingItem) {

          collection.deleteDocument(existingItem._self, {}, function (err, responseOptions) {
              if (err) throw err;
          });
      }
  } else if (isTombstone) {
      // delete always wins.
  } else {
      if (existingItem) {
          if (incomingItem.myCustomId > existingItem.myCustomId) {
              return; // existing item wins
          }
      }

      var i;
      for (i = 0; i < conflictingItems.length; i++) {
          if (incomingItem.myCustomId > conflictingItems[i].myCustomId) {
              return; // existing conflict item wins
          }
      }

      // incoming item wins - clear conflicts and replace existing with incoming.
      tryDelete(conflictingItems, incomingItem, existingItem);
  }

  function tryDelete(documents, incoming, existing) {
      if (documents.length > 0) {
          collection.deleteDocument(documents[0]._self, {}, function (err, responseOptions) {
              if (err) throw err;

              documents.shift();
              tryDelete(documents, incoming, existing);
          });
      } else if (existing) {
          collection.replaceDocument(existing._self, incoming,
              function (err, documentCreated) {
                  if (err) throw err;
              });
      } else {
          collection.createDocument(collection.getSelfLink(), incoming,
              function (err, documentCreated) {
                  if (err) throw err;
              });
      }
  }
}
```

.NET SDK

- .NET SDK V2
- .NET SDK V3

```
DocumentCollection udpCollection = await createClient.CreateDocumentCollectionIfNotExistsAsync(
  UriFactory.CreateDatabaseUri(this.databaseName), new DocumentCollection
  {
      Id = this.udpCollectionName,
      ConflictResolutionPolicy = new ConflictResolutionPolicy
      {
          Mode = ConflictResolutionMode.Custom,
          ConflictResolutionProcedure = string.Format("dbs/{0}/colls/{1}/sprocs/{2}", this.databaseName,
this.udpCollectionName, "resolver"),
      },
  });

//Create the stored procedure
await clients[0].CreateStoredProcedureAsync(
UriFactory.CreateStoredProcedureUri(this.databaseName, this.udpCollectionName, "resolver"), new
StoredProcedure
{
    Id = "resolver",
    Body = File.ReadAllText(@"resolver.js")
});
```

Java V4 SDK

- Async
- Sync

Java SDK V4 (Maven com.azure::azure-cosmos) Async API

```
ConflictResolutionPolicy policy = ConflictResolutionPolicy.createCustomPolicy("resolver");

CosmosContainerProperties containerProperties = new CosmosContainerProperties(container_id, partition_key);
containerProperties.setConflictResolutionPolicy(policy);
/* ...other container config... */
database.createContainerIfNotExists(containerProperties).block();
```

Java V2 SDKs

- Async Java V2 SDK
- Sync Java V2 SDK

Async Java V2 SDK (Maven com.microsoft.azure::azure-cosmosdb)

```
DocumentCollection collection = new DocumentCollection();
collection.setId(id);
ConflictResolutionPolicy policy = ConflictResolutionPolicy.createCustomPolicy("resolver");
collection.setConflictResolutionPolicy(policy);
DocumentCollection createdCollection = client.createCollection(databaseUri, collection,
null).toBlocking().value();
```

After your container is created, you must create the `resolver` stored procedure.

Node.js/JavaScript/TypeScript SDK

```
const database = client.database(this.databaseName);
const { container: udpContainer } = await database.containers.createIfNotExists(
  {
    id: this.udpContainerName,
    conflictResolutionPolicy: {
      mode: "Custom",
      conflictResolutionProcedure: `dbs/${this.databaseName}/colls/${
        this.udpContainerName
      }/sprocs/resolver`
    }
  }
);
```

After your container is created, you must create the `resolver` stored procedure.

Python SDK

```
udp_collection = {
    'id': self.udp_collection_name,
    'conflictResolutionPolicy': {
        'mode': 'Custom',
        'conflictResolutionProcedure': 'dbs/' + self.database_name + "/colls/" + self.udp_collection_name +
'/sprocs/resolver'
    }
}
udp_collection = self.try_create_document_collection(
    create_client, database, udp_collection)
```

After your container is created, you must create the `resolver` stored procedure.

Create a custom conflict resolution policy

These samples show how to set up a container with a custom conflict resolution policy. These conflicts show up in the conflict feed.

.NET SDK

- .NET SDK V2
- .NET SDK V3

```
DocumentCollection manualCollection = await createClient.CreateDocumentCollectionIfNotExistsAsync(
  UriFactory.CreateDatabaseUri(this.databaseName), new DocumentCollection
  {
      Id = this.manualCollectionName,
      ConflictResolutionPolicy = new ConflictResolutionPolicy
      {
          Mode = ConflictResolutionMode.Custom,
      },
  });
```

Java V4 SDK

- Async
- Sync

Java SDK V4 (Maven com.azure::azure-cosmos) Async API

```
ConflictResolutionPolicy policy = ConflictResolutionPolicy.createCustomPolicy();

CosmosContainerProperties containerProperties = new CosmosContainerProperties(container_id, partition_key);
containerProperties.setConflictResolutionPolicy(policy);
/* ...other container config... */
database.createContainerIfNotExists(containerProperties).block();
```

Java V2 SDKs

- Async Java V2 SDK
- Sync Java V2 SDK

Async Java V2 SDK (Maven com.microsoft.azure::azure-cosmosdb)

```
DocumentCollection collection = new DocumentCollection();
collection.setId(id);
ConflictResolutionPolicy policy = ConflictResolutionPolicy.createCustomPolicy();
collection.setConflictResolutionPolicy(policy);
DocumentCollection createdCollection = client.createCollection(databaseUri, collection,
null).toBlocking().value();
```

Node.js/JavaScript/TypeScript SDK

```
const database = client.database(this.databaseName);
const {
  container: manualContainer
} = await database.containers.createIfNotExists({
  id: this.manualContainerName,
  conflictResolutionPolicy: {
    mode: "Custom"
  }
});
```

Python SDK

```
database = client.ReadDatabase("dbs/" + self.database_name)
manual_collection = {
    'id': self.manual_collection_name,
    'conflictResolutionPolicy': {
        'mode': 'Custom'
    }
}
manual_collection = client.CreateContainer(database['_self'], collection)
```

Read from conflict feed

These samples show how to read from a container's conflict feed. Conflicts show up in the conflict feed only if they weren't resolved automatically or if using a custom conflict policy.

.NET SDK

- .NET SDK V2
- .NET SDK V3

```
FeedResponse<Conflict> conflicts = await delClient.ReadConflictFeedAsync(this.collectionUri);
```

Java V2 SDKs

- Async Java V2 SDK
- Sync Java V2 SDK

Async Java V2 SDK (Maven com.microsoft.azure::azure-cosmosdb)

```
FeedResponse<Conflict> response = client.readConflicts(this.manualCollectionUri, null)
                    .first().toBlocking().single();
for (Conflict conflict : response.getResults()) {
    /* Do something with conflict */
}
```

Node.js/JavaScript/TypeScript SDK

```
const container = client
  .database(this.databaseName)
  .container(this.lwwContainerName);

const { result: conflicts } = await container.conflicts.readAll().toArray();
```

Python

```
conflicts_iterator = iter(client.ReadConflicts(self.manual_collection_link))
conflict = next(conflicts_iterator, None)
while conflict:
    # Do something with conflict
    conflict = next(conflicts_iterator, None)
```

Next steps

Learn about the following Azure Cosmos DB concepts:

- Global distribution - under the hood
- How to configure multi-region writes in your applications
- Configure clients for multihoming
- Add or remove regions from your Azure Cosmos DB account
- How to configuremulti-region writes in your applications.
- Partitioning and data distribution
- Indexing in Azure Cosmos DB

Convert session token formats in .NET SDK

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: SQL API

This article explains how to convert between different session token formats to ensure compatibility between SDK versions.

> **NOTE**
>
> By default, the SDK keeps track of the session token automatically and it will use the most recent session token. For more information, please visit Utilize session tokens. The instructions in this article only apply with the following conditions:
>
> - Your Azure Cosmos DB account uses Session consistency.
> - You are managing the session tokens are manually.
> - You are using multiple versions of the SDK at the same time.

Session token formats

There are two session token formats: **simple** and **vector**. These two formats are not interchangeable so, the format should be converted when passing to the client application with different versions.

- The **simple** session token format is used by the .NET SDK V1 (Microsoft.Azure.DocumentDB -version 1.x)
- The **vector** session token format is used by the .NET SDK V2 (Microsoft.Azure.DocumentDB -version 2.x)

Simple session token

A simple session token has this format: `{pkrangeid}:{globalLSN}`

Vector session token

A vector session token has the following format:

```
{pkrangeid}:{Version}#{GlobalLSN}#{RegionId1}={LocalLsn1}#{RegionId2}={LocalLsn2}....#{RegionIdN}={LocalLsnN}
```

Convert to Simple session token

To pass a session token to client using .NET SDK V1, use a **simple** session token format. For example, use the following sample code to convert it.

```
private static readonly char[] SegmentSeparator = (new[] { '#' });
private static readonly char[] PkRangeSeparator = (new[] { ':' });

// sessionTokenToConvert = session token from previous response
string[] items = sessionTokenToConvert.Split(PkRangeSeparator, StringSplitOptions.RemoveEmptyEntries);
string[] sessionTokenSegments = items[1].Split(SessionTokenHelpers.SegmentSeparator,
StringSplitOptions.RemoveEmptyEntries);

string sessionTokenInSimpleFormat;

if (sessionTokenSegments.Length == 1)
{
    // returning the same token since it already has the correct format
    sessionTokenInSimpleFormat = sessionTokenToConvert;
}
else
{
    long version = 0;
    long globalLSN = 0;

    if (!long.TryParse(sessionTokenSegments[0], out version)
        || !long.TryParse(sessionTokenSegments[1], out globalLSN))
    {
        throw new ArgumentException("Invalid session token format", sessionTokenToConvert);
    }

    sessionTokenInSimpleFormat = string.Format("{0}:{1}", items[0], globalLSN);
}
```

Convert to Vector session token

To pass a session token to client using .NET SDK V2, use the **vector** session token format. For example, use the following sample code to convert it.

```
private static readonly char[] SegmentSeparator = (new[] { '#' });
private static readonly char[] PkRangeSeparator = (new[] { ':' });

// sessionTokenToConvert = session token from previous response
string[] items = sessionTokenToConvert.Split(PkRangeSeparator, StringSplitOptions.RemoveEmptyEntries);
string[] sessionTokenSegments = items[1].Split(SegmentSeparator, StringSplitOptions.RemoveEmptyEntries);

string sessionTokenInVectorFormat;

if (sessionTokenSegments.Length == 1)
{
    long globalLSN = 0;
    if (long.TryParse(sessionTokenSegments[0], out globalLSN))
    {
        sessionTokenInVectorFormat = string.Format("{0}:-2#{1}", items[0], globalLSN);
    }
    else
    {
        throw new ArgumentException("Invalid session token format", sessionTokenToConvert);
    }
}
else
{
    // returning the same token since it already has the correct format
    sessionTokenInVectorFormat = sessionTokenToConvert;
}
```

Next steps

Read the following articles:

- Use session tokens to manage consistency in Azure Cosmos DB
- Choose the right consistency level in Azure Cosmos DB
- Consistency, availability, and performance tradeoffs in Azure Cosmos DB
- Availability and performance tradeoffs for various consistency levels

Provision standard (manual) throughput on an Azure Cosmos container - SQL API

11/2/2020 • 2 minutes to read • Edit Online

APPLIES TO: SQL API

This article explains how to provision standard (manual) throughput on a container in Azure Cosmos DB SQL API. You can provision throughput on a single container, or provision throughput on a database and share it among the containers within the database. You can provision throughput on a container using Azure portal, Azure CLI, or Azure Cosmos DB SDKs.

If you are using a different API, see API for MongoDB, Cassandra API, Gremlin API articles to provision the throughput.

Azure portal

1. Sign in to the Azure portal.
2. Create a new Azure Cosmos account, or select an existing Azure Cosmos account.
3. Open the **Data Explorer** pane, and select **New Container**. Next, provide the following details:
   - Indicate whether you are creating a new database or using an existing one.
   - Enter a Container ID.
   - Enter a partition key value (for example, `/ItemID`).
   - Enter a throughput that you want to provision (for example, 1000 RUs).
   - Select **OK**.

Azure CLI or PowerShell

To create a container with dedicated throughput see,

- Create a container using Azure CLI
- Create a container using PowerShell

.NET SDK

> **NOTE**
>
> Use the Cosmos SDKs for SQL API to provision throughput for all Cosmos DB APIs, except Cassandra and MongoDB API.

- .NET SDK V2
- .NET SDK V3

```
// Create a container with a partition key and provision throughput of 400 RU/s
DocumentCollection myCollection = new DocumentCollection();
myCollection.Id = "myContainerName";
myCollection.PartitionKey.Paths.Add("/myPartitionKey");

await client.CreateDocumentCollectionAsync(
    UriFactory.CreateDatabaseUri("myDatabaseName"),
    myCollection,
    new RequestOptions { OfferThroughput = 400 });
```

JavaScript SDK

```
// Create a new Client
const client = new CosmosClient({ endpoint, key });

// Create a database
const { database } = await client.databases.createIfNotExists({ id: "databaseId" });

// Create a container with the specified throughput
const { resource } = await database.containers.createIfNotExists({
id: "containerId",
throughput: 1000
});

// To update an existing container or databases throughput, you need to user the offers API
// Get all the offers
const { resources: offers } = await client.offers.readAll().fetchAll();

// Find the offer associated with your container or the database
const offer = offers.find((_offer) => _offer.offerResourceId === resource._rid);

// Change the throughput value
offer.content.offerThroughput = 2000;

// Replace the offer.
await client.offer(offer.id).replace(offer);
```

Next steps

See the following articles to learn about throughput provisioning in Azure Cosmos DB:

- How to provision standard (manual) throughput on a database
- How to provision autoscale throughput on a database
- Request units and throughput in Azure Cosmos DB

Provision standard (manual) throughput on a database in Azure Cosmos DB - SQL API

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: SQL API

This article explains how to provision standard (manual) throughput on a database in Azure Cosmos DB SQL API. You can provision throughput for a single container, or for a database and share the throughput among the containers within it. To learn when to use container level and database level throughput, see the Use cases for provisioning throughput on containers and databases article. You can provision database level throughput by using the Azure portal or Azure Cosmos DB SDKs.

If you are using a different API, see API for MongoDB, Cassandra API, Gremlin API articles to provision the throughput.

Provision throughput using Azure portal

1. Sign in to the Azure portal.
2. Create a new Azure Cosmos account, or select an existing Azure Cosmos account.
3. Open the **Data Explorer** pane, and select **New Database**. Provide the following details:
   - Enter a database ID.
   - Select the **Provision database throughput** option.
   - Enter a throughput (for example, 1000 RUs).
   - Select **OK**.

Provision throughput using Azure CLI or PowerShell

To create a database with shared throughput see,

- Create a database using Azure CLI
- Create a database using PowerShell

Provision throughput using .NET SDK

> **NOTE**
>
> You can use Azure Cosmos SDKs for SQL API to provision throughput for all APIs. You can optionally use the following example for Cassandra API as well.

- .NET SDK V2
- .NET SDK V3

```
//set the throughput for the database
RequestOptions options = new RequestOptions
{
    OfferThroughput = 500
};

//create the database
await client.CreateDatabaseIfNotExistsAsync(
    new Database {Id = databaseName},
    options);
```

Next steps

See the following articles to learn about provisioned throughput in Azure Cosmos DB:

- Globally scale provisioned throughput
- Provision throughput on containers and databases
- How to provision standard (manual) throughput for a container
- How to provision autoscale throughput for a container
- Request units and throughput in Azure Cosmos DB

Provision autoscale throughput on database or container in Azure Cosmos DB - SQL API

11/2/2020 • 5 minutes to read • Edit Online

APPLIES TO: SQL API

This article explains how to provision autoscale throughput on a database or container (collection, graph, or table) in Azure Cosmos DB SQL API. You can enable autoscale on a single container, or provision autoscale throughput on a database and share it among all the containers in the database.

If you are using a different API, see API for MongoDB, Cassandra API, Gremlin API articles to provision the throughput.

Azure portal

Create new database or container with autoscale

1. Sign in to the Azure portal or the Azure Cosmos DB explorer.
2. Navigate to your Azure Cosmos DB account and open the **Data Explorer** tab.
3. Select **New Container.** Enter a name for your database, container, and a partition key. Under **Throughput**, select the **autoscale** option, and set the maximum throughput (RU/s) that you want the database or container to scale to.

4. Select **OK**.

To provision autoscale on shared throughput database, select the **Provision database throughput** option when creating a new database.

Enable autoscale on existing database or container

> **IMPORTANT**
>
> In the current release, the Azure portal is the only way to migrate between autoscale and standard (manual) provisioned throughput.

1. Sign in to the Azure portal or the Azure Cosmos DB explorer.
2. Navigate to your Azure Cosmos DB account and open the **Data Explorer** tab.
3. Select **Scale and Settings** for your container, or **Scale** for your database.
4. Under **Scale**, select the **Autoscale** option and **Save**.

> **NOTE**
>
> When you enable autoscale on an existing database or container, the starting value for max RU/s is determined by the system, based on your current manual provisioned throughput settings and storage. After the operation completes, you can change the max RU/s if needed. Learn more.

Azure Cosmos DB .NET V3 SDK

Use version 3.9 or higher of the Azure Cosmos DB .NET SDK for SQL API to manage autoscale resources.

> **IMPORTANT**
>
> You can use the .NET SDK to create new autoscale resources. The SDK does not support migrating between autoscale and standard (manual) throughput. The migration scenario is currently supported in only the Azure portal.

Create database with shared throughput

```
// Create instance of CosmosClient
CosmosClient cosmosClient = new CosmosClient(Endpoint, PrimaryKey);

// Autoscale throughput settings
ThroughputProperties autoscaleThroughputProperties = ThroughputProperties.CreateAutoscaleThroughput(4000);
//Set autoscale max RU/s

//Create the database with autoscale enabled
database = await cosmosClient.CreateDatabaseAsync(DatabaseName, throughputProperties:
autoscaleThroughputProperties);
```

Create container with dedicated throughput

```
// Get reference to database that container will be created in
Database database = await cosmosClient.GetDatabase("DatabaseName");

// Container and autoscale throughput settings
ContainerProperties autoscaleContainerProperties = new ContainerProperties("ContainerName",
"/partitionKey");
ThroughputProperties autoscaleThroughputProperties = ThroughputProperties.CreateAutoscaleThroughput(4000);
//Set autoscale max RU/s

// Create the container with autoscale enabled
container = await database.CreateContainerAsync(autoscaleContainerProperties,
autoscaleThroughputProperties);
```

Read the current throughput (RU/s)

```
// Get a reference to the resource
Container container = cosmosClient.GetDatabase("DatabaseName").GetContainer("ContainerName");

// Read the throughput on a resource
ThroughputProperties autoscaleContainerThroughput = await container.ReadThroughputAsync(requestOptions:
null);

// The autoscale max throughput (RU/s) of the resource
int? autoscaleMaxThroughput = autoscaleContainerThroughput.AutoscaleMaxThroughput;

// The throughput (RU/s) the resource is currently scaled to
int? currentThroughput = autoscaleContainerThroughput.Throughput;
```

Change the autoscale max throughput (RU/s)

```
// Change the autoscale max throughput (RU/s)
await
container.ReplaceThroughputAsync(ThroughputProperties.CreateAutoscaleThroughput(newAutoscaleMaxThroughput));
```

Azure Cosmos DB Java V4 SDK

You can use version 4.0 or higher of the Azure Cosmos DB Java SDK for SQL API to manage autoscale resources.

> **IMPORTANT**
>
> You can use the Java SDK to create new autoscale resources. The SDK does not support migrating between autoscale and standard (manual) throughput. The migration scenario is currently supported in only the Azure portal.

Create database with shared throughput

- Async

- Sync

```
// Create instance of CosmosClient
CosmosAsyncClient client = new CosmosClientBuilder()
    .setEndpoint(HOST)
    .setKey(MASTER)
    .setConnectionPolicy(CONNECTIONPOLICY)
    .buildAsyncClient();

// Autoscale throughput settings
ThroughputProperties autoscaleThroughputProperties = ThroughputProperties.createAutoscaledThroughput(4000);
//Set autoscale max RU/s

//Create the database with autoscale enabled
CosmosAsyncDatabase database = client.createDatabase(databaseName,
autoscaleThroughputProperties).block().getDatabase();
```

Create container with dedicated throughput

- Async
- Sync

```
// Get reference to database that container will be created in
CosmosAsyncDatabase database = client.createDatabase("DatabaseName").block().getDatabase();

// Container and autoscale throughput settings
CosmosContainerProperties autoscaleContainerProperties = new CosmosContainerProperties("ContainerName",
"/partitionKey");
ThroughputProperties autoscaleThroughputProperties = ThroughputProperties.createAutoscaledThroughput(4000);
//Set autoscale max RU/s

// Create the container with autoscale enabled
CosmosAsyncContainer container = database.createContainer(autoscaleContainerProperties,
autoscaleThroughputProperties, new CosmosContainerRequestOptions())
                                .block()
                                .getContainer();
```

Read the current throughput (RU/s)

- Async
- Sync

```
// Get a reference to the resource
CosmosAsyncContainer container = client.getDatabase("DatabaseName").getContainer("ContainerName");

// Read the throughput on a resource
ThroughputProperties autoscaleContainerThroughput = container.readThroughput().block().getProperties();

// The autoscale max throughput (RU/s) of the resource
int autoscaleMaxThroughput = autoscaleContainerThroughput.getAutoscaleMaxThroughput();

// The throughput (RU/s) the resource is currently scaled to
int currentThroughput = autoscaleContainerThroughput.Throughput;
```

Change the autoscale max throughput (RU/s)

- Async
- Sync

```
// Change the autoscale max throughput (RU/s)
container.replaceThroughput(ThroughputProperties.createAutoscaledThroughput(newAutoscaleMaxThroughput)).bloc
k();
```

Azure Resource Manager

Azure Resource Manager templates can be used to provision autoscale throughput on database or container-level resources for all Azure Cosmos DB APIs. See Azure Resource Manager templates for Azure Cosmos DB for samples.

Azure CLI

Azure CLI can be used to provision autoscale throughput on a database or container-level resources for all Azure Cosmos DB APIs. For samples see Azure CLI Samples for Azure Cosmos DB.

Azure PowerShell

Azure PowerShell can be used to provision autoscale throughput on a database or container-level resources for all Azure Cosmos DB APIs. For samples see Azure PowerShell samples for Azure Cosmos DB.

Next steps

- Learn about the benefits of provisioned throughput with autoscale.
- Learn how to choose between manual and autoscale throughput.
- Review the autoscale FAQ.

Estimate RU/s using the Azure Cosmos DB capacity planner

12/4/2020 • 6 minutes to read • Edit Online

APPLIES TO: SQL API

Configuring your Azure Cosmos databases and containers with the right amount of provisioned throughput, or Request Units (RU/s), for your workload is essential to optimizing cost and performance. This article describes how to use the Azure Cosmos DB capacity planner to get an estimate of the required RU/s and cost of your workload.

How to estimate throughput and cost with Azure Cosmos DB capacity planner

The capacity planner can be used in two modes.

MODE	DESCRIPTION
Basic	Provides a quick, high-level RU/s and cost estimate. This mode assumes the default Azure Cosmos DB settings for indexing policy, consistency, and other parameters. Use basic mode for a quick, high-level estimate when you are evaluating a potential workload to run on Azure Cosmos DB.
Advanced	Provides a more detailed RU/s and cost estimate, with the ability to tune additional settings — indexing policy, consistency level, and other parameters that affect the cost and throughput. Use advanced mode when you are estimating RU/s for a new project or want a more detailed estimate.

Estimate provisioned throughput and cost using basic mode

To get a quick estimate for your workload using the basic mode, navigate to the capacity planner. Enter in the following parameters based on your workload:

INPUT	DESCRIPTION
Number of regions	Azure Cosmos DB is available in all Azure regions. Select the number of regions required for your workload. You can associate any number of regions with your Cosmos account. See global distribution in Azure Cosmos DB for more details.

INPUT	DESCRIPTION
Multi-region writes	If you enable multi-region writes, your application can read and write to any Azure region. If you disable multi-region writes, your application can write data to a single region. Enable multi-region writes if you expect to have an active-active workload that requires low latency writes in different regions. For example, an IOT workload that writes data to the database at high volumes in different regions. Multi-region writes guarantees 99.999% read and write availability. Multi-region writes require more throughput when compared to the single write regions. To learn more, see how RUs are different for single and multiple-write regions article.
Total data stored (per region)	Total estimated data stored in GB in a single region.
Item size	The estimated size of the data item (e.g. document), ranging from 1 KB to 2 MB.
Reads/sec per region	Number of reads expected per second.
Writes/sec per region	Number of writes expected per second.

After filling the required details, select **Calculate**. The **Cost Estimate** tab shows the total cost for storage and provisioned throughput. You can expand the **Show Details** link in this tab to get the breakdown of the throughput required for read and write requests. Each time you change the value of any field, select **Calculate** to re-calculate the estimated cost.

Estimate provisioned throughput and cost using advanced mode

Advanced mode allows you to provide more settings that impact the RU/s estimate. To use this option, navigate to the capacity planner and sign in to the tool with an account you use for Azure. The sign-in option is available at

the right-hand corner.

After you sign in, you can see additional fields compared to the fields in basic mode. Enter the additional parameters based on your workload.

INPUT	DESCRIPTION
API	Azure Cosmos DB is a multi-model and multi-API service. For new workloads, select SQL (Core) API.
Number of regions	Azure Cosmos DB is available in all Azure regions. Select the number of regions required for your workload. You can associate any number of regions with your Cosmos account. See global distribution in Azure Cosmos DB for more details.
Multi-region writes	If you enable multi-region writes, your application can read and write to any Azure region. If you disable multi-region writes, your application can write data to a single region. Enable multi-region writes if you expect to have an active-active workload that requires low latency writes in different regions. For example, an IOT workload that writes data to the database at high volumes in different regions. Multi-region writes guarantees 99.999% read and write availability. Multi-region writes require more throughput when compared to the single write regions. To learn more, see how RUs are different for single and multiple-write regions article.
Default consistency	Azure Cosmos DB supports 5 consistency levels, to allow developers to balance the tradeoff between consistency, availability, and latency tradeoffs. To learn more, see the consistency levels article. By default, Azure Cosmos DB uses session consistency, which guarantees the ability to read your own writes in a session. Choosing strong or bounded staleness will require double the required RU/s for reads, when compared to session, consistent prefix, and eventual consistency. Strong consistency with multi-region writes is not supported and will automatically default to single-region writes with strong consistency.
Indexing policy	By default, Azure Cosmos DB indexes all properties in all items for flexible and efficient queries (maps to the **Automatic** indexing policy). If you choose **off**, none of the properties are indexed. This results in the lowest RU charge for writes. Select **off** policy if you expect to only do point reads (key value lookups) and/or writes, and no queries. Custom indexing policy allows you to include or exclude specific properties from the index for lower write throughput and storage. To learn more, see indexing policy and sample indexing policies articles.
Total data stored (per region)	Total estimated data stored in GB in a single region.

INPUT	DESCRIPTION
Workload mode	Select **Steady** if your workload volume is constant. Select **Variable** if your workload volume changes over time. For example, during a specific day or a month. The following settings are available if you choose the variable workload option: • Percentage of time at peak: Percentage of time in a month where your workload requires peak (highest) throughput. For example, if you have a workload that has high activity during 9am – 6pm weekday business hours, then the percentage of time at peak is: 45 hours at peak / 730 hours / month = ~6%. • Reads/sec per region at peak - Number of reads expected per second at peak. • Writes/sec per region at peak – Number of writes expected per second at peak. • Reads/sec per region off peak – Number of reads expected per second during off peak. • Writes/sec per region off peak – Number of writes expected per second during off peak. With peak and off-peak intervals, you can optimize your cost by programmatically scaling your provisioned throughput up and down accordingly.
Item size	The size of the data item (e.g. document), ranging from 1 KB to 2 MB. You can also **Upload sample (JSON)** document for a more accurate estimate. If your workload has multiple types of items (with different JSON content) in the same container, you can upload multiple JSON documents and get the estimate. Use the **Add new item** button to add multiple sample JSON documents.

You can also use the **Save Estimate** button to download a CSV file containing the current estimate.

The prices shown in the Azure Cosmos DB capacity planner are estimates based on the public pricing rates for throughput and storage. All prices are shown in US dollars. Refer to the Azure Cosmos DB pricing page to see all rates by region.

Estimating throughput for queries

The Azure Cosmos capacity calculator assumes point reads (a read of a single item, e.g. document, by ID and partition key value) and writes for the workload. To estimate the throughput needed for queries, run your query on a representative data set in a Cosmos container and obtain the RU charge. Multiply the RU charge by the number of queries that you anticipate to run per second to get the total RU/s required.

For example, if your workload requires a query, `SELECT * FROM c WHERE c.id = 'Alice'` that is run 100 times per second, and the RU charge of the query is 10 RUs, you will need 100 query / sec * 10 RU / query = 1000 RU/s in total to serve these requests. Add these RU/s to the RU/s required for any reads or writes happening in your workload.

Next steps

- Learn more about Azure Cosmos DB's pricing model.
- Create a new Cosmos account, database, and container.
- Learn how to optimize provisioned throughput cost.
- Learn how to optimize cost with reserved capacity.

Scale Azure Cosmos DB throughput by using Azure Functions Timer trigger

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: SQL API

The performance of an Azure Cosmos account is based on the amount of provisioned throughput expressed in Request Units per second (RU/s). The provisioning is at a second granularity and is billed based upon the highest RU/s per hour. This provisioned capacity model enables the service to provide a predictable and consistent throughput, guaranteed low latency, and high availability. Most production workloads these features. However, in development and testing environments where Azure Cosmos DB is only used during working hours, you can scale up the throughput in the morning and scale back down in the evening after working hours.

You can set the throughput via Azure Resource Manager Templates, Azure CLI, and PowerShell, for Core (SQL) API accounts, or by using the language-specific Azure Cosmos DB SDKs. The benefit of using Resource Manager Templates, Azure CLI or PowerShell is that they support all Azure Cosmos DB model APIs.

Throughput scheduler sample project

To simplify the process to scale Azure Cosmos DB on a schedule we've created a sample project called Azure Cosmos throughput scheduler. This project is an Azure Functions app with two timer triggers- "ScaleUpTrigger" and "ScaleDownTrigger". The triggers run a PowerShell script that sets the throughput on each resource as defined in the `resources.json` file in each trigger. The ScaleUpTrigger is configured to run at 8 AM UTC and the ScaleDownTrigger is configured to run at 6 PM UTC and these times can be easily updated within the `function.json` file for each trigger.

You can clone this project locally, modify it to specify the Azure Cosmos DB resources to scale up and down and the schedule to run. Later you can deploy it in an Azure subscription and secure it using managed service identity with Azure role-based access control (Azure RBAC) permissions with the "Azure Cosmos DB operator" role to set throughput on your Azure Cosmos accounts.

Next Steps

- Learn more and download the sample from Azure Cosmos DB throughput scheduler.

Find the request unit charge for operations executed in Azure Cosmos DB SQL API

11/2/2020 • 4 minutes to read • Edit Online

APPLIES TO: SQL API

Azure Cosmos DB supports many APIs, such as SQL, MongoDB, Cassandra, Gremlin, and Table. Each API has its own set of database operations. These operations range from simple point reads and writes to complex queries. Each database operation consumes system resources based on the complexity of the operation.

The cost of all database operations is normalized by Azure Cosmos DB and is expressed by Request Units (or RUs, for short). You can think of RUs as a performance currency abstracting the system resources such as CPU, IOPS, and memory that are required to perform the database operations supported by Azure Cosmos DB. No matter which API you use to interact with your Azure Cosmos container, costs are always measured by RUs. Whether the database operation is a write, point read, or query, costs are always measured in RUs. To learn more, see the request units and it's considerations article.

This article presents the different ways you can find the request unit (RU) consumption for any operation executed against a container in Azure Cosmos DB SQL API. If you are using a different API, see API for MongoDB, Cassandra API, Gremlin API, and Table API articles to find the RU/s charge.

Currently, you can measure this consumption only by using the Azure portal or by inspecting the response sent back from Azure Cosmos DB through one of the SDKs. If you're using the SQL API, you have multiple options for finding the RU consumption for an operation against an Azure Cosmos container.

Use the Azure portal

1. Sign in to the Azure portal.
2. Create a new Azure Cosmos account and feed it with data, or selectan existing Azure Cosmos account that already contains data.
3. Go to the **Data Explorer** pane, and then select the container you want to work on.
4. Select **New SQL Query**.
5. Enter a valid query, and then select **Execute Query**.
6. Select **Query Stats** to display the actual request charge for the request you executed.

Use the .NET SDK

- .NET SDK V2
- .NET SDK V3

Objects that are returned from the .NET SDK v2 expose a `RequestCharge` property:

```
ResourceResponse<Document> fetchDocumentResponse = await client.ReadDocumentAsync(
    UriFactory.CreateDocumentUri("database", "container", "itemId"),
    new RequestOptions
    {
        PartitionKey = new PartitionKey("partitionKey")
    });
var requestCharge = fetchDocumentResponse.RequestCharge;

StoredProcedureResponse<string> storedProcedureCallResponse = await
client.ExecuteStoredProcedureAsync<string>(
    UriFactory.CreateStoredProcedureUri("database", "container", "storedProcedureId"),
    new RequestOptions
    {
        PartitionKey = new PartitionKey("partitionKey")
    });
requestCharge = storedProcedureCallResponse.RequestCharge;

IDocumentQuery<dynamic> query = client.CreateDocumentQuery(
    UriFactory.CreateDocumentCollectionUri("database", "container"),
    "SELECT * FROM c",
    new FeedOptions
    {
        PartitionKey = new PartitionKey("partitionKey")
    }).AsDocumentQuery();
while (query.HasMoreResults)
{
    FeedResponse<dynamic> queryResponse = await query.ExecuteNextAsync<dynamic>();
    requestCharge = queryResponse.RequestCharge;
}
```

Use the Java SDK

Objects that are returned from the Java SDK expose a `getRequestCharge()` method:

```
RequestOptions requestOptions = new RequestOptions();
requestOptions.setPartitionKey(new PartitionKey("partitionKey"));

Observable<ResourceResponse<Document>> readDocumentResponse =
client.readDocument(String.format("/dbs/%s/colls/%s/docs/%s", "database", "container", "itemId"),
requestOptions);
readDocumentResponse.subscribe(result -> {
    double requestCharge = result.getRequestCharge();
});

Observable<StoredProcedureResponse> storedProcedureResponse =
client.executeStoredProcedure(String.format("/dbs/%s/colls/%s/sprocs/%s", "database", "container",
"storedProcedureId"), requestOptions, null);
storedProcedureResponse.subscribe(result -> {
    double requestCharge = result.getRequestCharge();
});

FeedOptions feedOptions = new FeedOptions();
feedOptions.setPartitionKey(new PartitionKey("partitionKey"));

Observable<FeedResponse<Document>> feedResponse = client
    .queryDocuments(String.format("/dbs/%s/colls/%s", "database", "container"), "SELECT * FROM c",
feedOptions);
feedResponse.forEach(result -> {
    double requestCharge = result.getRequestCharge();
});
```

For more information, see Quickstart: Build a Java application by using an Azure Cosmos DB SQL API account.

Use the Node.js SDK

Objects that are returned from the Node.js SDK expose a `headers` subobject that maps all the headers returned by the underlying HTTP API. The request charge is available under the `x-ms-request-charge` key:

```
const item = await client
    .database('database')
    .container('container')
    .item('itemId', 'partitionKey')
    .read();
var requestCharge = item.headers['x-ms-request-charge'];

const storedProcedureResult = await client
    .database('database')
    .container('container')
    .storedProcedure('storedProcedureId')
    .execute({
        partitionKey: 'partitionKey'
    });
requestCharge = storedProcedureResult.headers['x-ms-request-charge'];

const query = client.database('database')
    .container('container')
    .items
    .query('SELECT * FROM c', {
        partitionKey: 'partitionKey'
    });
while (query.hasMoreResults()) {
    var result = await query.executeNext();
    requestCharge = result.headers['x-ms-request-charge'];
}
```

For more information, see Quickstart: Build a Node.js app by using an Azure Cosmos DB SQL API account.

Use the Python SDK

The `CosmosClient` object from the Python SDK exposes a `last_response_headers` dictionary that maps all the headers returned by the underlying HTTP API for the last operation executed. The request charge is available under the `x-ms-request-charge` key:

```
response = client.ReadItem(
    'dbs/database/colls/container/docs/itemId', {'partitionKey': 'partitionKey'})
request_charge = client.last_response_headers['x-ms-request-charge']

response = client.ExecuteStoredProcedure(
    'dbs/database/colls/container/sprocs/storedProcedureId', None, {'partitionKey': 'partitionKey'})
request_charge = client.last_response_headers['x-ms-request-charge']
```

For more information, see Quickstart: Build a Python app by using an Azure Cosmos DB SQL API account.

Next steps

To learn about optimizing your RU consumption, see these articles:

- Request units and throughput in Azure Cosmos DB
- Optimize provisioned throughput cost in Azure Cosmos DB
- Optimize query cost in Azure Cosmos DB
- Globally scale provisioned throughput
- Provision throughput on containers and databases
- Provision throughput for a container
- Monitor and debug with metrics in Azure Cosmos DB

Create a container in Azure Cosmos DB SQL API

11/2/2020 • 2 minutes to read • Edit Online

APPLIES TO: SQL API

This article explains the different ways to create an container in Azure Cosmos DB SQL API. It shows how to create a container using the Azure portal, Azure CLI, PowerShell, or supported SDKs. This article demonstrates how to create a container, specify the partition key, and provision throughput.

This article explains the different ways to create a container in Azure Cosmos DB SQL API. If you are using a different API, see API for MongoDB, Cassandra API, Gremlin API, and Table API articles to create the container.

> **NOTE**
>
> When creating containers, make sure you don't create two containers with the same name but different casing. That's because some parts of the Azure platform are not case-sensitive, and this can result in confusion/collision of telemetry and actions on containers with such names.

Create a container using Azure portal

1. Sign in to the Azure portal.
2. Create a new Azure Cosmos account, or select an existing account.
3. Open the **Data Explorer** pane, and select **New Container**. Next, provide the following details:
   - Indicate whether you are creating a new database or using an existing one.
   - Enter a container ID.
   - Enter a partition key.
   - Enter a throughput to be provisioned (for example, 1000 RUs).
   - Select **OK**.

Create a container using Azure CLI

Create a container with Azure CLI. For a listing of all Azure CLI samples across all Azure Cosmos DB APIs see, Azure CLI samples for Azure Cosmos DB.

Create a container using PowerShell

Create a container with PowerShell. For a listing of all PowerShell samples across all Azure Cosmos DB APIs see, PowerShell Samples

Create a container using .NET SDK

If you encounter timeout exception when creating a collection, do a read operation to validate if the collection was created successfully. The read operation throws an exception until the collection create operation is successful. For the list of status codes supported by the create operation see the HTTP Status Codes for Azure Cosmos DB article.

```
// Create a container with a partition key and provision 1000 RU/s throughput.
DocumentCollection myCollection = new DocumentCollection();
myCollection.Id = "myContainerName";
myCollection.PartitionKey.Paths.Add("/myPartitionKey");

await client.CreateDocumentCollectionAsync(
    UriFactory.CreateDatabaseUri("myDatabaseName"),
    myCollection,
    new RequestOptions { OfferThroughput = 1000 });
```

Next steps

- Partitioning in Azure Cosmos DB
- Request Units in Azure Cosmos DB
- Provision throughput on containers and databases
- Work with Azure Cosmos account

Create containers with large partition key

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: SQL API

Azure Cosmos DB uses hash-based partitioning scheme to achieve horizontal scaling of data. All Azure Cosmos containers created before May 3 2019 use a hash function that computes hash based on the first 100 bytes of the partition key. If there are multiple partition keys that have the same first 100 bytes, then those logical partitions are considered as the same logical partition by the service. This can lead to issues like partition size quota being incorrect, and unique indexes being applied across the partition keys. Large partition keys are introduced to solve this issue. Azure Cosmos DB now supports large partition keys with values up to 2 KB.

Large partition keys are supported by using the functionality of an enhanced version of the hash function, which can generate a unique hash from large partition keys up to 2 KB. This hash version is also recommended for scenarios with high partition key cardinality irrespective of the size of the partition key. A partition key cardinality is defined as the number of unique logical partitions, for example in the order of ~30000 logical partitions in a container. This article describes how to create a container with a large partition key using the Azure portal and different SDKs.

Create a large partition key (Azure portal)

To create a large partition key, when you create a new container using the Azure portal, check the **My partition key is larger than 100-bytes** option. Unselect the checkbox if you don't need large partition keys or if you have applications running on SDKs version older than 1.18.

Create a large partition key (PowerShell)

To create a container with large partition key support see,

- Create an Azure Cosmos container with a large partition key size

Create a large partition key (.Net SDK)

To create a container with a large partition key using the .NET SDK, specify the `PartitionKeyDefinitionVersion.V2` property. The following example shows how to specify the Version property within the PartitionKeyDefinition object and set it to PartitionKeyDefinitionVersion.V2.

- .NET SDK V3
- .NET SDK V2

```
await database.CreateContainerAsync(
    new ContainerProperties(collectionName, $"/longpartitionkey")
    {
        PartitionKeyDefinitionVersion = PartitionKeyDefinitionVersion.V2,
    })
```

Supported SDK versions

The Large partition keys are supported with the following minimum versions of SDKs:

SDK TYPE	MINIMUM VERSION
.Net	1.18
Java sync	2.4.0
Java Async	2.5.0
REST API	version higher than `2017-05-03` by using the `x-ms-version` request header.
Resource Manager template	version 2 by using the `"version":2` property within the `partitionKey` object.

Currently, you cannot use containers with large partition key within in Power BI and Azure Logic Apps. You can use containers without a large partition key from these applications.

Next steps

- Partitioning in Azure Cosmos DB
- Request Units in Azure Cosmos DB
- Provision throughput on containers and databases
- Work with Azure Cosmos account

Query an Azure Cosmos container

12/4/2020 • 5 minutes to read • Edit Online

APPLIES TO: SQL API

This article explains how to query a container (collection, graph, or table) in Azure Cosmos DB. In particular, it covers how in-partition and cross-partition queries work in Azure Cosmos DB.

In-partition query

When you query data from containers, if the query has a partition key filter specified, Azure Cosmos DB automatically optimizes the query. It routes the query to the physical partitions corresponding to the partition key values specified in the filter.

For example, consider the below query with an equality filter on `DeviceId`. If we run this query on a container partitioned on `DeviceId`, this query will filter to a single physical partition.

```
SELECT * FROM c WHERE c.DeviceId = 'XMS-0001'
```

As with the earlier example, this query will also filter to a single partition. Adding the additional filter on `Location` does not change this:

```
SELECT * FROM c WHERE c.DeviceId = 'XMS-0001' AND c.Location = 'Seattle'
```

Here's a query that has a range filter on the partition key and won't be scoped to a single physical partition. In order to be an in-partition query, the query must have an equality filter that includes the partition key:

```
SELECT * FROM c WHERE c.DeviceId > 'XMS-0001'
```

Cross-partition query

The following query doesn't have a filter on the partition key (`DeviceId`). Therefore, it must fan-out to all physical partitions where it is run against each partition's index:

```
SELECT * FROM c WHERE c.Location = 'Seattle`
```

Each physical partition has its own index. Therefore, when you run a cross-partition query on a container, you are effectively running one query *per* physical partition. Azure Cosmos DB will automatically aggregate results across different physical partitions.

The indexes in different physical partitions are independent from one another. There is no global index in Azure Cosmos DB.

Parallel cross-partition query

The Azure Cosmos DB SDKs 1.9.0 and later support parallel query execution options. Parallel cross-partition queries allow you to perform low latency, cross-partition queries.

You can manage parallel query execution by tuning the following parameters:

- **MaxConcurrency**: Sets the maximum number of simultaneous network connections to the container's partitions. If you set this property to `-1`, the SDK manages the degree of parallelism. If the `MaxConcurrency` set to `0`, there is a single network connection to the container's partitions.
- **MaxBufferedItemCount**: Trades query latency versus client-side memory utilization. If this option is omitted or to set to -1, the SDK manages the number of items buffered during parallel query execution.

Because of the Azure Cosmos DB's ability to parallelize cross-partition queries, query latency will generally scale well as the system adds physical partitions. However, RU charge will increase significantly as the total number of physical partitions increases.

When you run a cross-partition query, you are essentially doing a separate query per individual physical partition. While cross-partition queries queries will use the index, if available, they are still not nearly as efficient as in-partition queries.

Useful example

Here's an analogy to better understand cross-partition queries:

Let's imagine you are a delivery driver that has to deliver packages to different apartment complexes. Each apartment complex has a list on the premises that has all of the resident's unit numbers. We can compare each apartment complex to a physical partition and each list to the physical partition's index.

We can compare in-partition and cross-partition queries using this example:

In-partition query

If the delivery driver knows the correct apartment complex (physical partition), then they can immediately drive to the correct building. The driver can check the apartment complex's list of the resident's unit numbers (the index) and quickly deliver the appropriate packages. In this case, the driver does not waste any time or effort driving to an apartment complex to check and see if any package recipients live there.

Cross-partition query (fan-out)

If the delivery driver does not know the correct apartment complex (physical partition), they'll need to drive to every single apartment building and check the list with all of the resident's unit numbers (the index). Once they arrive at each apartment complex, they'll still be able to use the list of the addresses of each resident. However, they will need to check every apartment complex's list, whether any package recipients live there or not. This is how cross-partition queries work. While they can use the index (don't need to knock on every single door), they must separately check the index for every physical partition.

Cross-partition query (scoped to only a few physical partitions)

If the delivery driver knows that all package recipients live within a certain few apartment complexes, they won't need to drive to every single one. While driving to a few apartment complexes will still require more work than visiting just a single building, the delivery driver still saves significant time and effort. If a query has the partition key in its filter with the `IN` keyword, it will only check the relevant physical partition's indexes for data.

Avoiding cross-partition queries

For most containers, it's inevitable that you will have some cross-partition queries. Having some cross-partition queries is ok! Nearly all query operations are supported across partitions (both logical partition keys and physical partitions). Azure Cosmos DB also has many optimizations in the query engine and client SDKs to parallelize query execution across physical partitions.

For most read-heavy scenarios, we recommend simply selecting the most common property in your query filters. You should also make sure your partition key adheres to other partition key selection best practices.

Avoiding cross-partition queries typically only matters with large containers. You are charged a minimum of about 2.5 RU's each time you check a physical partition's index for results, even if no items in the physical partition match the query's filter. As such, if you have only one (or just a few) physical partitions, cross-partition queries will not consume significantly more RU's than in-partition queries.

The number of physical partitions is tied to the amount of provisioned RU's. Each physical partition allows for up to 10,000 provisioned RU's and can store up to 50 GB of data. Azure Cosmos DB will automatically manage physical partitions for you. The number of physical partitions in your container is dependent on your provisioned throughput and consumed storage.

You should try to avoid cross-partition queries if your workload meets the criteria below:

- You plan to have over 30,000 RU's provisioned
- You plan to store over 100 GB of data

Next steps

See the following articles to learn about partitioning in Azure Cosmos DB:

- Partitioning in Azure Cosmos DB
- Synthetic partition keys in Azure Cosmos DB

Migrate non-partitioned containers to partitioned containers

12/4/2020 • 4 minutes to read • Edit Online

APPLIES TO: SQL API

Azure Cosmos DB supports creating containers without a partition key. Currently you can create non-partitioned containers by using Azure CLI and Azure Cosmos DB SDKs (.Net, Java, NodeJs) that have a version less than or equal to 2.x. You cannot create non-partitioned containers using the Azure portal. However, such non-partitioned containers aren't elastic and have fixed storage capacity of 20 GB and throughput limit of 10K RU/s.

The non-partitioned containers are legacy and you should migrate your existing non-partitioned containers to partitioned containers to scale storage and throughput. Azure Cosmos DB provides a system defined mechanism to migrate your non-partitioned containers to partitioned containers. This document explains how all the existing non-partitioned containers are auto-migrated into partitioned containers. You can take advantage of the auto-migration feature only if you are using the V3 version of SDKs in all the languages.

> **NOTE**
>
> Currently, you cannot migrate Azure Cosmos DB MongoDB and Gremlin API accounts by using the steps described in this document.

Migrate container using the system defined partition key

To support the migration, Azure Cosmos DB provides a system defined partition key named `/_partitionkey` on all the containers that don't have a partition key. You cannot change the partition key definition after the containers are migrated. For example, the definition of a container that is migrated to a partitioned container will be as follows:

```
{
    "Id": "CollId"
  "partitionKey": {
    "paths": [
      "/_partitionKey"
    ],
    "kind": "Hash"
  },
}
```

After the container is migrated, you can create documents by populating the `_partitionKey` property along with the other properties of the document. The `_partitionKey` property represents the partition key of your documents.

Choosing the right partition key is important to utilize the provisioned throughput optimally. For more information, see how to choose a partition key article.

> **NOTE**
>
> You can take advantage of system defined partition key only if you are using the latest/V3 version of SDKs in all the languages.

The following example shows a sample code to create a document with the system defined partition key and read

that document:

JSON representation of the document

```
DeviceInformationItem = new DeviceInformationItem
{
   "id": "elevator/PugetSound/Building44/Floor1/1",
   "deviceId": "3cf4c52d-cc67-4bb8-b02f-f6185007a808",
   "_partitionKey": "3cf4c52d-cc67-4bb8-b02f-f6185007a808"
}

public class DeviceInformationItem
{
    [JsonProperty(PropertyName = "id")]
    public string Id { get; set; }

    [JsonProperty(PropertyName = "deviceId")]
    public string DeviceId { get; set; }

    [JsonProperty(PropertyName = "_partitionKey", NullValueHandling = NullValueHandling.Ignore)]
    public string PartitionKey {get {return this.DeviceId; set; }
}

CosmosContainer migratedContainer = database.Containers["testContainer"];

DeviceInformationItem deviceItem = new DeviceInformationItem() {
  Id = "1234",
  DeviceId = "3cf4c52d-cc67-4bb8-b02f-f6185007a808"
}

ItemResponse<DeviceInformationItem > response =
  await migratedContainer.CreateItemAsync<DeviceInformationItem>(
    deviceItem.PartitionKey,
    deviceItem
  );

// Read back the document providing the same partition key
ItemResponse<DeviceInformationItem> readResponse =
  await migratedContainer.ReadItemAsync<DeviceInformationItem>(
    partitionKey:deviceItem.PartitionKey,
    id: device.Id
  );
```

For the complete sample, see the .Net samples GitHub repository.

Migrate the documents

While the container definition is enhanced with a partition key property, the documents within the container aren't auto migrated. Which means the system partition key property `/_partitionKey` path is not automatically added to the existing documents. You need to repartition the existing documents by reading the documents that were created without a partition key and rewrite them back with `_partitionKey` property in the documents.

Access documents that don't have a partition key

Applications can access the existing documents that don't have a partition key by using the special system property called "PartitionKey.None", this is the value of the non-migrated documents. You can use this property in all the CRUD and query operations. The following example shows a sample to read a single Document from the NonePartitionKey.

```
CosmosItemResponse<DeviceInformationItem> readResponse =
await migratedContainer.Items.ReadItemAsync<DeviceInformationItem>(
  partitionKey: PartitionKey.None,
  id: device.Id
);
```

For the complete sample on how to repartition the documents, see the .Net samples GitHub repository.

Compatibility with SDKs

Older version of Azure Cosmos DB SDKs such as V2.x.x and V1.x.x don't support the system defined partition key property. So, when you read the container definition from an older SDK, it doesn't contain any partition key definition and these containers will behave exactly as before. Applications that are built with the older version of SDKs continue to work with non-partitioned as is without any changes.

If a migrated container is consumed by the latest/V3 version of SDK and you start populating the system defined partition key within the new documents, you cannot access (read, update, delete, query) such documents from the older SDKs anymore.

Known issues

Querying for the count of items that were inserted without a partition key by using V3 SDK may involve higher throughput consumption

If you query from the V3 SDK for the items that are inserted by using V2 SDK, or the items inserted by using the V3 SDK with `PartitionKey.None` parameter, the count query may consume more RU/s if the `PartitionKey.None` parameter is supplied in the FeedOptions. We recommend that you don't supply the `PartitionKey.None` parameter if no other items are inserted with a partition key.

If new items are inserted with different values for the partition key, querying for such item counts by passing the appropriate key in `FeedOptions` will not have any issues. After inserting new documents with partition key, if you need to query just the document count without the partition key value, that query may again incur higher RU/s similar to the regular partitioned collections.

Next steps

- Partitioning in Azure Cosmos DB
- Request Units in Azure Cosmos DB
- Provision throughput on containers and databases
- Work with Azure Cosmos account

Configure time to live in Azure Cosmos DB

12/4/2020 • 10 minutes to read • Edit Online

APPLIES TO: SQL API

In Azure Cosmos DB, you can choose to configure Time to Live (TTL) at the container level, or you can override it at an item level after setting for the container. You can configure TTL for a container by using Azure portal or the language-specific SDKs. Item level TTL overrides can be configured by using the SDKs.

> This content is related to Azure Cosmos DB transactional store TTL. If you are looking for analitycal store TTL, that enables NoETL HTAP scenarios through Azure Synapse Link, please click here.

Enable time to live on a container using Azure portal

Use the following steps to enable time to live on a container with no expiration. Enable this to allow TTL to be overridden at the item level. You can also set the TTL by entering a non-zero value for seconds.

1. Sign in to the Azure portal.
2. Create a new Azure Cosmos account or selectan existing account.
3. Open the **Data Explorer** pane.
4. Select an existing container, expand it and modify the following values:
   - Open the **Scale & Settings** window.
   - Under **Setting** find, **Time to Live**.
   - Select **On (no default)** or select **On** and set a TTL value
   - Click **Save** to save the changes.

- When DefaultTimeToLive is null then your Time to Live is Off
- When DefaultTimeToLive is -1 then your Time to Live setting is On (No default)
- When DefaultTimeToLive has any other Int value (except 0) your Time to Live setting is On

Enable time to live on a container using Azure CLI or PowerShell

To create or enable TTL on a container see,

- Create a container with TTL using Azure CLI
- Create a container with TTL using PowerShell

Enable time to live on a container using SDK

.NET SDK

- .NET SDK V2
- .NET SDK V3

.NET SDK V2 (Microsoft.Azure.DocumentDB)

```
// Create a new container with TTL enabled and without any expiration value
DocumentCollection collectionDefinition = new DocumentCollection();
collectionDefinition.Id = "myContainer";
collectionDefinition.PartitionKey.Paths.Add("/myPartitionKey");
collectionDefinition.DefaultTimeToLive = -1; //(never expire by default)

DocumentCollection ttlEnabledCollection = await client.CreateDocumentCollectionAsync(
    UriFactory.CreateDatabaseUri("myDatabaseName"),
    collectionDefinition);
```

Java SDK

- Java SDK V4
- Java SDK V3

Java SDK V4 (Maven com.azure::azure-cosmos)

```
CosmosAsyncContainer container;

// Create a new container with TTL enabled and without any expiration value
CosmosContainerProperties containerProperties = new CosmosContainerProperties("myContainer",
"/myPartitionKey");
containerProperties.setDefaultTimeToLiveInSeconds(-1);
container = database.createContainerIfNotExists(containerProperties, 400).block().getContainer();
```

Set time to live on a container using SDK

To set the time to live on a container, you need to provide a non-zero positive number that indicates the time period in seconds. Based on the configured TTL value, all items in the container after the last modified timestamp of the item `_ts` are deleted. You can optionally set `TimeToLivePropertyPath`, which will use a different property instead of the system-generated `_ts` property to determine which items to delete based on TTL.

.NET SDK

- .NET SDK V2
- .NET SDK V3

.NET SDK V2 (Microsoft.Azure.DocumentDB)

```
// Create a new container with TTL enabled and a 90 day expiration
DocumentCollection collectionDefinition = new DocumentCollection();
collectionDefinition.Id = "myContainer";
collectionDefinition.PartitionKey.Paths.Add("/myPartitionKey");
collectionDefinition.DefaultTimeToLive = 90 * 60 * 60 * 24 // expire all documents after 90 days

DocumentCollection ttlEnabledCollection = await client.CreateDocumentCollectionAsync(
    UriFactory.CreateDatabaseUri("myDatabaseName"),
    collectionDefinition;
```

Java SDK

- Java SDK V4
- Java SDK V3

Java SDK V4 (Maven com.azure::azure-cosmos)

```
CosmosAsyncContainer container;

// Create a new container with TTL enabled with default expiration value
CosmosContainerProperties containerProperties = new CosmosContainerProperties("myContainer",
"/myPartitionKey");
containerProperties.setDefaultTimeToLiveInSeconds(90 * 60 * 60 * 24);
container = database.createContainerIfNotExists(containerProperties, 400).block().getContainer();
```

NodeJS SDK

```
const containerDefinition = {
          id: "sample container1",
        };

async function createcontainerWithTTL(db: Database, containerDefinition: ContainerDefinition, collId: any,
defaultTtl: number) {
      containerDefinition.id = collId;
      containerDefinition.defaultTtl = defaultTtl;
      await db.containers.create(containerDefinition);
}
```

Set time to live on an item

In addition to setting a default time to live on a container, you can set a time to live for an item. Setting time to live at the item level will override the default TTL of the item in that container.

- To set the TTL on an item, you need to provide a non-zero positive number, which indicates the period, in seconds, to expire the item after the last modified timestamp of the item `_ts`.
- If the item doesn't have a TTL field, then by default, the TTL set to the container will apply to the item.
- If TTL is disabled at the container level, the TTL field on the item will be ignored until TTL is re-enabled on the container.

Azure portal

Use the following steps to enable time to live on an item:

1. Sign in to the Azure portal.
2. Create a new Azure Cosmos account or selectan existing account.
3. Open the **Data Explorer** pane.
4. Select an existing container, expand it and modify the following values:
   - Open the **Scale & Settings** window.
   - Under **Setting** find, **Time to Live**.
   - Select **On (no default)** or select **On** and set a TTL value.
   - Click **Save** to save the changes.
5. Next navigate to the item for which you want to set time to live, add the `ttl` property and select **Update**.

```
{
 "id": "1",
 "_rid": "Jic9ANWdO-EFAAAAAAAAAA==",
 "_self": "dbs/Jic9AA==/colls/Jic9ANWdO-E=/docs/Jic9ANWdO-EFAAAAAAAAAA==/",
 "_etag": "\"0d00b23f-0000-0000-0000-5c7712e80000\"",
 "_attachments": "attachments/",
 "ttl": 10,
 "_ts": 1551307496
}
```

.NET SDK (any)

```
// Include a property that serializes to "ttl" in JSON
public class SalesOrder
{
    [JsonProperty(PropertyName = "id")]
    public string Id { get; set; }
    [JsonProperty(PropertyName="cid")]
    public string CustomerId { get; set; }
    // used to set expiration policy
    [JsonProperty(PropertyName = "ttl", NullValueHandling = NullValueHandling.Ignore)]
    public int? ttl { get; set; }

    //...
}
// Set the value to the expiration in seconds
SalesOrder salesOrder = new SalesOrder
{
    Id = "SO05",
    CustomerId = "CO18009186470",
    ttl = 60 * 60 * 24 * 30;  // Expire sales orders in 30 days
};
```

NodeJS SDK

```
const itemDefinition = {
          id: "doc",
          name: "sample Item",
          key: "value",
          ttl: 2
        };
```

Java SDK

- Java SDK V4
- Java SDK V3

Java SDK V4 (Maven com.azure::azure-cosmos)

```
// Include a property that serializes to "ttl" in JSON
public class SalesOrder
{
    private String id;
    private String customerId;
    private Integer ttl;

    public SalesOrder(String id, String customerId, Integer ttl) {
        this.id = id;
        this.customerId = customerId;
        this.ttl = ttl;
    }

    public String getId() {return this.id;}
    public void setId(String new_id) {this.id = new_id;}
    public String getCustomerId() {return this.customerId;}
    public void setCustomerId(String new_cid) {this.customerId = new_cid;}
    public Integer getTtl() {return this.ttl;}
    public void setTtl(Integer new_ttl) {this.ttl = new_ttl;}

    //...
}

// Set the value to the expiration in seconds
SalesOrder salesOrder = new SalesOrder(
    "SO05",
    "CO18009186470",
    60 * 60 * 24 * 30  // Expire sales orders in 30 days
);
```

Reset time to live

You can reset the time to live on an item by performing a write or update operation on the item. The write or update operation will set the `_ts` to the current time, and the TTL for the item to expire will begin again. If you wish to change the TTL of an item, you can update the field just as you update any other field.

.NET SDK

- .NET SDK V2
- .NET SDK V3

.NET SDK V2 (Microsoft.Azure.DocumentDB)

```
// This examples leverages the Sales Order class above.
// Read a document, update its TTL, save it.
response = await client.ReadDocumentAsync(
    "/dbs/salesdb/colls/orders/docs/SO05"),
    new RequestOptions { PartitionKey = new PartitionKey("CO18009186470") });

Document readDocument = response.Resource;
readDocument.ttl = 60 * 30 * 30; // update time to live
response = await client.ReplaceDocumentAsync(readDocument);
```

Java SDK

- Java SDK V4
- Java SDK V3

Java SDK V4 (Maven com.azure::azure-cosmos)

```
// This examples leverages the Sales Order class above.
// Read a document, update its TTL, save it.
CosmosAsyncItemResponse<SalesOrder> itemResponse = container.readItem("SO05", new
PartitionKey("CO18009186470"), SalesOrder.class)
        .flatMap(readResponse -> {
            SalesOrder salesOrder = readResponse.getItem();
            salesOrder.setTtl(60 * 30 * 30);
            return container.createItem(salesOrder);
}).block();
```

Turn off time to live

If time to live has been set on an item and you no longer want that item to expire, then you can get the item, remove the TTL field, and replace the item on the server. When the TTL field is removed from the item, the default TTL value assigned to the container is applied to the item. Set the TTL value to -1 to prevent an item from expiring and to not inherit the TTL value from the container.

.NET SDK

- .NET SDK V2
- .NET SDK V3

.NET SDK V2 (Microsoft.Azure.DocumentDB)

```
// This examples leverages the Sales Order class above.
// Read a document, turn off its override TTL, save it.
response = await client.ReadDocumentAsync(
    "/dbs/salesdb/colls/orders/docs/SO05"),
    new RequestOptions { PartitionKey = new PartitionKey("CO18009186470") });

Document readDocument = response.Resource;
readDocument.ttl = null; // inherit the default TTL of the container

response = await client.ReplaceDocumentAsync(readDocument);
```

Java SDK

- Java SDK V4
- Java SDK V3

Java SDK V4 (Maven com.azure::azure-cosmos)

```
// This examples leverages the Sales Order class above.
// Read a document, update its TTL, save it.
CosmosAsyncItemResponse<SalesOrder> itemResponse = container.readItem("SO05", new
PartitionKey("CO18009186470"), SalesOrder.class)
        .flatMap(readResponse -> {
            SalesOrder salesOrder = readResponse.getItem();
            salesOrder.setTtl(null);
            return container.createItem(salesOrder);
}).block();
```

Disable time to live

To disable time to live on a container and stop the background process from checking for expired items, the `DefaultTimeToLive` property on the container should be deleted. Deleting this property is different from setting it to -1. When you set it to -1, new items added to the container will live forever, however you can override this value on specific items in the container. When you remove the TTL property from the container the items will never

expire, even if there are they have explicitly overridden the previous default TTL value.

.NET SDK

- .NET SDK V2
- .NET SDK V3

.NET SDK V2 (Microsoft.Azure.DocumentDB)

```
// Get the container, update DefaultTimeToLive to null
DocumentCollection collection = await client.ReadDocumentCollectionAsync("/dbs/salesdb/colls/orders");
// Disable TTL
collection.DefaultTimeToLive = null;
await client.ReplaceDocumentCollectionAsync(collection);
```

Java SDK

- Java SDK V4
- Java SDK V3

Java SDK V4 (Maven com.azure::azure-cosmos)

```
CosmosContainerProperties containerProperties = new CosmosContainerProperties("myContainer",
"/myPartitionKey");
// Disable TTL
containerProperties.setDefaultTimeToLiveInSeconds(null);
// Update container settings
container.replace(containerProperties).block();
```

Next steps

Learn more about time to live in the following article:

- Time to live

Define unique keys for an Azure Cosmos container

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: SQL API

This article presents the different ways to define unique keys when creating an Azure Cosmos container. It's currently possible to perform this operation either by using the Azure portal or through one of the SDKs.

Use the Azure portal

1. Sign in to the Azure portal.
2. Create a new Azure Cosmos account or selectan existing one.
3. Open the **Data Explorer** pane and select the container that you want to work on.
4. Click on **New Container**.
5. In the **Add Container** dialog, click on **+ Add unique key** to add a unique key entry.
6. Enter the path(s) of the unique key constraint
7. If needed, add more unique key entries by clicking on **+ Add unique key**

Use PowerShell

To create a container with unique keys see, Create an Azure Cosmos container with unique key and TTL

Use the .NET SDK

- .NET SDK V2
- .NET SDK V3

When creating a new container using the .NET SDK v2, a `UniqueKeyPolicy` object can be used to define unique key constraints.

```
client.CreateDocumentCollectionAsync(UriFactory.CreateDatabaseUri("database"), new DocumentCollection
{
    Id = "container",
    PartitionKey = new PartitionKeyDefinition { Paths = new Collection<string>(new List<string> {
"/myPartitionKey" }) },
    UniqueKeyPolicy = new UniqueKeyPolicy
    {
        UniqueKeys = new Collection<UniqueKey>(new List<UniqueKey>
        {
            new UniqueKey { Paths = new Collection<string>(new List<string> { "/firstName", "/lastName",
"/emailAddress" }) },
            new UniqueKey { Paths = new Collection<string>(new List<string> { "/address/zipCode" }) }
        })
    }
});
```

Use the Java SDK

When creating a new container using the Java SDK, a `UniqueKeyPolicy` object can be used to define unique key constraints.

```
// create a new DocumentCollection object
DocumentCollection container = new DocumentCollection();
container.setId("container");

// create array of strings and populate them with the unique key paths
Collection<String> uniqueKey1Paths = new ArrayList<String>();
uniqueKey1Paths.add("/firstName");
uniqueKey1Paths.add("/lastName");
uniqueKey1Paths.add("/emailAddress");
Collection<String> uniqueKey2Paths = new ArrayList<String>();
uniqueKey2Paths.add("/address/zipCode");

// create UniqueKey objects and set their paths
UniqueKey uniqueKey1 = new UniqueKey();
UniqueKey uniqueKey2 = new UniqueKey();
uniqueKey1.setPaths(uniqueKey1Paths);
uniqueKey2.setPaths(uniqueKey2Paths);

// create a new UniqueKeyPolicy object and set its unique keys
UniqueKeyPolicy uniqueKeyPolicy = new UniqueKeyPolicy();
Collection<UniqueKey> uniqueKeys = new ArrayList<UniqueKey>();
uniqueKeys.add(uniqueKey1);
uniqueKeys.add(uniqueKey2);
uniqueKeyPolicy.setUniqueKeys(uniqueKeys);

// set the unique key policy
container.setUniqueKeyPolicy(uniqueKeyPolicy);

// create the container
client.createCollection(String.format("/dbs/%s", "database"), container, null);
```

Use the Node.js SDK

When creating a new container using the Node.js SDK, a `UniqueKeyPolicy` object can be used to define unique key constraints.

```
client.database('database').containers.create({
    id: 'container',
    uniqueKeyPolicy: {
        uniqueKeys: [
            { paths: ['/firstName', '/lastName', '/emailAddress'] },
            { paths: ['/address/zipCode'] }
        ]
    }
});
```

Use the Python SDK

When creating a new container using the Python SDK, unique key constraints can be specified as part of the dictionary passed as parameter.

```
client.CreateContainer('dbs/' + config['DATABASE'], {
    'id': 'container',
    'uniqueKeyPolicy': {
        'uniqueKeys': [
            {'paths': ['/firstName', '/lastName', '/emailAddress']},
            {'paths': ['/address/zipCode']}
        ]
    }
})
```

Next steps

- Learn more about partitioning
- Explore how indexing works

Transactional batch operations in Azure Cosmos DB using the .NET SDK

12/4/2020 • 3 minutes to read • Edit Online

APPLIES TO: SQL API

Transactional batch describes a group of point operations that need to either succeed or fail together with the same partition key in a container. In the .NET SDK, the `TranscationalBatch` class is used to define this batch of operations. If all operations succeed in the order they are described within the transactional batch operation, the transaction will be committed. However, if any operation fails, the entire transaction is rolled back.

What's a transaction in Azure Cosmos DB

A transaction in a typical database can be defined as a sequence of operations performed as a single logical unit of work. Each transaction provides ACID (Atomicity, Consistency, Isolation, Durability) property guarantees.

- **Atomicity** guarantees that all the operations done inside a transaction are treated as a single unit, and either all of them are committed or none of them are.
- **Consistency** makes sure that the data is always in a valid state across transactions.
- **Isolation** guarantees that no two transactions interfere with each other – many commercial systems provide multiple isolation levels that can be used based on the application needs.
- **Durability** ensures that any change that is committed in a database will always be present. Azure Cosmos DB supports full ACID compliant transactions with snapshot isolation for operations within the same logical partition key.

Transactional batch operations Vs stored procedures

Azure Cosmos DB currently supports stored procedures, which also provide the transactional scope on operations. However, transactional batch operations offer the following benefits:

- **Language option** – Transactional batch is supported on the SDK and language you work with already, while stored procedures need to be written in JavaScript.
- **Code versioning** – Versioning application code and onboarding it onto your CI/CD pipeline is much more natural than orchestrating the update of a stored procedure and making sure the rollover happens at the right time. It also makes rolling back changes easier.
- **Performance** – Reduced the latency on equivalent operations up to 30% when compared to the stored procedure execution.
- **Content serialization** – Each operation within a transactional batch can leverage custom serialization options for its payload.

How to create a transactional batch operation

When creating a transactional batch operation, you begin from a container instance and call `CreateTransactionalBatch`:

```
string partitionKey = "The Family";
ParentClass parent = new ParentClass(){ Id = "The Parent", PartitionKey = partitionKey, Name = "John", Age =
30 };
ChildClass child = new ChildClass(){ Id = "The Child", ParentId = parent.Id, PartitionKey = partitionKey };
TransactionalBatch batch = container.CreateTransactionalBatch(new PartitionKey(parent.PartitionKey))
  .CreateItem<ParentClass>(parent)
  .CreateItem<ChildClass>(child);
```

Next, you'll need to call the `ExecuteAsync`:

```
TransactionalBatchResponse batchResponse = await batch.ExecuteAsync();
```

Once the response is received, you'll need to examine if it is successful or not, and extract the results:

```
using (batchResponse)
{
  if (batchResponse.IsSuccessStatusCode)
  {
    TransactionalBatchOperationResult<ParentClass> parentResult =
batchResponse.GetOperationResultAtIndex<ParentClass>(0);
    ParentClass parentClassResult = parentResult.Resource;
    TransactionalBatchOperationResult<ChildClass> childResult =
batchResponse.GetOperationResultAtIndex<ChildClass>(1);
    ChildClass childClassResult = childResult.Resource;
  }
}
```

If there is a failure, the failed operation will have a status code of its corresponding error. Whereas all the other operations will have a 424 status code (failed dependency). In the example below, the operation fails because it tries to create an item that already exists (409 HttpStatusCode.Conflict). Status codes make it easier to identify the cause of transaction failure.

```
// Parent's birthday!
parent.Age = 31;
// Naming two children with the same name, should abort the transaction
ChildClass anotherChild = new ChildClass(){ Id = "The Child", ParentId = parent.Id, PartitionKey =
partitionKey };
TransactionalBatchResponse failedBatchResponse = await container.CreateTransactionalBatch(new
PartitionKey(partitionKey))
  .ReplaceItem<ParentClass>(parent.Id, parent)
  .CreateItem<ChildClass>(anotherChild)
  .ExecuteAsync();

using (failedBatchResponse)
{
  if (!failedBatchResponse.IsSuccessStatusCode)
  {
    TransactionalBatchOperationResult<ParentClass> parentResult =
failedBatchResponse.GetOperationResultAtIndex<ParentClass>(0);
    // parentResult.StatusCode is 424
    TransactionalBatchOperationResult<ChildClass> childResult =
failedBatchResponse.GetOperationResultAtIndex<ChildClass>(1);
    // childResult.StatusCode is 409
  }
}
```

How are transactional batch operations executed

When the `ExecuteAsync` method is called, all operations in the `TransactionalBatch` object are grouped, serialized

into a single payload, and sent as a single request to the Azure Cosmos DB service.

The service receives the request and executes all operations within a transactional scope, and returns a response using the same serialization protocol. This response is either a success, or a failure, and contains all the individual operation responses internally.

The SDK exposes the response for you to verify the result and, optionally, extract each of the internal operation results.

Limitations

Currently, there are two known limits:

- Azure Cosmos DB request size limit specifies the size of the `TransactionalBatch` payload cannot exceed 2 MB, and the maximum execution time is 5 seconds.
- There is a current limit of 100 operations per `TransactionalBatch` to make sure the performance is as expected and within SLAs.

Next steps

- Learn more about what you can do with TransactionalBatch
- Visit our samples for more ways to use our Cosmos DB .NET SDK

Get SQL query execution metrics and analyze query performance using .NET SDK

11/2/2020 • 5 minutes to read • Edit Online

APPLIES TO: SQL API

This article presents how to profile SQL query performance on Azure Cosmos DB. This profiling can be done using `QueryMetrics` retrieved from the .NET SDK and is detailed here. QueryMetrics is a strongly typed object with information about the backend query execution. These metrics are documented in more detail in the Tune Query Performance article.

Set the FeedOptions parameter

All the overloads for DocumentClient.CreateDocumentQuery take in an optional FeedOptions parameter. This option is what allows query execution to be tuned and parameterized.

To collect the Sql query execution metrics, you must set the parameter PopulateQueryMetrics in the FeedOptions to `true`. Setting `PopulateQueryMetrics` to true will make it so that the `FeedResponse` will contain the relevant `QueryMetrics`.

Get query metrics with AsDocumentQuery()

The following code sample shows how to do retrieve metrics when using AsDocumentQuery() method:

```
// Initialize this DocumentClient and Collection
DocumentClient documentClient = null;
DocumentCollection collection = null;

// Setting PopulateQueryMetrics to true in the FeedOptions
FeedOptions feedOptions = new FeedOptions
{
    PopulateQueryMetrics = true
};

string query = "SELECT TOP 5 * FROM c";
IDocumentQuery<dynamic> documentQuery = documentClient.CreateDocumentQuery(Collection.SelfLink, query,
feedOptions).AsDocumentQuery();

while (documentQuery.HasMoreResults)
{
    // Execute one continuation of the query
    FeedResponse<dynamic> feedResponse = await documentQuery.ExecuteNextAsync();

    // This dictionary maps the partitionId to the QueryMetrics of that query
    IReadOnlyDictionary<string, QueryMetrics> partitionIdToQueryMetrics = feedResponse.QueryMetrics;

    // At this point you have QueryMetrics which you can serialize using .ToString()
    foreach (KeyValuePair<string, QueryMetrics> kvp in partitionIdToQueryMetrics)
    {
        string partitionId = kvp.Key;
        QueryMetrics queryMetrics = kvp.Value;

        // Do whatever logging you need
        DoSomeLoggingOfQueryMetrics(query, partitionId, queryMetrics);
    }
}
```

Aggregating QueryMetrics

In the previous section, notice that there were multiple calls to ExecuteNextAsync method. Each call returned a `FeedResponse` object that has a dictionary of `QueryMetrics`; one for every continuation of the query. The following example shows how to aggregate these `QueryMetrics` using LINQ:

```
List<QueryMetrics> queryMetricsList = new List<QueryMetrics>();

while (documentQuery.HasMoreResults)
{
    // Execute one continuation of the query
    FeedResponse<dynamic> feedResponse = await documentQuery.ExecuteNextAsync();

    // This dictionary maps the partitionId to the QueryMetrics of that query
    IReadOnlyDictionary<string, QueryMetrics> partitionIdToQueryMetrics = feedResponse.QueryMetrics;
    queryMetricsList.AddRange(partitionIdToQueryMetrics.Values);
}

// Aggregate the QueryMetrics using the + operator overload of the QueryMetrics class.
QueryMetrics aggregatedQueryMetrics = queryMetricsList.Aggregate((curr, acc) => curr + acc);
Console.WriteLine(aggregatedQueryMetrics);
```

Grouping query metrics by Partition ID

You can group the `QueryMetrics` by the Partition ID. Grouping by Partition ID allows you to see if a specific Partition is causing performance issues when compared to others. The following example shows how to group `QueryMetrics` with LINQ:

```
List<KeyValuePair<string, QueryMetrics>> partitionedQueryMetrics = new List<KeyValuePair<string,
QueryMetrics>>();
while (documentQuery.HasMoreResults)
{
    // Execute one continuation of the query
    FeedResponse<dynamic> feedResponse = await documentQuery.ExecuteNextAsync();

    // This dictionary is maps the partitionId to the QueryMetrics of that query
    IReadOnlyDictionary<string, QueryMetrics> partitionIdToQueryMetrics = feedResponse.QueryMetrics;
    partitionedQueryMetrics.AddRange(partitionIdToQueryMetrics.ToList());
}

// Now we are able to group the query metrics by partitionId
IEnumerable<IGrouping<string, KeyValuePair<string, QueryMetrics>>> groupedByQueryMetrics =
partitionedQueryMetrics
    .GroupBy(kvp => kvp.Key);

// If we wanted to we could even aggregate the groupedby QueryMetrics
foreach(IGrouping<string, KeyValuePair<string, QueryMetrics>> grouping in groupedByQueryMetrics)
{
    string partitionId = grouping.Key;
    QueryMetrics aggregatedQueryMetricsForPartition = grouping
        .Select(kvp => kvp.Value)
        .Aggregate((curr, acc) => curr + acc);
    DoSomeLoggingOfQueryMetrics(query, partitionId, aggregatedQueryMetricsForPartition);
}
```

LINQ on DocumentQuery

You can also get the `FeedResponse` from a LINQ Query using the `AsDocumentQuery()` method:

```
IDocumentQuery<Document> linqQuery = client.CreateDocumentQuery(collection.SelfLink, feedOptions)
    .Take(1)
    .Where(document => document.Id == "42")
    .OrderBy(document => document.Timestamp)
    .AsDocumentQuery();
FeedResponse<Document> feedResponse = await linqQuery.ExecuteNextAsync<Document>();
IReadOnlyDictionary<string, QueryMetrics> queryMetrics = feedResponse.QueryMetrics;
```

Expensive Queries

You can capture the request units consumed by each query to investigate expensive queries or queries that consume high throughput. You can get the request charge by using the RequestCharge property in `FeedResponse`. To learn more about how to get the request charge using the Azure portal and different SDKs, see find the request unit charge article.

```
string query = "SELECT * FROM c";
IDocumentQuery<dynamic> documentQuery = documentClient.CreateDocumentQuery(Collection.SelfLink, query,
feedOptions).AsDocumentQuery();

while (documentQuery.HasMoreResults)
{
    // Execute one continuation of the query
    FeedResponse<dynamic> feedResponse = await documentQuery.ExecuteNextAsync();
    double requestCharge = feedResponse.RequestCharge

    // Log the RequestCharge how ever you want.
    DoSomeLogging(requestCharge);
}
```

Get the query execution time

When calculating the time required to execute a client-side query, make sure that you only include the time to call the `ExecuteNextAsync` method and not other parts of your code base. Just these calls help you in calculating how long the query execution took as shown in the following example:

```
string query = "SELECT * FROM c";
IDocumentQuery<dynamic> documentQuery = documentClient.CreateDocumentQuery(Collection.SelfLink, query,
feedOptions).AsDocumentQuery();
Stopwatch queryExecutionTimeEndToEndTotal = new Stopwatch();
while (documentQuery.HasMoreResults)
{
    // Execute one continuation of the query
    queryExecutionTimeEndToEndTotal.Start();
    FeedResponse<dynamic> feedResponse = await documentQuery.ExecuteNextAsync();
    queryExecutionTimeEndToEndTotal.Stop();
}

// Log the elapsed time
DoSomeLogging(queryExecutionTimeEndToEndTotal.Elapsed);
```

Scan queries (commonly slow and expensive)

A scan query refers to a query that wasn't served by the index, due to which, many documents are loaded before returning the result set.

Below is an example of a scan query:

```
SELECT VALUE c.description
FROM   c
WHERE UPPER(c.description) = "BABYFOOD, DESSERT, FRUIT DESSERT, WITHOUT ASCORBIC ACID, JUNIOR"
```

This query's filter uses the system function UPPER, which isn't served from the index. Executing this query against a large collection produced the following query metrics for the first continuation:

```
QueryMetrics

Retrieved Document Count                 :          60,951
Retrieved Document Size                  :     399,998,938 bytes
Output Document Count                    :               7
Output Document Size                     :             510 bytes
Index Utilization                        :            0.00 %
Total Query Execution Time               :        4,500.34 milliseconds
  Query Preparation Times
    Query Compilation Time               :            0.09 milliseconds
    Logical Plan Build Time              :            0.05 milliseconds
    Physical Plan Build Time             :            0.04 milliseconds
    Query Optimization Time              :            0.01 milliseconds
  Index Lookup Time                      :            0.01 milliseconds
  Document Load Time                     :        4,177.66 milliseconds
  Runtime Execution Times
    Query Engine Times                   :          322.16 milliseconds
    System Function Execution Time       :           85.74 milliseconds
    User-defined Function Execution Time :            0.00 milliseconds
  Document Write Time                    :            0.01 milliseconds
Client Side Metrics
  Retry Count                            :               0
  Request Charge                         :        4,059.95 RUs
```

Note the following values from the query metrics output:

```
Retrieved Document Count                  :          60,951
Retrieved Document Size                   :     399,998,938 bytes
```

This query loaded 60,951 documents, which totaled 399,998,938 bytes. Loading this many bytes results in high cost or request unit charge. It also takes a long time to execute the query, which is clear with the total time spent property:

```
Total Query Execution Time                :        4,500.34 milliseconds
```

Meaning that the query took 4.5 seconds to execute (and this was only one continuation).

To optimize this example query, avoid the use of UPPER in the filter. Instead, when documents are created or updated, the `c.description` values must be inserted in all uppercase characters. The query then becomes:

```
SELECT VALUE c.description
FROM   c
WHERE c.description = "BABYFOOD, DESSERT, FRUIT DESSERT, WITHOUT ASCORBIC ACID, JUNIOR"
```

This query is now able to be served from the index.

To learn more about tuning query performance, see the Tune Query Performance article.

References

- Azure Cosmos DB SQL specification
- ANSI SQL 2011
- JSON
- LINQ

Next steps

- Tune query performance
- Indexing overview
- Azure Cosmos DB .NET samples

Tuning query performance with Azure Cosmos DB

11/2/2020 • 12 minutes to read • Edit Online

APPLIES TO: SQL API

Azure Cosmos DB provides a SQL API for querying data, without requiring schema or secondary indexes. This article provides the following information for developers:

- High-level details on how Azure Cosmos DB's SQL query execution works
- Details on query request and response headers, and client SDK options
- Tips and best practices for query performance
- Examples of how to utilize SQL execution statistics to debug query performance

About SQL query execution

In Azure Cosmos DB, you store data in containers, which can grow to any storage size or request throughput. Azure Cosmos DB seamlessly scales data across physical partitions under the covers to handle data growth or increase in provisioned throughput. You can issue SQL queries to any container using the REST API or one of the supported SQL SDKs.

A brief overview of partitioning: you define a partition key like "city", which determines how data is split across physical partitions. Data belonging to a single partition key (for example, "city" == "Seattle") is stored within a physical partition, but typically a single physical partition has multiple partition keys. When a partition reaches its storage size, the service seamlessly splits the partition into two new partitions, and divides the partition key evenly across these partitions. Since partitions are transient, the APIs use an abstraction of a "partition key range", which denotes the ranges of partition key hashes.

When you issue a query to Azure Cosmos DB, the SDK performs these logical steps:

- Parse the SQL query to determine the query execution plan.
- If the query includes a filter against the partition key, like `SELECT * FROM c WHERE c.city = "Seattle"`, it is routed to a single partition. If the query does not have a filter on partition key, then it is executed in all partitions, and results are merged client side.
- The query is executed within each partition in series or parallel, based on client configuration. Within each partition, the query might make one or more round trips depending on the query complexity, configured page size, and provisioned throughput of the collection. Each execution returns the number of request units consumed by query execution, and optionally, query execution statistics.
- The SDK performs a summarization of the query results across partitions. For example, if the query involves an ORDER BY across partitions, then results from individual partitions are merge-sorted to return results in globally sorted order. If the query is an aggregation like `COUNT`, the counts from individual partitions are summed to produce the overall count.

The SDKs provide various options for query execution. For example, in .NET these options are available in the `FeedOptions` class. The following table describes these options and how they impact query execution time.

OPTION	DESCRIPTION
`EnableCrossPartitionQuery`	Must be set to true for any query that requires to be executed across more than one partition. This is an explicit flag to enable you to make conscious performance tradeoffs during development time.

OPTION	DESCRIPTION
`EnableScanInQuery`	Must be set to true if you have opted out of indexing, but want to run the query via a scan anyway. Only applicable if indexing for the requested filter path is disabled.
`MaxItemCount`	The maximum number of items to return per round trip to the server. By setting to -1, you can let the server manage the number of items. Or, you can lower this value to retrieve only a small number of items per round trip.
`MaxBufferedItemCount`	This is a client-side option, and used to limit the memory consumption when performing cross-partition ORDER BY. A higher value helps reduce the latency of cross-partition sorting.
`MaxDegreeOfParallelism`	Gets or sets the number of concurrent operations run client side during parallel query execution in the Azure Cosmos database service. A positive property value limits the number of concurrent operations to the set value. If it is set to less than 0, the system automatically decides the number of concurrent operations to run.
`PopulateQueryMetrics`	Enables detailed logging of statistics of time spent in various phases of query execution like compilation time, index loop time, and document load time. You can share output from query statistics with Azure Support to diagnose query performance issues.
`RequestContinuation`	You can resume query execution by passing in the opaque continuation token returned by any query. The continuation token encapsulates all state required for query execution.
`ResponseContinuationTokenLimitInKb`	You can limit the maximum size of the continuation token returned by the server. You might need to set this if your application host has limits on response header size. Setting this may increase the overall duration and RUs consumed for the query.

For example, let's take an example query on partition key requested on a collection with `/city` as the partition key and provisioned with 100,000 RU/s of throughput. You request this query using `CreateDocumentQuery<T>` in .NET like the following:

```
IDocumentQuery<dynamic> query = client.CreateDocumentQuery(
    UriFactory.CreateDocumentCollectionUri(DatabaseName, CollectionName),
    "SELECT * FROM c WHERE c.city = 'Seattle'",
    new FeedOptions
    {
        PopulateQueryMetrics = true,
        MaxItemCount = -1,
        MaxDegreeOfParallelism = -1,
        EnableCrossPartitionQuery = true
    }).AsDocumentQuery();

FeedResponse<dynamic> result = await query.ExecuteNextAsync();
```

The SDK snippet shown above, corresponds to the following REST API request:

```
POST https://arramacquerymetrics-westus.documents.azure.com/dbs/db/colls/sample/docs HTTP/1.1
x-ms-continuation:
x-ms-documentdb-isquery: True
x-ms-max-item-count: -1
x-ms-documentdb-query-enablecrosspartition: True
x-ms-documentdb-query-parallelizecrosspartitionquery: True
x-ms-documentdb-query-iscontinuationexpected: True
x-ms-documentdb-populatequerymetrics: True
x-ms-date: Tue, 27 Jun 2017 21:52:18 GMT
authorization: type%3dmaster%26ver%3d1.0%26sig%3drp1Hi83Y8aVV5V6LzZ6xhtQVXRAMz0WNMnUuvriUv%2b4%3d
x-ms-session-token: 7:8,6:2008,5:8,4:2008,3:8,2:2008,1:8,0:8,9:8,8:4008
Cache-Control: no-cache
x-ms-consistency-level: Session
User-Agent: documentdb-dotnet-sdk/1.14.1 Host/32-bit MicrosoftWindowsNT/6.2.9200.0
x-ms-version: 2017-02-22
Accept: application/json
Content-Type: application/query+json
Host: arramacquerymetrics-westus.documents.azure.com
Content-Length: 52
Expect: 100-continue

{"query":"SELECT * FROM c WHERE c.city = 'Seattle'"}
```

Each query execution page corresponds to a REST API `POST` with the `Accept: application/query+json` header, and the SQL query in the body. Each query makes one or more round trips to the server with the `x-ms-continuation` token echoed between the client and server to resume execution. The configuration options in FeedOptions are passed to the server in the form of request headers. For example, `MaxItemCount` corresponds to `x-ms-max-item-count`.

The request returns the following (truncated for readability) response:

```
HTTP/1.1 200 Ok
Cache-Control: no-store, no-cache
Pragma: no-cache
Transfer-Encoding: chunked
Content-Type: application/json
Server: Microsoft-HTTPAPI/2.0
Strict-Transport-Security: max-age=31536000
x-ms-last-state-change-utc: Tue, 27 Jun 2017 21:01:57.561 GMT
x-ms-resource-quota: documentSize=10240;documentsSize=10485760;documentsCount=-1;collectionSize=10485760;
x-ms-resource-usage: documentSize=1;documentsSize=884;documentsCount=2000;collectionSize=1408;
x-ms-item-count: 2000
x-ms-schemaversion: 1.3
x-ms-alt-content-path: dbs/db/colls/sample
x-ms-content-path: +9kEANVq0wA=
x-ms-xp-role: 1
x-ms-documentdb-query-metrics:
totalExecutionTimeInMs=33.67;queryCompileTimeInMs=0.06;queryLogicalPlanBuildTimeInMs=0.02;queryPhysicalPlanBu
ildTimeInMs=0.10;queryOptimizationTimeInMs=0.00;VMExecutionTimeInMs=32.56;indexLookupTimeInMs=0.36;documentLo
adTimeInMs=9.58;systemFunctionExecuteTimeInMs=0.00;userFunctionExecuteTimeInMs=0.00;retrievedDocumentCount=20
00;retrievedDocumentSize=1125600;outputDocumentCount=2000;writeOutputTimeInMs=18.10;indexUtilizationRatio=1.0
0
x-ms-request-charge: 604.42
x-ms-serviceversion: version=1.14.34.4
x-ms-activity-id: 0df8b5f6-83b9-4493-abda-cce6d0f91486
x-ms-session-token: 2:2008
x-ms-gatewayversion: version=1.14.33.2
Date: Tue, 27 Jun 2017 21:59:49 GMT
```

The key response headers returned from the query include the following:

OPTION	DESCRIPTION
`x-ms-item-count`	The number of items returned in the response. This is dependent on the supplied `x-ms-max-item-count`, the number of items that can be fit within the maximum response payload size, the provisioned throughput, and query execution time.
`x-ms-continuation:`	The continuation token to resume execution of the query, if additional results are available.
`x-ms-documentdb-query-metrics`	The query statistics for the execution. This is a delimited string containing statistics of time spent in the various phases of query execution. Returned if `x-ms-documentdb-populatequerymetrics` is set to `True`.
`x-ms-request-charge`	The number of request units consumed by the query.

For details on the REST API request headers and options, see Querying resources using the REST API.

Best practices for query performance

The following are the most common factors that impact Azure Cosmos DB query performance. We dig deeper into each of these topics in this article.

FACTOR	TIP
Provisioned throughput	Measure RU per query, and ensure that you have the required provisioned throughput for your queries.
Partitioning and partition keys	Favor queries with the partition key value in the filter clause for low latency.
SDK and query options	Follow SDK best practices like direct connectivity, and tune client-side query execution options.
Network latency	Account for network overhead in measurement, and use multi-homing APIs to read from the nearest region.
Indexing Policy	Ensure that you have the required indexing paths/policy for the query.
Query execution metrics	Analyze the query execution metrics to identify potential rewrites of query and data shapes.

Provisioned throughput

In Cosmos DB, you create containers of data, each with reserved throughput expressed in request units (RU) per-second. A read of a 1-KB document is 1 RU, and every operation (including queries) is normalized to a fixed number of RUs based on its complexity. For example, if you have 1000 RU/s provisioned for your container, and you have a query like `SELECT * FROM c WHERE c.city = 'Seattle'` that consumes 5 RUs, then you can perform (1000 RU/s) / (5 RU/query) = 200 query/s such queries per second.

If you submit more than 200 queries/sec, the service starts rate-limiting incoming requests above 200/s. The SDKs automatically handle this case by performing a backoff/retry, therefore you might notice a higher latency for these queries. Increasing the provisioned throughput to the required value improves your query latency and

throughput.

To learn more about request units, see Request units.

Partitioning and partition keys

With Azure Cosmos DB, typically queries perform in the following order from fastest/most efficient to slower/less efficient.

- GET on a single partition key and item key
- Query with a filter clause on a single partition key
- Query without an equality or range filter clause on any property
- Query without filters

Queries that need to consult all partitions need higher latency, and can consume higher RUs. Since each partition has automatic indexing against all properties, the query can be served efficiently from the index in this case. You can make queries that span partitions faster by using the parallelism options.

To learn more about partitioning and partition keys, see Partitioning in Azure Cosmos DB.

SDK and query options

See Performance Tips and Performance testing for how to get the best client-side performance from Azure Cosmos DB. This includes using the latest SDKs, configuring platform-specific configurations like default number of connections, frequency of garbage collection, and using lightweight connectivity options like Direct/TCP.

Max Item Count

For queries, the value of `MaxItemCount` can have a significant impact on end-to-end query time. Each round trip to the server will return no more than the number of items in `MaxItemCount` (Default of 100 items). Setting this to a higher value (-1 is maximum, and recommended) will improve your query duration overall by limiting the number of round trips between server and client, especially for queries with large result sets.

```
IDocumentQuery<dynamic> query = client.CreateDocumentQuery(
    UriFactory.CreateDocumentCollectionUri(DatabaseName, CollectionName),
    "SELECT * FROM c WHERE c.city = 'Seattle'",
    new FeedOptions
    {
        MaxItemCount = -1,
    }).AsDocumentQuery();
```

Max Degree of Parallelism

For queries, tune the `MaxDegreeOfParallelism` to identify the best configurations for your application, especially if you perform cross-partition queries (without a filter on the partition-key value). `MaxDegreeOfParallelism` controls the maximum number of parallel tasks, i.e., the maximum of partitions to be visited in parallel.

```
IDocumentQuery<dynamic> query = client.CreateDocumentQuery(
    UriFactory.CreateDocumentCollectionUri(DatabaseName, CollectionName),
    "SELECT * FROM c WHERE c.city = 'Seattle'",
    new FeedOptions
    {
        MaxDegreeOfParallelism = -1,
        EnableCrossPartitionQuery = true
    }).AsDocumentQuery();
```

Let's assume that

- D = Default Maximum number of parallel tasks (= total number of processor in the client machine)
- P = User-specified maximum number of parallel tasks
- N = Number of partitions that needs to be visited for answering a query

Following are implications of how the parallel queries would behave for different values of P.

- (P == 0) => Serial Mode
- (P == 1) => Maximum of one task
- (P > 1) => Min (P, N) parallel tasks
- (P < 1) => Min (N, D) parallel tasks

For SDK release notes, and details on implemented classes and methods see SQL SDKs

Network latency

See Azure Cosmos DB global distribution for how to set up global distribution, and connect to the closest region. Network latency has a significant impact on query performance when you need to make multiple round-trips or retrieve a large result set from the query.

The section on query execution metrics explains how to retrieve the server execution time of queries (`totalExecutionTimeInMs`), so that you can differentiate between time spent in query execution and time spent in network transit.

Indexing policy

See Configuring indexing policy for indexing paths, kinds, and modes, and how they impact query execution. By default, the indexing policy uses Hash indexing for strings, which is effective for equality queries, but not for range queries/order by queries. If you need range queries for strings, we recommend specifying the Range index type for all strings.

By default, Azure Cosmos DB will apply automatic indexing to all data. For high performance insert scenarios, consider excluding paths as this will reduce the RU cost for each insert operation.

Query execution metrics

You can obtain detailed metrics on query execution by passing in the optional `x-ms-documentdb-populatequerymetrics` header (`FeedOptions.PopulateQueryMetrics` in the .NET SDK). The value returned in `x-ms-documentdb-query-metrics` has the following key-value pairs meant for advanced troubleshooting of query execution.

```
IDocumentQuery<dynamic> query = client.CreateDocumentQuery(
    UriFactory.CreateDocumentCollectionUri(DatabaseName, CollectionName),
    "SELECT * FROM c WHERE c.city = 'Seattle'",
    new FeedOptions
    {
        PopulateQueryMetrics = true,
    }).AsDocumentQuery();

FeedResponse<dynamic> result = await query.ExecuteNextAsync();

// Returns metrics by partition key range Id
IReadOnlyDictionary<string, QueryMetrics> metrics = result.QueryMetrics;
```

METRIC	UNIT	DESCRIPTION
`totalExecutionTimeInMs`	milliseconds	Query execution time
`queryCompileTimeInMs`	milliseconds	Query compile time
`queryLogicalPlanBuildTimeInMs`	milliseconds	Time to build logical query plan

METRIC	UNIT	DESCRIPTION
`queryPhysicalPlanBuildTimeInMs`	milliseconds	Time to build physical query plan
`queryOptimizationTimeInMs`	milliseconds	Time spent in optimizing query
`VMExecutionTimeInMs`	milliseconds	Time spent in query runtime
`indexLookupTimeInMs`	milliseconds	Time spent in physical index layer
`documentLoadTimeInMs`	milliseconds	Time spent in loading documents
`systemFunctionExecuteTimeInMs`	milliseconds	Total time spent executing system (built-in) functions in milliseconds
`userFunctionExecuteTimeInMs`	milliseconds	Total time spent executing user-defined functions in milliseconds
`retrievedDocumentCount`	count	Total number of retrieved documents
`retrievedDocumentSize`	bytes	Total size of retrieved documents in bytes
`outputDocumentCount`	count	Number of output documents
`writeOutputTimeInMs`	milliseconds	Query execution time in milliseconds
`indexUtilizationRatio`	ratio (<=1)	Ratio of number of documents matched by the filter to the number of documents loaded

The client SDKs may internally make multiple query operations to serve the query within each partition. The client makes more than one call per-partition if the total results exceed `x-ms-max-item-count`, if the query exceeds the provisioned throughput for the partition, or if the query payload reaches the maximum size per page, or if the query reaches the system allocated timeout limit. Each partial query execution returns a `x-ms-documentdb-query-metrics` for that page.

Here are some sample queries, and how to interpret some of the metrics returned from query execution:

QUERY	SAMPLE METRIC	DESCRIPTION
`SELECT TOP 100 * FROM c`	`"RetrievedDocumentCount": 101`	The number of documents retrieved is 100+1 to match the TOP clause. Query time is mostly spent in `WriteOutputTime` and `DocumentLoadTime` since it is a scan.
`SELECT TOP 500 * FROM c`	`"RetrievedDocumentCount": 501`	RetrievedDocumentCount is now higher (500+1 to match the TOP clause).
`SELECT * FROM c WHERE c.N = 55`	`"IndexLookupTime": "00:00:00.0009500"`	About 0.9 ms is spent in IndexLookupTime for a key lookup, because it's an index lookup on `/N/?`.

QUERY	SAMPLE METRIC	DESCRIPTION
`SELECT * FROM c WHERE c.N > 55`	`"IndexLookupTime": "00:00:00.0017700"`	Slightly more time (1.7 ms) spent in IndexLookupTime over a range scan, because it's an index lookup on `/N/?`.
`SELECT TOP 500 c.N FROM c`	`"IndexLookupTime": "00:00:00.0017700"`	Same time spent on `DocumentLoadTime` as previous queries, but lower `WriteOutputTime` because we're projecting only one property.
`SELECT TOP 500 udf.toPercent(c.N) FROM c`	`"UserDefinedFunctionExecutionTime": "00:00:00.2136500"`	About 213 ms is spent in `UserDefinedFunctionExecutionTime` executing the UDF on each value of `c.N`.
`SELECT TOP 500 c.Name FROM c WHERE STARTSWITH(c.Name, 'Den')`	`"IndexLookupTime": "00:00:00.0006400", "SystemFunctionExecutionTime": "00:00:00.0074100"`	About 0.6 ms is spent in `IndexLookupTime` on `/Name/?`. Most of the query execution time (~7 ms) in `SystemFunctionExecutionTime`.
`SELECT TOP 500 c.Name FROM c WHERE STARTSWITH(LOWER(c.Name), 'den')`	`"IndexLookupTime": "00:00:00", "RetrievedDocumentCount": 2491, "OutputDocumentCount": 500`	Query is performed as a scan because it uses `LOWER`, and 500 out of 2491 retrieved documents are returned.

Next steps

- To learn about the supported SQL query operators and keywords, see SQL query.
- To learn about request units, see request units.
- To learn about indexing policy, see indexing policy

Indexing policies in Azure Cosmos DB

12/4/2020 • 14 minutes to read • Edit Online

APPLIES TO: SQL API

In Azure Cosmos DB, every container has an indexing policy that dictates how the container's items should be indexed. The default indexing policy for newly created containers indexes every property of every item and enforces range indexes for any string or number. This allows you to get good query performance without having to think about indexing and index management upfront.

In some situations, you may want to override this automatic behavior to better suit your requirements. You can customize a container's indexing policy by setting its *indexing mode*, and include or exclude *property paths*.

> **NOTE**
>
> The method of updating indexing policies described in this article only applies to Azure Cosmos DB's SQL (Core) API. Learn about indexing in Azure Cosmos DB's API for MongoDB

Indexing mode

Azure Cosmos DB supports two indexing modes:

- **Consistent**: The index is updated synchronously as you create, update or delete items. This means that the consistency of your read queries will be the consistency configured for the account.
- **None**: Indexing is disabled on the container. This is commonly used when a container is used as a pure key-value store without the need for secondary indexes. It can also be used to improve the performance of bulk operations. After the bulk operations are complete, the index mode can be set to Consistent and then monitored using the IndexTransformationProgress until complete.

> **NOTE**
>
> Azure Cosmos DB also supports a Lazy indexing mode. Lazy indexing performs updates to the index at a much lower priority level when the engine is not doing any other work. This can result in **inconsistent or incomplete** query results. If you plan to query a Cosmos container, you should not select lazy indexing. New containers cannot select lazy indexing. You can request an exemption by contacting Azure support (except if you are using an Azure Cosmos account in serverless mode which doesn't support lazy indexing).

By default, indexing policy is set to `automatic`. It's achieved by setting the `automatic` property in the indexing policy to `true`. Setting this property to `true` allows Azure CosmosDB to automatically index documents as they are written.

Including and excluding property paths

A custom indexing policy can specify property paths that are explicitly included or excluded from indexing. By optimizing the number of paths that are indexed, you can substantially reduce the latency and RU charge of write operations. These paths are defined following the method described in the indexing overview section with the following additions:

- a path leading to a scalar value (string or number) ends with `/?`

- elements from an array are addressed together through the `/[]` notation (instead of `/0`, `/1` etc.)
- the `/*` wildcard can be used to match any elements below the node

Taking the same example again:

```
{
    "locations": [
        { "country": "Germany", "city": "Berlin" },
        { "country": "France", "city": "Paris" }
    ],
    "headquarters": { "country": "Belgium", "employees": 250 }
    "exports": [
        { "city": "Moscow" },
        { "city": "Athens" }
    ]
}
```

- the `headquarters`'s `employees` path is `/headquarters/employees/?`
- the `locations`' `country` path is `/locations/[]/country/?`
- the path to anything under `headquarters` is `/headquarters/*`

For example, we could include the `/headquarters/employees/?` path. This path would ensure that we index the employees property but would not index additional nested JSON within this property.

Include/exclude strategy

Any indexing policy has to include the root path `/*` as either an included or an excluded path.

- Include the root path to selectively exclude paths that don't need to be indexed. This is the recommended approach as it lets Azure Cosmos DB proactively index any new property that may be added to your model.
- Exclude the root path to selectively include paths that need to be indexed.
- For paths with regular characters that include: alphanumeric characters and _ (underscore), you don't have to escape the path string around double quotes (for example, "/path/?"). For paths with other special characters, you need to escape the path string around double quotes (for example, "/"path-abc"/?"). If you expect special characters in your path, you can escape every path for safety. Functionally, it doesn't make any difference if you escape every path Vs just the ones that have special characters.
- The system property `_etag` is excluded from indexing by default, unless the etag is added to the included path for indexing.
- If the indexing mode is set to **consistent**, the system properties `id` and `_ts` are automatically indexed.

When including and excluding paths, you may encounter the following attributes:

- `kind` can be either `range` or `hash`. Hash index support is limited to equality filters. Range index functionality provides all of the functionality of hash indexes as well as efficient sorting, range filters, system functions. We always recommend using a range index.
- `precision` is a number defined at the index level for included paths. A value of `-1` indicates maximum precision. We recommend always setting this value to `-1`.
- `dataType` can be either `String` or `Number`. This indicates the types of JSON properties which will be

indexed.

When not specified, these properties will have the following default values:

PROPERTY NAME	DEFAULT VALUE
`kind`	`range`
`precision`	`-1`
`dataType`	`String` and `Number`

See this section for indexing policy examples for including and excluding paths.

Include/exclude precedence

If your included paths and excluded paths have a conflict, the more precise path takes precedence.

Here's an example:

Included Path: `/food/ingredients/nutrition/*`

Excluded Path: `/food/ingredients/*`

In this case, the included path takes precedence over the excluded path because it is more precise. Based on these paths, any data in the `food/ingredients` path or nested within would be excluded from the index. The exception would be data within the included path: `/food/ingredients/nutrition/*`, which would be indexed.

Here are some rules for included and excluded paths precedence in Azure Cosmos DB:

- Deeper paths are more precise than narrower paths. for example: `/a/b/?` is more precise than `/a/?` .
- The `/?` is more precise than `/*`. For example `/a/?` is more precise than `/a/*` so `/a/?` takes precedence.
- The path `/*` must be either an included path or excluded path.

Spatial indexes

When you define a spatial path in the indexing policy, you should define which index `type` should be applied to that path. Possible types for spatial indexes include:

- Point
- Polygon
- MultiPolygon
- LineString

Azure Cosmos DB, by default, will not create any spatial indexes. If you would like to use spatial SQL built-in functions, you should create a spatial index on the required properties. See this section for indexing policy examples for adding spatial indexes.

Composite indexes

Queries that have an `ORDER BY` clause with two or more properties require a composite index. You can also

define a composite index to improve the performance of many equality and range queries. By default, no composite indexes are defined so you should add composite indexes as needed.

Unlike with included or excluded paths, you can't create a path with the `/*` wildcard. Every composite path has an implicit `/?` at the end of the path that you don't need to specify. Composite paths lead to a scalar value and this is the only value that is included in the composite index.

When defining a composite index, you specify:

- Two or more property paths. The sequence in which property paths are defined matters.
- The order (ascending or descending).

> **NOTE**
>
> When you add a composite index, the query will utilize existing range indexes until the new composite index addition is complete. Therefore, when you add a composite index, you may not immediately observe performance improvements. It is possible to track the progress of index transformation by using one of the SDKs.

ORDER BY queries on multiple properties:

The following considerations are used when using composite indexes for queries with an `ORDER BY` clause with two or more properties:

- If the composite index paths do not match the sequence of the properties in the `ORDER BY` clause, then the composite index can't support the query.
- The order of composite index paths (ascending or descending) should also match the `order` in the `ORDER BY` clause.
- The composite index also supports an `ORDER BY` clause with the opposite order on all paths.

Consider the following example where a composite index is defined on properties name, age, and _ts:

COMPOSITE INDEX	SAMPLE `ORDER BY` QUERY	SUPPORTED BY COMPOSITE INDEX?
`(name ASC, age ASC)`	`SELECT * FROM c ORDER BY c.name ASC, c.age asc`	`Yes`
`(name ASC, age ASC)`	`SELECT * FROM c ORDER BY c.age ASC, c.name asc`	`No`
`(name ASC, age ASC)`	`SELECT * FROM c ORDER BY c.name DESC, c.age DESC`	`Yes`
`(name ASC, age ASC)`	`SELECT * FROM c ORDER BY c.name ASC, c.age DESC`	`No`
`(name ASC, age ASC, timestamp ASC)`	`SELECT * FROM c ORDER BY c.name ASC, c.age ASC, timestamp ASC`	`Yes`
`(name ASC, age ASC, timestamp ASC)`	`SELECT * FROM c ORDER BY c.name ASC, c.age ASC`	`No`

You should customize your indexing policy so you can serve all necessary `ORDER BY` queries.

Queries with filters on multiple properties

If a query has filters on two or more properties, it may be helpful to create a composite index for these

properties.

For example, consider the following query which has an equality filter on two properties:

```
SELECT * FROM c WHERE c.name = "John" AND c.age = 18
```

This query will be more efficient, taking less time and consuming fewer RU's, if it is able to leverage a composite index on (name ASC, age ASC).

Queries with range filters can also be optimized with a composite index. However, the query can only have a single range filter. Range filters include `>`, `<`, `<=`, `>=`, and `!=`. The range filter should be defined last in the composite index.

Consider the following query with both equality and range filters:

```
SELECT * FROM c WHERE c.name = "John" AND c.age > 18
```

This query will be more efficient with a composite index on (name ASC, age ASC). However, the query would not utilize a composite index on (age ASC, name ASC) because the equality filters must be defined first in the composite index.

The following considerations are used when creating composite indexes for queries with filters on multiple properties

- The properties in the query's filter should match those in composite index. If a property is in the composite index but is not included in the query as a filter, the query will not utilize the composite index.
- If a query has additional properties in the filter that were not defined in a composite index, then a combination of composite and range indexes will be used to evaluate the query. This will require fewer RU's than exclusively using range indexes.
- If a property has a range filter (`>`, `<`, `<=`, `>=`, or `!=`), then this property should be defined last in the composite index. If a query has more than one range filter, it will not utilize the composite index.
- When creating a composite index to optimize queries with multiple filters, the `ORDER` of the composite index will have no impact on the results. This property is optional.
- If you do not define a composite index for a query with filters on multiple properties, the query will still succeed. However, the RU cost of the query can be reduced with a composite index.
- Queries with both aggregates (for example, COUNT or SUM) and filters also benefit from composite indexes.

Consider the following examples where a composite index is defined on properties name, age, and timestamp:

COMPOSITE INDEX	SAMPLE QUERY	SUPPORTED BY COMPOSITE INDEX?
`(name ASC, age ASC)`	`SELECT * FROM c WHERE c.name = "John" AND c.age = 18`	`Yes`
`(name ASC, age ASC)`	`SELECT * FROM c WHERE c.name = "John" AND c.age > 18`	`Yes`
`(name ASC, age ASC)`	`SELECT COUNT(1) FROM c WHERE c.name = "John" AND c.age > 18`	`Yes`
`(name DESC, age ASC)`	`SELECT * FROM c WHERE c.name = "John" AND c.age > 18`	`Yes`

COMPOSITE INDEX	SAMPLE QUERY	SUPPORTED BY COMPOSITE INDEX?
`(name ASC, age ASC)`	`SELECT * FROM c WHERE c.name != "John" AND c.age > 18`	`No`
`(name ASC, age ASC, timestamp ASC)`	`SELECT * FROM c WHERE c.name = "John" AND c.age = 18 AND c.timestamp > 123049923`	`Yes`
`(name ASC, age ASC, timestamp ASC)`	`SELECT * FROM c WHERE c.name = "John" AND c.age < 18 AND c.timestamp = 123049923`	`No`

Queries with a filter as well as an ORDER BY clause

If a query filters on one or more properties and has different properties in the ORDER BY clause, it may be helpful to add the properties in the filter to the `ORDER BY` clause.

For example, by adding the properties in the filter to the ORDER BY clause, the following query could be rewritten to leverage a composite index:

Query using range index:

```
SELECT * FROM c WHERE c.name = "John" ORDER BY c.timestamp
```

Query using composite index:

```
SELECT * FROM c WHERE c.name = "John" ORDER BY c.name, c.timestamp
```

The same pattern and query optimizations can be generalized for queries with multiple equality filters:

Query using range index:

```
SELECT * FROM c WHERE c.name = "John", c.age = 18 ORDER BY c.timestamp
```

Query using composite index:

```
SELECT * FROM c WHERE c.name = "John", c.age = 18 ORDER BY c.name, c.age, c.timestamp
```

The following considerations are used when creating composite indexes to optimize a query with a filter and `ORDER BY` clause:

- If the query filters on properties, these should be included first in the `ORDER BY` clause.
- If the query filters on multiple properties, the equality filters must be the first properties in the `ORDER BY` clause
- If you do not define a composite index on a query with a filter on one property and a separate `ORDER BY` clause using a different property, the query will still succeed. However, the RU cost of the query can be reduced with a composite index, particularly if the property in the `ORDER BY` clause has a high cardinality.
- All considerations for creating composite indexes for `ORDER BY` queries with multiple properties as well as queries with filters on multiple properties still apply.

COMPOSITE INDEX	SAMPLE `ORDER BY` QUERY	SUPPORTED BY COMPOSITE INDEX?
`(name ASC, timestamp ASC)`	`SELECT * FROM c WHERE c.name = "John" ORDER BY c.name ASC, c.timestamp ASC`	`Yes`
`(name ASC, timestamp ASC)`	`SELECT * FROM c WHERE c.name = "John" AND c.timestamp > 1589840355 ORDER BY c.name ASC, c.timestamp ASC`	`Yes`
`(timestamp ASC, name ASC)`	`SELECT * FROM c WHERE c.timestamp > 1589840355 AND c.name = "John" ORDER BY c.timestamp ASC, c.name ASC`	`No`
`(name ASC, timestamp ASC)`	`SELECT * FROM c WHERE c.name = "John" ORDER BY c.timestamp ASC, c.name ASC`	`No`
`(name ASC, timestamp ASC)`	`SELECT * FROM c WHERE c.name = "John" ORDER BY c.timestamp ASC`	`No`
`(age ASC, name ASC, timestamp ASC)`	`SELECT * FROM c WHERE c.age = 18 and c.name = "John" ORDER BY c.age ASC, c.name ASC,c.timestamp ASC`	`Yes`
`(age ASC, name ASC, timestamp ASC)`	`SELECT * FROM c WHERE c.age = 18 and c.name = "John" ORDER BY c.timestamp ASC`	`No`

Modifying the indexing policy

A container's indexing policy can be updated at any time by using the Azure portal or one of the supported SDKs. An update to the indexing policy triggers a transformation from the old index to the new one, which is performed online and in-place (so no additional storage space is consumed during the operation). The old indexing policy is efficiently transformed to the new policy without affecting the write availability, read availability, or the throughput provisioned on the container. Index transformation is an asynchronous operation, and the time it takes to complete depends on the provisioned throughput, the number of items and their size.

> **IMPORTANT**
>
> Index transformation is an operation that consumes Request Units. Request Units consumed by an index transformation aren't currently billed if you are using serverless containers. These Request Units will get billed once serverless becomes generally available.

> **NOTE**
>
> It is possible to track the progress of index transformation by using one of the SDKs.

There is no impact to write availability during any index transformations. The index transformation uses your provisioned RUs but at a lower priority than your CRUD operations or queries.

There is no impact to read availability when adding a new index. Queries will only utilize new indexes once the index transformation is complete. During the index transformation, the query engine will continue to

use existing indexes, so you'll observe similar read performance during the indexing transformation to what you had observed before initiating the indexing change. When adding new indexes, there is also no risk of incomplete or inconsistent query results.

When removing indexes and immediately running queries that filter on the dropped indexes, there is not a guarantee of consistent or complete query results. If you remove multiple indexes and do so in one single indexing policy change, the query engine provides consistent and complete results throughout the index transformation. However, if you remove indexes through multiple indexing policy changes, the query engine will not provide consistent or complete results until all index transformations complete. Most developers do not drop indexes and then immediately try to run queries that utilize these indexes so, in practice, this situation is unlikely.

> **NOTE**
> Where possible, you should always try to group multiple indexing changes into one single indexing policy modification

Indexing policies and TTL

Using the Time-to-Live (TTL) feature requires indexing. This means that:

- it is not possible to activate TTL on a container where the indexing mode is set to `none`,
- it is not possible to set the indexing mode to None on a container where TTL is activated.

For scenarios where no property path needs to be indexed, but TTL is required, you can use an indexing policy with an indexing mode set to `consistent`, no included paths, and `/*` as the only excluded path.

Next steps

Read more about the indexing in the following articles:

- Indexing overview
- How to manage indexing policy

Manage indexing policies in Azure Cosmos DB

12/4/2020 • 10 minutes to read • Edit Online

APPLIES TO: SQL API

In Azure Cosmos DB, data is indexed following indexing policies that are defined for each container. The default indexing policy for newly created containers enforces range indexes for any string or number. This policy can be overridden with your own custom indexing policy.

> **NOTE**
> The method of updating indexing policies described in this article only applies to Azure Cosmos DB's SQL (Core) API. Learn about indexing in Azure Cosmos DB's API for MongoDB and Secondary indexing in Azure Cosmos DB Cassandra API.

Indexing policy examples

Here are some examples of indexing policies shown in their JSON format, which is how they are exposed on the Azure portal. The same parameters can be set through the Azure CLI or any SDK.

Opt-out policy to selectively exclude some property paths

```
{
    "indexingMode": "consistent",
    "includedPaths": [
        {
            "path": "/*"
        }
    ],
    "excludedPaths": [
        {
            "path": "/path/to/single/excluded/property/?"
        },
        {
            "path": "/path/to/root/of/multiple/excluded/properties/*"
        }
    ]
}
```

This indexing policy is equivalent to the one below which manually sets `kind`, `dataType`, and `precision` to their default values. These properties are no longer necessary to explicitly set and you should omit them from your indexing policy entirely (as shown in above example).

```
{
    "indexingMode": "consistent",
    "includedPaths": [
        {
            "path": "/*",
            "indexes": [
                {
                    "kind": "Range",
                    "dataType": "Number",
                    "precision": -1
                },
                {
                    "kind": "Range",
                    "dataType": "String",
                    "precision": -1
                }
            ]
        }
    ],
    "excludedPaths": [
        {
            "path": "/path/to/single/excluded/property/?"
        },
        {
            "path": "/path/to/root/of/multiple/excluded/properties/*"
        }
    ]
}
```

Opt-in policy to selectively include some property paths

```
{
    "indexingMode": "consistent",
    "includedPaths": [
        {
            "path": "/path/to/included/property/?"
        },
        {
            "path": "/path/to/root/of/multiple/included/properties/*"
        }
    ],
    "excludedPaths": [
        {
            "path": "/*"
        }
    ]
}
```

This indexing policy is equivalent to the one below which manually sets `kind`, `dataType`, and `precision` to their default values. These properties are no longer necessary to explicitly set and you should omit them from your indexing policy entirely (as shown in above example).

```
{
    "indexingMode": "consistent",
    "includedPaths": [
        {
            "path": "/path/to/included/property/?",
            "indexes": [
                {
                    "kind": "Range",
                    "dataType": "Number"
                },
                {
                    "kind": "Range",
                    "dataType": "String"
                }
            ]
        },
        {
            "path": "/path/to/root/of/multiple/included/properties/*",
            "indexes": [
                {
                    "kind": "Range",
                    "dataType": "Number"
                },
                {
                    "kind": "Range",
                    "dataType": "String"
                }
            ]
        }
    ],
    "excludedPaths": [
        {
            "path": "/*"
        }
    ]
}
```

NOTE

It is generally recommended to use an **opt-out** indexing policy to let Azure Cosmos DB proactively index any new property that may be added to your data model.

Using a spatial index on a specific property path only

```
{
    "indexingMode": "consistent",
    "automatic": true,
    "includedPaths": [
        {
            "path": "/*"
        }
    ],
    "excludedPaths": [
        {
            "path": "/_etag/?"
        }
    ],
    "spatialIndexes": [
        {
            "path": "/path/to/geojson/property/?",
            "types": [
                "Point",
                "Polygon",
                "MultiPolygon",
                "LineString"
            ]
        }
    ]
}
```

Composite indexing policy examples

In addition to including or excluding paths for individual properties, you can also specify a composite index. If you would like to perform a query that has an `ORDER BY` clause for multiple properties, a composite index on those properties is required. Additionally, composite indexes will have a performance benefit for queries that have a multiple filters or both a filter and an ORDER BY clause.

> **NOTE**
>
> Composite paths have an implicit `/?` since only the scalar value at that path is indexed. The `/*` wildcard is not supported in composite paths. You shouldn't specify `/?` or `/*` in a composite path.

Composite index defined for (name asc, age desc):

```
    {
        "automatic":true,
        "indexingMode":"Consistent",
        "includedPaths":[
            {
                "path":"/*"
            }
        ],
        "excludedPaths":[],
        "compositeIndexes":[
            [
                {
                    "path":"/name",
                    "order":"ascending"
                },
                {
                    "path":"/age",
                    "order":"descending"
                }
            ]
        ]
    }
```

The above composite index on name and age is required for Query #1 and Query #2:

Query #1:

```
    SELECT *
    FROM c
    ORDER BY c.name ASC, c.age DESC
```

Query #2:

```
    SELECT *
    FROM c
    ORDER BY c.name DESC, c.age ASC
```

This composite index will benefit Query #3 and Query #4 and optimize the filters:

Query #3:

```
SELECT *
FROM c
WHERE c.name = "Tim"
ORDER BY c.name DESC, c.age ASC
```

Query #4:

```
SELECT *
FROM c
WHERE c.name = "Tim" AND c.age > 18
```

Composite index defined for (name ASC, age ASC) and (name ASC, age DESC):

You can define multiple different composite indexes within the same indexing policy.

```
    {
        "automatic":true,
        "indexingMode":"Consistent",
        "includedPaths":[
            {
                "path":"/*"
            }
        ],
        "excludedPaths":[],
        "compositeIndexes":[
            [
                {
                    "path":"/name",
                    "order":"ascending"
                },
                {
                    "path":"/age",
                    "order":"ascending"
                }
            ],
            [
                {
                    "path":"/name",
                    "order":"ascending"
                },
                {
                    "path":"/age",
                    "order":"descending"
                }
            ]
        ]
    }
```

Composite index defined for (name ASC, age ASC):

It is optional to specify the order. If not specified, the order is ascending.

```
{
        "automatic":true,
        "indexingMode":"Consistent",
        "includedPaths":[
            {
                "path":"/*"
            }
        ],
        "excludedPaths":[],
        "compositeIndexes":[
            [
                {
                    "path":"/name",
                },
                {
                    "path":"/age",
                }
            ]
        ]
}
```

Excluding all property paths but keeping indexing active

This policy can be used in situations where the Time-to-Live (TTL) feature is active but no additional indexes are necessary (to use Azure Cosmos DB as a pure key-value store).

```
{
    "indexingMode": "consistent",
    "includedPaths": [],
    "excludedPaths": [{
        "path": "/*"
    }]
}
```

No indexing

This policy will turn off indexing. If `indexingMode` is set to `none`, you cannot set a TTL on the container.

```
{
    "indexingMode": "none"
}
```

Updating indexing policy

In Azure Cosmos DB, the indexing policy can be updated using any of the below methods:

- from the Azure portal
- using the Azure CLI
- using PowerShell
- using one of the SDKs

An indexing policy update triggers an index transformation. The progress of this transformation can also be tracked from the SDKs.

> **NOTE**
> When updating indexing policy, writes to Azure Cosmos DB will be uninterrupted. Learn more about indexing transformations

Use the Azure portal

Azure Cosmos containers store their indexing policy as a JSON document that the Azure portal lets you directly edit.

1. Sign in to the Azure portal.
2. Create a new Azure Cosmos account or selectan existing account.
3. Open the **Data Explorer** pane and select the container that you want to work on.
4. Click on **Scale & Settings**.
5. Modify the indexing policy JSON document (see examples below)
6. Click **Save** when you are done.

Use the Azure CLI

To create a container with a custom indexing policy see, Create a container with a custom index policy using CLI

Use PowerShell

To create a container with a custom indexing policy see, Create a container with a custom index policy using PowerShell

Use the .NET SDK

- .NET SDK V2
- .NET SDK V3

The `DocumentCollection` object from the .NET SDK v2 exposes an `IndexingPolicy` property that lets you change the `IndexingMode` and add or remove `IncludedPaths` and `ExcludedPaths`.

```
// Retrieve the container's details
ResourceResponse<DocumentCollection> containerResponse = await
client.ReadDocumentCollectionAsync(UriFactory.CreateDocumentCollectionUri("database", "container"));
// Set the indexing mode to consistent
containerResponse.Resource.IndexingPolicy.IndexingMode = IndexingMode.Consistent;
// Add an included path
containerResponse.Resource.IndexingPolicy.IncludedPaths.Add(new IncludedPath { Path = "/*" });
// Add an excluded path
containerResponse.Resource.IndexingPolicy.ExcludedPaths.Add(new ExcludedPath { Path = "/name/*" });
// Add a spatial index
containerResponse.Resource.IndexingPolicy.SpatialIndexes.Add(new SpatialSpec() { Path = "/locations/*",
SpatialTypes = new Collection<SpatialType>() { SpatialType.Point } } );
// Add a composite index
containerResponse.Resource.IndexingPolicy.CompositeIndexes.Add(new Collection<CompositePath> {new
CompositePath() { Path = "/name", Order = CompositePathSortOrder.Ascending }, new CompositePath() { Path =
"/age", Order = CompositePathSortOrder.Descending }});
// Update container with changes
await client.ReplaceDocumentCollectionAsync(containerResponse.Resource);
```

To track the index transformation progress, pass a `RequestOptions` object that sets the `PopulateQuotaInfo` property to `true`.

```
// retrieve the container's details
ResourceResponse<DocumentCollection> container = await
client.ReadDocumentCollectionAsync(UriFactory.CreateDocumentCollectionUri("database", "container"), new
RequestOptions { PopulateQuotaInfo = true });
// retrieve the index transformation progress from the result
long indexTransformationProgress = container.IndexTransformationProgress;
```

Use the Java SDK

The `DocumentCollection` object from the Java SDK (see this Quickstart regarding its usage) exposes `getIndexingPolicy()` and `setIndexingPolicy()` methods. The `IndexingPolicy` object they manipulate lets you change the indexing mode and add or remove included and excluded paths.

```
// Retrieve the container's details
Observable<ResourceResponse<DocumentCollection>> containerResponse =
client.readCollection(String.format("/dbs/%s/colls/%s", "database", "container"), null);
containerResponse.subscribe(result -> {
DocumentCollection container = result.getResource();
IndexingPolicy indexingPolicy = container.getIndexingPolicy();

// Set the indexing mode to consistent
indexingPolicy.setIndexingMode(IndexingMode.Consistent);

// Add an included path

Collection<IncludedPath> includedPaths = new ArrayList<>();
IncludedPath includedPath = new IncludedPath();
includedPath.setPath("/*");
includedPaths.add(includedPath);
indexingPolicy.setIncludedPaths(includedPaths);

// Add an excluded path

Collection<ExcludedPath> excludedPaths = new ArrayList<>();
ExcludedPath excludedPath = new ExcludedPath();
excludedPath.setPath("/name/*");
excludedPaths.add(excludedPath);
indexingPolicy.setExcludedPaths(excludedPaths);

// Add a spatial index

Collection<SpatialSpec> spatialIndexes = new ArrayList<SpatialSpec>();
Collection<SpatialType> collectionOfSpatialTypes = new ArrayList<SpatialType>();

SpatialSpec spec = new SpatialSpec();
spec.setPath("/locations/*");
collectionOfSpatialTypes.add(SpatialType.Point);
spec.setSpatialTypes(collectionOfSpatialTypes);
spatialIndexes.add(spec);

indexingPolicy.setSpatialIndexes(spatialIndexes);

// Add a composite index

Collection<ArrayList<CompositePath>> compositeIndexes = new ArrayList<>();
ArrayList<CompositePath> compositePaths = new ArrayList<>();

CompositePath nameCompositePath = new CompositePath();
nameCompositePath.setPath("/name");
nameCompositePath.setOrder(CompositePathSortOrder.Ascending);

CompositePath ageCompositePath = new CompositePath();
ageCompositePath.setPath("/age");
ageCompositePath.setOrder(CompositePathSortOrder.Descending);

compositePaths.add(ageCompositePath);
compositePaths.add(nameCompositePath);

compositeIndexes.add(compositePaths);
indexingPolicy.setCompositeIndexes(compositeIndexes);

// Update the container with changes

 client.replaceCollection(container, null);
});
```

To track the index transformation progress on a container, pass a `RequestOptions` object that requests the quota info to be populated, then retrieve the value from the `x-ms-documentdb-collection-index-transformation-progress` response header.

```
// set the RequestOptions object
RequestOptions requestOptions = new RequestOptions();
requestOptions.setPopulateQuotaInfo(true);
// retrieve the container's details
Observable<ResourceResponse<DocumentCollection>> containerResponse =
client.readCollection(String.format("/dbs/%s/colls/%s", "database", "container"), requestOptions);
containerResponse.subscribe(result -> {
    // retrieve the index transformation progress from the response headers
    String indexTransformationProgress = result.getResponseHeaders().get("x-ms-documentdb-collection-index-
transformation-progress");
});
```

Use the Node.js SDK

The `ContainerDefinition` interface from Node.js SDK (see this Quickstart regarding its usage) exposes an `indexingPolicy` property that lets you change the `indexingMode` and add or remove `includedPaths` and `excludedPaths`.

Retrieve the container's details

```
const containerResponse = await client.database('database').container('container').read();
```

Set the indexing mode to consistent

```
containerResponse.body.indexingPolicy.indexingMode = "consistent";
```

Add included path including a spatial index

```
containerResponse.body.indexingPolicy.includedPaths.push({
    includedPaths: [
      {
        path: "/age/*",
        indexes: [
          {
            kind: cosmos.DocumentBase.IndexKind.Range,
            dataType: cosmos.DocumentBase.DataType.String
          },
          {
            kind: cosmos.DocumentBase.IndexKind.Range,
            dataType: cosmos.DocumentBase.DataType.Number
          }
        ]
      },
      {
        path: "/locations/*",
        indexes: [
          {
            kind: cosmos.DocumentBase.IndexKind.Spatial,
            dataType: cosmos.DocumentBase.DataType.Point
          }
        ]
      }
    ]
  });
```

Add excluded path

```
containerResponse.body.indexingPolicy.excludedPaths.push({ path: '/name/*' });
```

Update the container with changes

```
const replaceResponse = await
client.database('database').container('container').replace(containerResponse.body);
```

To track the index transformation progress on a container, pass a `RequestOptions` object that sets the `populateQuotaInfo` property to `true`, then retrieve the value from the `x-ms-documentdb-collection-index-transformation-progress` response header.

```
// retrieve the container's details
const containerResponse = await client.database('database').container('container').read({
    populateQuotaInfo: true
});
// retrieve the index transformation progress from the response headers
const indexTransformationProgress = replaceResponse.headers['x-ms-documentdb-collection-index-transformation-
progress'];
```

Use the Python SDK

- Python SDK V3
- Python SDK V4

When using the Python SDK V3 (see this Quickstart regarding its usage), the container configuration is managed as a dictionary. From this dictionary, it is possible to access the indexing policy and all its attributes.

Retrieve the container's details

```
containerPath = 'dbs/database/colls/collection'
container = client.ReadContainer(containerPath)
```

Set the indexing mode to consistent

```
container['indexingPolicy']['indexingMode'] = 'consistent'
```

Define an indexing policy with an included path and a spatial index

```
container["indexingPolicy"] = {

    "indexingMode":"consistent",
    "spatialIndexes":[
                {"path":"/location/*","types":["Point"]}
            ],
    "includedPaths":[{"path":"/age/*","indexes":[]}],
    "excludedPaths":[{"path":"/*"}]
}
```

Define an indexing policy with an excluded path

```
container["indexingPolicy"] = {
    "indexingMode":"consistent",
    "includedPaths":[{"path":"/*","indexes":[]}],
    "excludedPaths":[{"path":"/name/*"}]
}
```

Add a composite index

```
container['indexingPolicy']['compositeIndexes'] = [
                [
                    {
                        "path": "/name",
                        "order": "ascending"
                    },
                    {
                        "path": "/age",
                        "order": "descending"
                    }
                ]
                ]
```

Update the container with changes

```
response = client.ReplaceContainer(containerPath, container)
```

Next steps

Read more about the indexing in the following articles:

- Indexing overview
- Indexing policy

Performance tips for Azure Cosmos DB and .NET

12/4/2020 • 13 minutes to read • Edit Online

APPLIES TO: SQL API

Azure Cosmos DB is a fast, flexible distributed database that scales seamlessly with guaranteed latency and throughput levels. You don't have to make major architecture changes or write complex code to scale your database with Azure Cosmos DB. Scaling up and down is as easy as making a single API call. To learn more, see provision container throughput or provision database throughput.

Because Azure Cosmos DB is accessed via network calls, you can make client-side optimizations to achieve peak performance when you use the SQL .NET SDK.

If you're trying to improve your database performance, consider the options presented in the following sections.

Hosting recommendations

For query-intensive workloads, use Windows 64-bit instead of Linux or Windows 32-bit host processing

We recommend Windows 64-bit host processing for improved performance. The SQL SDK includes a native ServiceInterop.dll to parse and optimize queries locally. ServiceInterop.dll is supported only on the Windows x64 platform.

For Linux and other unsupported platforms where ServiceInterop.dll isn't available, an additional network call is made to the gateway to get the optimized query.

The four application types listed here use 32-bit host processing by default. To change host processing to 64-bit processing for your application type, do the following:

- **For executable applications**: In the **Project Properties** window, on the **Build** pane, set the platform target to **x64**.
- **For VSTest-based test projects**: On the Visual Studio **Test** menu, select **Test** > **Test Settings**, and then set **Default Processor Architecture** to **X64**.
- **For locally deployed ASP.NET web applications**: Select **Tools** > **Options** > **Projects and Solutions** > **Web Projects**, and then select **Use the 64-bit version of IIS Express for web sites and projects**.
- **For ASP.NET web applications deployed on Azure**: In the Azure portal, in **Application settings**, select the **64-bit** platform.

> **NOTE**
>
> By default, new Visual Studio projects are set to **Any CPU**. We recommend that you set your project to **x64** so it doesn't switch to **x86**. A project that's set to **Any CPU** can easily switch to **x86** if an x86-only dependency is added.
> The ServiceInterop.dll file needs to be in the folder that the SDK DLL is being executed from. This should be a concern only if you manually copy DLLs or have custom build or deployment systems.

Turn on server-side garbage collection

Reducing the frequency of garbage collection can help in some cases. In .NET, set gcServer to `true`.

Scale out your client workload

If you're testing at high throughput levels, or at rates that are greater than 50,000 Request Units per second (RU/s), the client application could become a workload bottleneck. This is because the machine might cap out on CPU or network utilization. If you reach this point, you can continue to push the Azure Cosmos DB account further by scaling out your client applications across multiple servers.

> **NOTE**
>
> High CPU usage can cause increased latency and request timeout exceptions.

Networking

Connection policy: Use direct connection mode

.NET V3 SDK default connection mode is direct. You configure the connection mode when you create the `CosmosClient` instance in `CosmosClientOptions`. To learn more about different connectivity options, see the connectivity modes article.

```
string connectionString = "<your-account-connection-string>";
CosmosClient client = new CosmosClient(connectionString,
new CosmosClientOptions
{
    ConnectionMode = ConnectionMode.Gateway // ConnectionMode.Direct is the default
});
```

Ephemeral port exhaustion

If you see a high connection volume or high port usage on your instances, first verify that your client instances are singletons. In other words, the client instances should be unique for the lifetime of the application.

When it's running on the TCP protocol, the client optimizes for latency by using the long-lived connections. This is in contrast with the HTTPS protocol, which terminates the connections after two minutes of inactivity.

In scenarios where you have sparse access, and if you notice a higher connection count when compared to Gateway mode access, you can:

- Configure the CosmosClientOptions.PortReuseMode property to `PrivatePortPool` (effective with framework versions 4.6.1 and later and .NET Core versions 2.0 and later). This property allows the SDK to use a small pool of ephemeral ports for various Azure Cosmos DB destination endpoints.
- Configure the CosmosClientOptions.IdleConnectionTimeout property as greater than or equal to 10 minutes. The recommended values are from 20 minutes to 24 hours.

For performance, collocate clients in the same Azure region

When possible, place any applications that call Azure Cosmos DB in the same region as the Azure Cosmos DB database. Here's an approximate comparison: calls to Azure Cosmos DB within the same region finish within 1 millisecond (ms) to 2 ms, but the latency between the West and East coast of the US is more than 50 ms. This latency can vary from request to request, depending on the route taken by the request as it passes from the client to the Azure datacenter boundary.

You can get the lowest possible latency by ensuring that the calling application is located within the same Azure region as the provisioned Azure Cosmos DB endpoint. For a list of available regions, see Azure regions.

Increase the number of threads/tasks

Because calls to Azure Cosmos DB are made over the network, you might need to vary the degree of concurrency of your requests so that the client application spends minimal time waiting between requests. For example, if you're using the .NET Task Parallel Library, create on the order of hundreds of tasks that read from or write to Azure Cosmos DB.

Enable accelerated networking

To reduce latency and CPU jitter, we recommend that you enable accelerated networking on your client virtual machines. For more information, see Create a Windows virtual machine with accelerated networking or Create a Linux virtual machine with accelerated networking.

SDK usage

Install the most recent SDK

The Azure Cosmos DB SDKs are constantly being improved to provide the best performance. To determine the most recent SDK and review improvements, see Azure Cosmos DB SDK.

Use stream APIs

.NET SDK V3 contains stream APIs that can receive and return data without serializing.

Middle-tier applications that don't consume responses directly from the SDK but relay them to other application tiers can benefit from the stream APIs. For examples of stream handling, see the item management samples.

Use a singleton Azure Cosmos DB client for the lifetime of your application

Each `CosmosClient` instance is thread-safe and performs efficient connection management and address caching when it operates in Direct mode. To allow efficient connection management and better SDK client performance, we recommend that you use a single instance per `AppDomain` for the lifetime of the application.

When you're working on Azure Functions, instances should also follow the existing guidelines and maintain a single instance.

Disable content response on write operations

For workloads that have heavy create payloads, set the `EnableContentResponseOnWrite` request option to `false`. The service will no longer return the created or updated resource to the SDK. Normally, because the application has the object that's being created, it doesn't need the service to return it. The header values are still accessible, like a request charge. Disabling the content response can help improve performance, because the SDK no longer needs to allocate memory or serialize the body of the response. It also reduces the network bandwidth usage to further help performance.

```
ItemRequestOptions requestOptions = new ItemRequestOptions() { EnableContentResponseOnWrite = false };
ItemResponse<Book> itemResponse = await this.container.CreateItemAsync<Book>(book, new
PartitionKey(book.pk), requestOptions);
// Resource will be null
itemResponse.Resource
```

Enable Bulk to optimize for throughput instead of latency

Enable *Bulk* for scenarios where the workload requires a large amount of throughput, and latency is not as important. For more information about how to enable the Bulk feature, and to learn which scenarios it should be used for, see Introduction to Bulk support.

Increase System.Net MaxConnections per host when you use Gateway mode

Azure Cosmos DB requests are made over HTTPS/REST when you use Gateway mode. They're subject to the default connection limit per hostname or IP address. You might need to set `MaxConnections` to a higher value (from 100 through 1,000) so that the client library can use multiple simultaneous connections to Azure Cosmos DB. In .NET SDK 1.8.0 and later, the default value for ServicePointManager.DefaultConnectionLimit is 50. To change the value, you can set `Documents.Client.ConnectionPolicy.MaxConnectionLimit` to a higher value.

Tune parallel queries for partitioned collections

SQL .NET SDK supports parallel queries, which enable you to query a partitioned container in parallel. For more information, see code samples related to working with the SDKs. Parallel queries are designed to provide better query latency and throughput than their serial counterpart.

Parallel queries provide two parameters that you can tune to fit your requirements:

- **MaxConcurrency**: Controls the maximum number of partitions that can be queried in parallel.

  Parallel query works by querying multiple partitions in parallel. But data from an individual partition is fetched serially with respect to the query. Setting `MaxConcurrency` in SDK V3 to the number of partitions has the best chance of achieving the most performant query, provided all other system conditions remain the same. If you don't know the number of partitions, you can set the degree of parallelism to a high number. The system will choose the minimum (number of partitions, user provided input) as the degree of parallelism.

  Parallel queries produce the most benefit if the data is evenly distributed across all partitions with respect to the query. If the partitioned collection is partitioned so that all or most of the data returned by a query is concentrated in a few partitions (one partition is the worst case), those partitions will bottleneck the performance of the query.

- **MaxBufferedItemCount**: Controls the number of pre-fetched results.

  Parallel query is designed to pre-fetch results while the current batch of results is being processed by the client. This pre-fetching helps improve the overall latency of a query. The `MaxBufferedItemCount` parameter limits the number of pre-fetched results. Set `MaxBufferedItemCount` to the expected number of results returned (or a higher number) to allow the query to receive the maximum benefit from pre-fetching.

  Pre-fetching works the same way regardless of the degree of parallelism, and there's a single buffer for the data from all partitions.

Implement backoff at RetryAfter intervals

During performance testing, you should increase load until a small rate of requests are throttled. If requests are throttled, the client application should back off throttling for the server-specified retry interval. Respecting the backoff helps ensure that you'll spend a minimal amount of time waiting between retries.

For more information, see RetryAfter.

There's a mechanism for logging additional diagnostics information and troubleshooting latency issues, as shown in the following sample. You can log the diagnostics string for requests that have a higher read latency. The captured diagnostics string will help you understand how many times you received a *429* error for a given request.

```
ItemResponse<Book> readItemResponse = await this.cosmosContainer.ReadItemAsync<Book>("ItemId", new
PartitionKey("PartitionKeyValue"));
readItemResponse.Diagnostics.ToString();
```

Increase the number of threads/tasks

See Increase the number of threads/tasks in the Networking section of this article.

Indexing policy

Exclude unused paths from indexing for faster writes

The Azure Cosmos DB indexing policy also allows you to specify which document paths to include or exclude from indexing by using indexing paths (IndexingPolicy.IncludedPaths and IndexingPolicy.ExcludedPaths).

Indexing only the paths you need can improve write performance, reduce RU charges on write operations, and reduce index storage for scenarios in which the query patterns are known beforehand. This is because indexing costs correlate directly to the number of unique paths indexed. For example, the following code shows how to exclude an entire section of the documents (a subtree) from indexing by using the "*" wildcard:

```
var containerProperties = new ContainerProperties(id: "excludedPathCollection", partitionKeyPath: "/pk" );
containerProperties.IndexingPolicy.IncludedPaths.Add(new IncludedPath { Path = "/*" });
containerProperties.IndexingPolicy.ExcludedPaths.Add(new ExcludedPath { Path = "/nonIndexedContent/*");
Container container = await this.cosmosDatabase.CreateContainerAsync(containerProperties);
```

For more information, see Azure Cosmos DB indexing policies.

Throughput

Measure and tune for lower RU/s usage

Azure Cosmos DB offers a rich set of database operations. These operations include relational and hierarchical queries with Universal Disk Format (UDF) files, stored procedures, and triggers, all operating on the documents within a database collection.

The costs associated with each of these operations vary depending on the CPU, IO, and memory that are required to complete the operation. Instead of thinking about and managing hardware resources, you can think of a Request Unit as a single measure for the resources that are required to perform various database operations and service an application request.

Throughput is provisioned based on the number of Request Units set for each container. Request Unit consumption is evaluated as a units-per-second rate. Applications that exceed the provisioned Request Unit rate for their container are limited until the rate drops below the provisioned level for the container. If your application requires a higher level of throughput, you can increase your throughput by provisioning additional Request Units.

The complexity of a query affects how many Request Units are consumed for an operation. The number of predicates, the nature of the predicates, the number of UDF files, and the size of the source dataset all influence the cost of query operations.

To measure the overhead of any operation (create, update, or delete), inspect the x-ms-request-charge header (or the equivalent `RequestCharge` property in `ResourceResponse\<T>` or `FeedResponse\<T>` in the .NET SDK) to measure the number of Request Units consumed by the operations:

```
// Measure the performance (Request Units) of writes
ItemResponse<Book> response = await container.CreateItemAsync<Book>(myBook, new
PartitionKey(myBook.PkValue));
Console.WriteLine("Insert of item consumed {0} request units", response.RequestCharge);
// Measure the performance (Request Units) of queries
FeedIterator<Book> queryable = container.GetItemQueryIterator<ToDoActivity>(queryString);
while (queryable.HasMoreResults)
    {
        FeedResponse<Book> queryResponse = await queryable.ExecuteNextAsync<Book>();
        Console.WriteLine("Query batch consumed {0} request units", queryResponse.RequestCharge);
    }
```

The request charge that's returned in this header is a fraction of your provisioned throughput (that is, 2,000 RU/s). For example, if the preceding query returns 1,000 1-KB documents, the cost of the operation is 1,000. So, within one second, the server honors only two such requests before it rate-limits later requests. For more information, see Request Units and the Request Unit calculator.

Handle rate limiting/request rate too large

When a client attempts to exceed the reserved throughput for an account, there's no performance degradation at the server and no use of throughput capacity beyond the reserved level. The server preemptively ends the request with RequestRateTooLarge (HTTP status code 429). It returns an x-ms-retry-after-ms header that indicates the amount of time, in milliseconds, that the user must wait before attempting the request again.

```
    HTTP Status 429,
    Status Line: RequestRateTooLarge
    x-ms-retry-after-ms :100
```

The SDKs all implicitly catch this response, respect the server-specified retry-after header, and retry the request. Unless your account is being accessed concurrently by multiple clients, the next retry will succeed.

If you have more than one client cumulatively operating consistently above the request rate, the default retry count that's currently set to 9 internally by the client might not suffice. In this case, the client throws a CosmosException with status code 429 to the application.

You can change the default retry count by setting the `RetryOptions` on the `CosmosClientOptions` instance. By default, the CosmosException with status code 429 is returned after a cumulative wait time of 30 seconds if the request continues to operate above the request rate. This error is returned even when the current retry count is less than the maximum retry count, whether the current value is the default of 9 or a user-defined value.

The automated retry behavior helps improve resiliency and usability for most applications. But it might not be the best behavior when you're doing performance benchmarks, especially when you're measuring latency. The client-observed latency will spike if the experiment hits the server throttle and causes the client SDK to silently retry. To avoid latency spikes during performance experiments, measure the charge that's returned by each operation, and ensure that requests are operating below the reserved request rate.

For more information, see Request Units.

For higher throughput, design for smaller documents

The request charge (that is, the request-processing cost) of a specified operation correlates directly to the size of the document. Operations on large documents cost more than operations on small documents.

Next steps

For a sample application that's used to evaluate Azure Cosmos DB for high-performance scenarios on a few client machines, see Performance and scale testing with Azure Cosmos DB.

To learn more about designing your application for scale and high performance, see Partitioning and scaling in Azure Cosmos DB.

Performance tips for Azure Cosmos DB and .NET SDK v2

12/4/2020 • 15 minutes to read • Edit Online

APPLIES TO: SQL API

Azure Cosmos DB is a fast and flexible distributed database that scales seamlessly with guaranteed latency and throughput. You don't have to make major architecture changes or write complex code to scale your database with Azure Cosmos DB. Scaling up and down is as easy as making a single API call. To learn more, see how to provision container throughput or how to provision database throughput. But because Azure Cosmos DB is accessed via network calls, there are client-side optimizations you can make to achieve peak performance when you use the SQL .NET SDK.

So, if you're trying to improve your database performance, consider these options:

Upgrade to the .NET V3 SDK

The .NET v3 SDK is released. If you use the .NET v3 SDK, see the .NET v3 performance guide for the following information:

- Defaults to Direct TCP mode
- Stream API support
- Support custom serializer to allow System.Text.JSON usage
- Integrated batch and bulk support

Hosting recommendations

For query-intensive workloads, use Windows 64-bit instead of Linux or Windows 32-bit host processing

We recommend Windows 64-bit host processing for improved performance. The SQL SDK includes a native ServiceInterop.dll to parse and optimize queries locally. ServiceInterop.dll is supported only on the Windows x64 platform. For Linux and other unsupported platforms where ServiceInterop.dll isn't available, an additional network call is made to the gateway to get the optimized query. The following types of applications use 32-bit host processing by default. To change host processing to 64-bit processing, follow these steps, based on the type of your application:

- For executable applications, you can change host processing by setting the platform target to **x64** in the **Project Properties** window, on the **Build** tab.
- For VSTest-based test projects, you can change host processing by selecting **Test** > **Test Settings** > **Default Processor Architecture as X64** on the Visual Studio **Test** menu.
- For locally deployed ASP.NET web applications, you can change host processing by selecting **Use the 64-bit version of IIS Express for web sites and projects** under **Tools** > **Options** > **Projects and Solutions** > **Web Projects**.
- For ASP.NET web applications deployed on Azure, you can change host processing by selecting the **64-bit** platform in **Application settings** in the Azure portal.

> **NOTE**
> By default, new Visual Studio projects are set to **Any CPU**. We recommend that you set your project to **x64** so it doesn't switch to **x86**. A project set to **Any CPU** can easily switch to **x86** if an x86-only dependency is added. ServiceInterop.dll needs to be in the folder that the SDK DLL is being executed from. This should be a concern only if you manually copy DLLs or have custom build/deployment systems.

Turn on server-side garbage collection (GC)

Reducing the frequency of garbage collection can help in some cases. In .NET, set gcServer to `true`.

Scale out your client workload

If you're testing at high throughput levels (more than 50,000 RU/s), the client application could become the bottleneck due to the machine capping out on CPU or network utilization. If you reach this point, you can continue to push the Azure Cosmos DB account further by scaling out your client applications across multiple servers.

> **NOTE**
> High CPU usage can cause increased latency and request timeout exceptions.

Networking

Connection policy: Use direct connection mode

.NET V2 SDK default connection mode is gateway. You configure the connection mode during the construction of the `DocumentClient` instance by using the `ConnectionPolicy` parameter. If you use direct mode, you need to also set the `Protocol` by using the `ConnectionPolicy` parameter. To learn more about different connectivity options, see the connectivity modes article.

```
Uri serviceEndpoint = new Uri("https://contoso.documents.net");
string authKey = "your authKey from the Azure portal";
DocumentClient client = new DocumentClient(serviceEndpoint, authKey,
new ConnectionPolicy
{
   ConnectionMode = ConnectionMode.Direct, // ConnectionMode.Gateway is the default
   ConnectionProtocol = Protocol.Tcp
});
```

Ephemeral port exhaustion

If you see a high connection volume or high port usage on your instances, first verify that your client instances are singletons. In other words, the client instances should be unique for the lifetime of the application.

When running on the TCP protocol, the client optimizes for latency by using the long-lived connections as opposed to the HTTPS protocol, which terminates the connections after 2 minutes of inactivity.

In scenarios where you have sparse access and if you notice a higher connection count when compared to the gateway mode access, you can:

- Configure the ConnectionPolicy.PortReuseMode property to `PrivatePortPool` (effective with framework version>= 4.6.1 and .NET core version >= 2.0): This property allows the SDK to use a small pool of ephemeral ports for different Azure Cosmos DB destination endpoints.
- Configure the ConnectionPolicy.IdleConnectionTimeout property must be greater than or equal to 10 minutes. The recommended values are between 20 minutes and 24 hours.

Call OpenAsync to avoid startup latency on first request

By default, the first request has higher latency because it needs to fetch the address routing table. When you use SDK V2, call `OpenAsync()` once during initialization to avoid this startup latency on the first request. The call looks like: `await client.OpenAsync();`

> **NOTE**
>
> `OpenAsync` will generate requests to obtain the address routing table for all the containers in the account. For accounts that have many containers but whose application accesses a subset of them, `OpenAsync` would generate an unnecessary amount of traffic, which would make the initialization slow. So using `OpenAsync` might not be useful in this scenario because it slows down application startup.

For performance, collocate clients in same Azure region

When possible, place any applications that call Azure Cosmos DB in the same region as the Azure Cosmos DB database. Here's an approximate comparison: calls to Azure Cosmos DB within the same region complete within 1 ms to 2 ms, but the latency between the West and East coast of the US is more than 50 ms. This latency can vary from request to request, depending on the route taken by the request as it passes from the client to the Azure datacenter boundary. You can get the lowest possible latency by ensuring the calling application is located within the same Azure region as the provisioned Azure Cosmos DB endpoint. For a list of available regions, see Azure regions.

Increase the number of threads/tasks

Because calls to Azure Cosmos DB are made over the network, you might need to vary the degree of parallelism of your requests so that the client application spends minimal time waiting between requests. For example, if you're using the .NET Task Parallel Library, create on the order of hundreds of tasks that read from or write to Azure Cosmos DB.

Enable accelerated networking

To reduce latency and CPU jitter, we recommend that you enable accelerated networking on client virtual machines. See Create a Windows virtual machine with accelerated networking or Create a Linux virtual machine with accelerated networking.

SDK usage

Install the most recent SDK

The Azure Cosmos DB SDKs are constantly being improved to provide the best performance. See the Azure Cosmos DB SDK pages to determine the most recent SDK and review improvements.

Use a singleton Azure Cosmos DB client for the lifetime of your application

Each `DocumentClient` instance is thread-safe and performs efficient connection management and address caching when operating in direct mode. To allow efficient connection management and better SDK client performance, we recommend that you use a single instance per `AppDomain` for the lifetime of the application.

Increase System.Net MaxConnections per host when using gateway mode

Azure Cosmos DB requests are made over HTTPS/REST when you use gateway mode. They're subjected to the default connection limit per hostname or IP address. You might need to set `MaxConnections` to a higher value (100 to 1,000) so the client library can use multiple simultaneous connections to Azure Cosmos DB. In .NET SDK 1.8.0 and later, the default value for ServicePointManager.DefaultConnectionLimit is 50. To change the value, you can set Documents.Client.ConnectionPolicy.MaxConnectionLimit to a higher value.

Tune parallel queries for partitioned collections

SQL .NET SDK 1.9.0 and later support parallel queries, which enable you to query a partitioned collection in parallel. For more information, see code samples related to working with the SDKs. Parallel queries are designed to provide better query latency and throughput than their serial counterpart. Parallel queries provide two parameters that you can tune to fit your requirements:

- `MaxDegreeOfParallelism` controls the maximum number of partitions that can be queried in parallel.
- `MaxBufferedItemCount` controls the number of pre-fetched results.

Tuning degree of parallelism

Parallel query works by querying multiple partitions in parallel. But data from an individual partition is fetched serially with respect to the query. Setting `MaxDegreeOfParallelism` in SDK V2 to the number of partitions has the best chance of achieving the most performant query, provided all other system conditions remain the same. If you don't know the number of partitions, you can set the degree of parallelism to a high number. The system will choose the minimum (number of partitions, user provided input) as the degree of parallelism.

Parallel queries produce the most benefit if the data is evenly distributed across all partitions with respect to the query. If the partitioned collection is partitioned so that all or most of the data returned by a query is concentrated in a few partitions (one partition is the worst case), those partitions will bottleneck the performance of the query.

Tuning MaxBufferedItemCount

Parallel query is designed to pre-fetch results while the current batch of results is being processed by the client. This pre-fetching helps improve the overall latency of a query. The `MaxBufferedItemCount` parameter limits the number of pre-fetched results. Set `MaxBufferedItemCount` to the expected number of results returned (or a higher number) to allow the query to receive the maximum benefit from pre-fetching.

Pre-fetching works the same way regardless of the degree of parallelism, and there's a single buffer for the data from all partitions.

Implement backoff at RetryAfter intervals

During performance testing, you should increase load until a small rate of requests are throttled. If requests are throttled, the client application should back off on throttle for the server-specified retry interval. Respecting the backoff ensures you spend a minimal amount of time waiting between retries.

Retry policy support is included in these SDKs:

- Version 1.8.0 and later of the .NET SDK for SQL and the Java SDK for SQL
- Version 1.9.0 and later of the Node.js SDK for SQL and the Python SDK for SQL
- All supported versions of the .NET Core SDKs

For more information, see RetryAfter.

In version 1.19 and later of the .NET SDK, there's a mechanism for logging additional diagnostic information and troubleshooting latency issues, as shown in the following sample. You can log the diagnostic string for requests that have a higher read latency. The captured diagnostic string will help you understand how many times you received 429 errors for a given request.

```
ResourceResponse<Document> readDocument = await
this.readClient.ReadDocumentAsync(oldDocuments[i].SelfLink);
readDocument.RequestDiagnosticsString
```

Cache document URIs for lower read latency

Cache document URIs whenever possible for the best read performance. You need to define logic to cache the resource ID when you create a resource. Lookups based on resource IDs are faster than name-based lookups, so caching these values improves performance.

Tune the page size for queries/read feeds for better performance

When you do a bulk read of documents by using read feed functionality (for example, `ReadDocumentFeedAsync`) or when you issue a SQL query, the results are returned in a segmented fashion if the result set is too large. By default, results are returned in chunks of 100 items or 1 MB, whichever limit is hit first.

To reduce the number of network round trips required to retrieve all applicable results, you can increase the page size by using x-ms-max-item-count to request as many as 1,000 headers. When you need to display only a few results, for example, if your user interface or application API returns only 10 results at a time, you can also decrease the page size to 10 to reduce the throughput consumed for reads and queries.

> **NOTE**
> The `maxItemCount` property shouldn't be used just for pagination. Its main use is to improve the performance of queries by reducing the maximum number of items returned in a single page.

You can also set the page size by using the available Azure Cosmos DB SDKs. The MaxItemCount property in `FeedOptions` allows you to set the maximum number of items to be returned in the enumeration operation. When `maxItemCount` is set to -1, the SDK automatically finds the optimal value, depending on the document size. For example:

```
IQueryable<dynamic> authorResults = client.CreateDocumentQuery(documentCollection.SelfLink, "SELECT
p.Author FROM Pages p WHERE p.Title = 'About Seattle'", new FeedOptions { MaxItemCount = 1000 });
```

When a query is executed, the resulting data is sent within a TCP packet. If you specify too low a value for `maxItemCount`, the number of trips required to send the data within the TCP packet is high, which affects performance. So if you're not sure what value to set for the `maxItemCount` property, it's best to set it to -1 and let the SDK choose the default value.

Increase the number of threads/tasks

See Increase the number of threads/tasks in the networking section of this article.

Indexing policy

Exclude unused paths from indexing for faster writes

The Azure Cosmos DB indexing policy also allows you to specify which document paths to include in or exclude

from indexing by using indexing paths (IndexingPolicy.IncludedPaths and IndexingPolicy.ExcludedPaths). Indexing paths can improve write performance and reduce index storage for scenarios in which the query patterns are known beforehand. This is because indexing costs correlate directly to the number of unique paths indexed. For example, this code shows how to exclude an entire section of the documents (a subtree) from indexing by using the "*" wildcard:

```
var collection = new DocumentCollection { Id = "excludedPathCollection" };
collection.IndexingPolicy.IncludedPaths.Add(new IncludedPath { Path = "/*" });
collection.IndexingPolicy.ExcludedPaths.Add(new ExcludedPath { Path = "/nonIndexedContent/*");
collection = await client.CreateDocumentCollectionAsync(UriFactory.CreateDatabaseUri("db"), collection);
```

For more information, see Azure Cosmos DB indexing policies.

Throughput

Measure and tune for lower Request Units/second usage

Azure Cosmos DB offers a rich set of database operations. These operations include relational and hierarchical queries with UDFs, stored procedures, and triggers, all operating on the documents within a database collection. The cost associated with each of these operations varies depending on the CPU, IO, and memory required to complete the operation. Instead of thinking about and managing hardware resources, you can think of a Request Unit (RU) as a single measure for the resources required to perform various database operations and service an application request.

Throughput is provisioned based on the number of Request Units set for each container. Request Unit consumption is evaluated as a rate per second. Applications that exceed the provisioned Request Unit rate for their container are limited until the rate drops below the provisioned level for the container. If your application requires a higher level of throughput, you can increase your throughput by provisioning additional Request Units.

The complexity of a query affects how many Request Units are consumed for an operation. The number of predicates, the nature of the predicates, the number of UDFs, and the size of the source dataset all influence the cost of query operations.

To measure the overhead of any operation (create, update, or delete), inspect the x-ms-request-charge header (or the equivalent `RequestCharge` property in `ResourceResponse\<T>` or `FeedResponse\<T>` in the .NET SDK) to measure the number of Request Units consumed by the operations:

```
// Measure the performance (Request Units) of writes
ResourceResponse<Document> response = await client.CreateDocumentAsync(collectionSelfLink, myDocument);
Console.WriteLine("Insert of document consumed {0} request units", response.RequestCharge);
// Measure the performance (Request Units) of queries
IDocumentQuery<dynamic> queryable = client.CreateDocumentQuery(collectionSelfLink,
queryString).AsDocumentQuery();
while (queryable.HasMoreResults)
    {
        FeedResponse<dynamic> queryResponse = await queryable.ExecuteNextAsync<dynamic>();
        Console.WriteLine("Query batch consumed {0} request units", queryResponse.RequestCharge);
    }
```

The request charge returned in this header is a fraction of your provisioned throughput (that is, 2,000 RUs / second). For example, if the preceding query returns 1,000 1-KB documents, the cost of the operation is 1,000. So, within one second, the server honors only two such requests before rate limiting later requests. For more information, see Request Units and the Request Unit calculator.

Handle rate limiting/request rate too large

When a client attempts to exceed the reserved throughput for an account, there's no performance degradation at the server and no use of throughput capacity beyond the reserved level. The server will preemptively end the request with RequestRateTooLarge (HTTP status code 429). It will return an x-ms-retry-after-ms header that indicates the amount of time, in milliseconds, that the user must wait before attempting the request again.

```
HTTP Status 429,
Status Line: RequestRateTooLarge
x-ms-retry-after-ms :100
```

The SDKs all implicitly catch this response, respect the server-specified retry-after header, and retry the request. Unless your account is being accessed concurrently by multiple clients, the next retry will succeed.

If you have more than one client cumulatively operating consistently above the request rate, the default retry count currently set to 9 internally by the client might not suffice. In this case, the client throws a DocumentClientException with status code 429 to the application.

You can change the default retry count by setting the `RetryOptions` on the `ConnectionPolicy` instance. By default, the DocumentClientException with status code 429 is returned after a cumulative wait time of 30 seconds if the request continues to operate above the request rate. This error returns even when the current retry count is less than the maximum retry count, whether the current value is the default of 9 or a user-defined value.

The automated retry behavior helps improve resiliency and usability for most applications. But it might not be the best behavior when you're doing performance benchmarks, especially when you're measuring latency. The client-observed latency will spike if the experiment hits the server throttle and causes the client SDK to silently retry. To avoid latency spikes during performance experiments, measure the charge returned by each operation and ensure that requests are operating below the reserved request rate. For more information, see Request Units.

For higher throughput, design for smaller documents

The request charge (that is, the request-processing cost) of a given operation correlates directly to the size of the document. Operations on large documents cost more than operations on small documents.

Next steps

For a sample application that's used to evaluate Azure Cosmos DB for high-performance scenarios on a few client machines, see Performance and scale testing with Azure Cosmos DB.

To learn more about designing your application for scale and high performance, see Partitioning and scaling in Azure Cosmos DB.

Performance tips for Azure Cosmos DB Java SDK v4

12/4/2020 • 21 minutes to read • Edit Online

APPLIES TO: SQL API

> **IMPORTANT**
>
> The performance tips in this article are for Azure Cosmos DB Java SDK v4 only. Please view the Azure Cosmos DB Java SDK v4 Release notes, Maven repository, and Azure Cosmos DB Java SDK v4 troubleshooting guide for more information. If you are currently using an older version than v4, see the Migrate to Azure Cosmos DB Java SDK v4 guide for help upgrading to v4.

Azure Cosmos DB is a fast and flexible distributed database that scales seamlessly with guaranteed latency and throughput. You do not have to make major architecture changes or write complex code to scale your database with Azure Cosmos DB. Scaling up and down is as easy as making a single API call or SDK method call. However, because Azure Cosmos DB is accessed via network calls there are client-side optimizations you can make to achieve peak performance when using Azure Cosmos DB Java SDK v4.

So if you're asking "How can I improve my database performance?" consider the following options:

Networking

- **Connection mode: Use Direct mode**

  Java SDK default connection mode is direct. You can configure the connection mode in the client builder using the *directMode()* or *gatewayMode()* methods, as shown below. To configure either mode with default settings, call either method without arguments. Otherwise, pass a configuration settings class instance as the argument (*DirectConnectionConfig* for *directMode()*, *GatewayConnectionConfig* for *gatewayMode()*.). To learn more about different connectivity options, see the connectivity modes article.

  Java V4 SDK

  - Async
  - Sync

  Java SDK V4 (Maven com.azure::azure-cosmos) Async API

```
/* Direct mode, default settings */
CosmosAsyncClient clientDirectDefault = new CosmosClientBuilder()
        .endpoint(HOSTNAME)
        .key(MASTERKEY)
        .consistencyLevel(CONSISTENCY)
        .directMode()
        .buildAsyncClient();

/* Direct mode, custom settings */
DirectConnectionConfig directConnectionConfig = DirectConnectionConfig.getDefaultConfig();

// Example config, do not use these settings as defaults
directConnectionConfig.setMaxConnectionsPerEndpoint(120);
directConnectionConfig.setIdleConnectionTimeout(Duration.ofMillis(100));

CosmosAsyncClient clientDirectCustom = new CosmosClientBuilder()
        .endpoint(HOSTNAME)
        .key(MASTERKEY)
        .consistencyLevel(CONSISTENCY)
        .directMode(directConnectionConfig)
        .buildAsyncClient();

/* Gateway mode, default settings */
CosmosAsyncClient clientGatewayDefault = new CosmosClientBuilder()
        .endpoint(HOSTNAME)
        .key(MASTERKEY)
        .consistencyLevel(CONSISTENCY)
        .gatewayMode()
        .buildAsyncClient();

/* Gateway mode, custom settings */
GatewayConnectionConfig gatewayConnectionConfig = GatewayConnectionConfig.getDefaultConfig();

// Example config, do not use these settings as defaults
gatewayConnectionConfig.setProxy(new ProxyOptions(ProxyOptions.Type.HTTP,
InetSocketAddress.createUnresolved("your.proxy.addr",80)));
gatewayConnectionConfig.setMaxConnectionPoolSize(150);

CosmosAsyncClient clientGatewayCustom = new CosmosClientBuilder()
        .endpoint(HOSTNAME)
        .key(MASTERKEY)
        .consistencyLevel(CONSISTENCY)
        .gatewayMode(gatewayConnectionConfig)
        .buildAsyncClient();

/* No connection mode, defaults to Direct mode with default settings */
CosmosAsyncClient clientDefault = new CosmosClientBuilder()
        .endpoint(HOSTNAME)
        .key(MASTERKEY)
        .consistencyLevel(CONSISTENCY)
        .buildAsyncClient();
```

The *directMode()* method has an additional override, for the following reason. Control plane operations such as database and container CRUD *always* utilize Gateway mode; when the user has configured Direct mode for data plane operations, control plane operations use default Gateway mode settings. This suits most users. However, users who want Direct mode for data plane operations as well as tunability of control plane Gateway mode parameters can use the following *directMode()* override:

Java V4 SDK

- Async
- Sync

Java SDK V4 (Maven com.azure::azure-cosmos) Async API

```
/* Independent customization of Direct mode data plane and Gateway mode control plane */
DirectConnectionConfig directConnectionConfig = DirectConnectionConfig.getDefaultConfig();

// Example config, do not use these settings as defaults
directConnectionConfig.setMaxConnectionsPerEndpoint(120);
directConnectionConfig.setIdleConnectionTimeout(Duration.ofMillis(100));

GatewayConnectionConfig gatewayConnectionConfig = GatewayConnectionConfig.getDefaultConfig();

// Example config, do not use these settings as defaults
gatewayConnectionConfig.setProxy(new ProxyOptions(ProxyOptions.Type.HTTP,
InetSocketAddress.createUnresolved("your.proxy.addr",80)));
gatewayConnectionConfig.setMaxConnectionPoolSize(150);

CosmosAsyncClient clientDirectCustom = new CosmosClientBuilder()
        .endpoint(HOSTNAME)
        .key(MASTERKEY)
        .consistencyLevel(CONSISTENCY)
        .directMode(directConnectionConfig,gatewayConnectionConfig)
        .buildAsyncClient();
```

- **Collocate clients in same Azure region for performance**

  When possible, place any applications calling Azure Cosmos DB in the same region as the Azure Cosmos database. For an approximate comparison, calls to Azure Cosmos DB within the same region complete within 1-2 ms, but the latency between the West and East coast of the US is >50 ms. This latency can likely vary from request to request depending on the route taken by the request as it passes from the client to the Azure datacenter boundary. The lowest possible latency is achieved by ensuring the calling application is located within the same Azure region as the provisioned Azure Cosmos DB endpoint. For a list of available regions, see Azure Regions.

  An app that interacts with a multi-region Azure Cosmos DB account needs to configure preferred locations to ensure that requests are going to a collocated region.

- **Enable Accelerated Networking on your Azure VM for lower latency.**

It is recommended that you follow the instructions to enable Accelerated Networking in your Windows (click for instructions) or Linux (click for instructions) Azure VM, in order to maximize performance.

Without accelerated networking, IO that transits between your Azure VM and other Azure resources may be unnecessarily routed through a host and virtual switch situated between the VM and its network card. Having the host and virtual switch inline in the datapath not only increases latency and jitter in the communication channel, it also steals CPU cycles from the VM. With accelerated networking, the VM interfaces directly with the

NIC without intermediaries; any network policy details which were being handled by the host and virtual switch are now handled in hardware at the NIC; the host and virtual switch are bypassed. Generally you can expect lower latency and higher throughput, as well as more *consistent* latency and decreased CPU utilization when you enable accelerated networking.

Limitations: accelerated networking must be supported on the VM OS, and can only be enabled when the VM is stopped and deallocated. The VM cannot be deployed with Azure Resource Manager.

Please see the Windows and Linux instructions for more details.

SDK usage

- **Install the most recent SDK**

  The Azure Cosmos DB SDKs are constantly being improved to provide the best performance. See the Azure Cosmos DB SDK pages to determine the most recent SDK and review improvements.

- **Use a singleton Azure Cosmos DB client for the lifetime of your application**

  Each Azure Cosmos DB client instance is thread-safe and performs efficient connection management and address caching. To allow efficient connection management and better performance by the Azure Cosmos DB client, it is recommended to use a single instance of the Azure Cosmos DB client per AppDomain for the lifetime of the application.

- **Use the lowest consistency level required for your application**

  When you create a *CosmosClient*, the default consistency used if not explicitly set is *Session*. If *Session* consistency is not required by your application logic set the *Consistency* to *Eventual*. Note: it is recommended to use at least *Session* consistency in applications employing the Azure Cosmos DB Change Feed processor.

- **Use Async API to max out provisioned throughput**

  Azure Cosmos DB Java SDK v4 bundles two APIs, Sync and Async. Roughly speaking, the Async API implements SDK functionality, whereas the Sync API is a thin wrapper that makes blocking calls to the Async API. This stands in contrast to the older Azure Cosmos DB Async Java SDK v2, which was Async-only, and to the older Azure Cosmos DB Sync Java SDK v2, which was Sync-only and had a completely separate implementation.

  The choice of API is determined during client initialization; a *CosmosAsyncClient* supports Async API while a *CosmosClient* supports Sync API.

  The Async API implements non-blocking IO and is the optimal choice if your goal is to max out throughput when issuing requests to Azure Cosmos DB.

  Using Sync API can be the right choice if you want or need an API which blocks on the response to each request, or if synchronous operation is the dominant paradigm in your application. For example, you might want the Sync API when you are persisting data to Azure Cosmos DB in a microservices application, provided throughput is not critical.

  Just be aware that Sync API throughput degrades with increasing request response-time, whereas the Async API can saturate the full bandwidth capabilities of your hardware.

  Geographic collocation can give you higher and more consistent throughput when using Sync API (see Collocate clients in same Azure region for performance) but still is not expected to exceed Async API attainable throughput.

  Some users may also be unfamiliar with Project Reactor, the Reactive Streams framework used to implement Azure Cosmos DB Java SDK v4 Async API. If this is a concern, we recommend you read our

introductory Reactor Pattern Guide and then take a look at this Introduction to Reactive Programming in order to familiarize yourself. If you have already used Azure Cosmos DB with an Async interface, and the SDK you used was Azure Cosmos DB Async Java SDK v2, then you may be familiar with ReactiveX/RxJava but be unsure what has changed in Project Reactor. In that case, please take a look at our Reactor vs. RxJava Guide to become familiarized.

The following code snippets show how to initialize your Azure Cosmos DB client for Async API or Sync API operation, respectively:

Java V4 SDK

- Async
- Sync

Java SDK V4 (Maven com.azure::azure-cosmos) Async API

```
CosmosAsyncClient client = new CosmosClientBuilder()
        .endpoint(HOSTNAME)
        .key(MASTERKEY)
        .consistencyLevel(CONSISTENCY)
        .buildAsyncClient();
```

- Tuning ConnectionPolicy

  By default, Direct mode Cosmos DB requests are made over TCP when using Azure Cosmos DB Java SDK v4. Internally Direct mode uses a special architecture to dynamically manage network resources and get the best performance.

  In Azure Cosmos DB Java SDK v4, Direct mode is the best choice to improve database performance with most workloads.

  - *Overview of Direct mode*

The client-side architecture employed in Direct mode enables predictable network utilization and multiplexed access to Azure Cosmos DB replicas. The diagram above shows how Direct mode routes client requests to replicas in the Cosmos DB backend. The Direct mode architecture allocates up to 10 **Channels** on the client side per DB replica. A Channel is a TCP connection preceded by a request buffer, which is 30 requests deep. The Channels belonging to a replica are dynamically allocated as needed by the replica's **Service Endpoint**. When the user issues a request in Direct mode, the **TransportClient** routes the request to the proper service endpoint based on the partition key. The **Request Queue** buffers requests before the Service Endpoint.

- ***Configuration options for Direct mode***

  If non-default Direct mode behavior is desired, create a *DirectConnectionConfig* instance and customize its properties, then pass the customized property instance to the *directMode()* method in the Azure Cosmos DB client builder.

  These configuration settings control the behavior of the underlying Direct mode architecture discussed above.

  As a first step, use the following recommended configuration settings below. These *DirectConnectionConfig* options are advanced configuration settings which can affect SDK performance in unexpected ways; we recommend users avoid modifying them unless they feel very comfortable in understanding the tradeoffs and it is absolutely necessary. Please contact the Azure Cosmos DB team if you run into issues on this particular topic.

CONFIGURATION OPTION	DEFAULT
idleConnectionTimeout	"PT0"
maxConnectionsPerEndpoint	"130"

CONFIGURATION OPTION	DEFAULT
connectTimeout	"PT5S"
idleEndpointTimeout	"PT1H"
maxRequestsPerConnection	"30"

- **Tuning parallel queries for partitioned collections**

  Azure Cosmos DB Java SDK v4 supports parallel queries, which enable you to query a partitioned collection in parallel. For more information, see code samples related to working with Azure Cosmos DB Java SDK v4. Parallel queries are designed to improve query latency and throughput over their serial counterpart.

  - ***Tuning setMaxDegreeOfParallelism:***

    Parallel queries work by querying multiple partitions in parallel. However, data from an individual partitioned collection is fetched serially with respect to the query. So, use setMaxDegreeOfParallelism to set the number of partitions that has the maximum chance of achieving the most performant query, provided all other system conditions remain the same. If you don't know the number of partitions, you can use setMaxDegreeOfParallelism to set a high number, and the system chooses the minimum (number of partitions, user provided input) as the maximum degree of parallelism.

    It is important to note that parallel queries produce the best benefits if the data is evenly distributed across all partitions with respect to the query. If the partitioned collection is partitioned such a way that all or a majority of the data returned by a query is concentrated in a few partitions (one partition in worst case), then the performance of the query would be bottlenecked by those partitions.

  - ***Tuning setMaxBufferedItemCount:***

    Parallel query is designed to pre-fetch results while the current batch of results is being processed by the client. The pre-fetching helps in overall latency improvement of a query. setMaxBufferedItemCount limits the number of pre-fetched results. Setting setMaxBufferedItemCount to the expected number of results returned (or a higher number) enables the query to receive maximum benefit from pre-fetching.

    Pre-fetching works the same way irrespective of the MaxDegreeOfParallelism, and there is a single buffer for the data from all partitions.

- **Scale out your client-workload**

  If you are testing at high throughput levels, the client application may become the bottleneck due to the machine capping out on CPU or network utilization. If you reach this point, you can continue to push the Azure Cosmos DB account further by scaling out your client applications across multiple servers.

  A good rule of thumb is not to exceed >50% CPU utilization on any given server, to keep latency low.

- **Tune the page size for queries/read feeds for better performance**

  When performing a bulk read of documents by using read feed functionality (for example, *readItems*) or when issuing a SQL query (*queryItems*), the results are returned in a segmented fashion if the result set is too large. By default, results are returned in chunks of 100 items or 1 MB, whichever limit is hit first.

  Suppose that your application issues a query to Azure Cosmos DB, and suppose that your application requires the full set of query results in order to complete its task. To reduce the number of network

round trips required to retrieve all applicable results, you can increase the page size by adjusting the x-ms-max-item-count request header field.

- For single-partition queries, adjusting the x-ms-max-item-count field value to -1 (no limit on page size) maximizes latency by minimizing the number of query response pages: either the full result set will return in a single page, or if the query takes longer than the timeout interval, then the full result set will be returned in the minimum number of pages possible. This saves on multiples of the request round-trip time.
- For cross-partition queries, setting the x-ms-max-item-count field to -1 and removing the page size limit risks overwhelming the SDK with unmanageable page sizes. In the cross-partition case we recommend raising the page size limit up to some large but finite value, perhaps 1000, to reduce latency.

In some applications, you may not require the full set of query results. In cases where you need to display only a few results, for example, if your user interface or application API returns only 10 results at a time, you can also decrease the page size to 10 to reduce the throughput consumed for reads and queries.

You may also set the preferred page size argument of the *byPage* method, rather than modifying the REST header field directly. Keep in mind that x-ms-max-item-count or the preferred page size argument of *byPage* are only setting an upper limit on page size, not an absolute requirement; so for a variety of reason you may see Azure Cosmos DB return pages which are smaller than your preferred page size.

- **Use Appropriate Scheduler (Avoid stealing Event loop IO Netty threads)**

  The asynchronous functionality of Azure Cosmos DB Java SDK is based on netty non-blocking IO. The SDK uses a fixed number of IO netty event loop threads (as many CPU cores your machine has) for executing IO operations. The Flux returned by API emits the result on one of the shared IO event loop netty threads. So it is important to not block the shared IO event loop netty threads. Doing CPU intensive work or blocking operation on the IO event loop netty thread may cause deadlock or significantly reduce SDK throughput.

  For example the following code executes a cpu intensive work on the event loop IO netty thread:

  Java SDK V4 (Maven com.azure::azure-cosmos) Async API

```
Mono<CosmosItemResponse<CustomPOJO>> createItemPub = asyncContainer.createItem(item);
createItemPub.subscribe(
        itemResponse -> {
            //this is executed on eventloop IO netty thread.
            //the eventloop thread is shared and is meant to return back quickly.
            //
            // DON'T do this on eventloop IO netty thread.
            veryCpuIntensiveWork();
        });
```

  After result is received if you want to do CPU intensive work on the result you should avoid doing so on event loop IO netty thread. You can instead provide your own Scheduler to provide your own thread for running your work, as shown below (requires `import reactor.core.scheduler.Schedulers`).

  Java SDK V4 (Maven com.azure::azure-cosmos) Async API

```
Mono<CosmosItemResponse<CustomPOJO>> createItemPub = asyncContainer.createItem(item);
createItemPub
        .subscribeOn(Schedulers.elastic())
        .subscribe(
                itemResponse -> {
                    //this is executed on eventloop IO netty thread.
                    //the eventloop thread is shared and is meant to return back quickly.
                    //
                    // DON'T do this on eventloop IO netty thread.
                    veryCpuIntensiveWork();
                });
```

Based on the type of your work you should use the appropriate existing Reactor Scheduler for your work. Read here `Schedulers`.

For more information on Azure Cosmos DB Java SDK v4, please look at the Cosmos DB directory of the Azure SDK for Java monorepo on GitHub.

- **Optimize logging settings in your application**

  For a variety of reasons, you may want or need to add logging in a thread which is generating high request throughput. If your goal is to fully saturate a container's provisioned throughput with requests generated by this thread, logging optimizations can greatly improve performance.

  - ***Configure an async logger***

    The latency of a synchronous logger necessarily factors into the overall latency calculation of your request-generating thread. An async logger such as log4j2 is recommended to decouple logging overhead from your high-performance application threads.

  - ***Disable netty's logging***

    Netty library logging is chatty and needs to be turned off (suppressing sign in the configuration may not be enough) to avoid additional CPU costs. If you are not in debugging mode, disable netty's logging altogether. So if you are using log4j to remove the additional CPU costs incurred by `org.apache.log4j.Category.callAppenders()` from netty add the following line to your codebase:

    ```
    org.apache.log4j.Logger.getLogger("io.netty").setLevel(org.apache.log4j.Level.OFF);
    ```

- **OS Open files Resource Limit**

  Some Linux systems (like Red Hat) have an upper limit on the number of open files and so the total number of connections. Run the following to view the current limits:

  ```
  ulimit -a
  ```

  The number of open files (nofile) needs to be large enough to have enough room for your configured connection pool size and other open files by the OS. It can be modified to allow for a larger connection pool size.

  Open the limits.conf file:

  ```
  vim /etc/security/limits.conf
  ```

Add/modify the following lines:

```
* - nofile 100000
```

- **Specify partition key in point writes**

  To improve the performance of point writes, specify item partition key in the point write API call, as shown below:

  - Async
  - Sync

  Java SDK V4 (Maven com.azure::azure-cosmos) Async API

  ```
  asyncContainer.createItem(item,new PartitionKey(pk),new CosmosItemRequestOptions()).block();
  ```

  rather than providing only the item instance, as shown below:

  - Async
  - Sync

  Java SDK V4 (Maven com.azure::azure-cosmos) Async API

  ```
  asyncContainer.createItem(item).block();
  ```

  The latter is supported but will add latency to your application; the SDK must parse the item and extract the partition key.

Indexing policy

- **Exclude unused paths from indexing for faster writes**

  Azure Cosmos DB's indexing policy allows you to specify which document paths to include or exclude from indexing by leveraging Indexing Paths (setIncludedPaths and setExcludedPaths). The use of indexing paths can offer improved write performance and lower index storage for scenarios in which the query patterns are known beforehand, as indexing costs are directly correlated to the number of unique paths indexed. For example, the following code shows how to include and exclude entire sections of the documents (also known as a subtree) from indexing using the "*" wildcard.

  Java SDK V4 (Maven com.azure::azure-cosmos)

```
CosmosContainerProperties containerProperties = new CosmosContainerProperties(containerName,
"/lastName");

// Custom indexing policy
IndexingPolicy indexingPolicy = new IndexingPolicy();
indexingPolicy.setIndexingMode(IndexingMode.CONSISTENT);

// Included paths
List<IncludedPath> includedPaths = new ArrayList<>();
includedPaths.add(new IncludedPath("/*"));
indexingPolicy.setIncludedPaths(includedPaths);

// Excluded paths
List<ExcludedPath> excludedPaths = new ArrayList<>();
excludedPaths.add(new ExcludedPath("/name/*"));
indexingPolicy.setExcludedPaths(excludedPaths);

containerProperties.setIndexingPolicy(indexingPolicy);

ThroughputProperties throughputProperties = ThroughputProperties.createManualThroughput(400);

database.createContainerIfNotExists(containerProperties, throughputProperties);
CosmosAsyncContainer containerIfNotExists = database.getContainer(containerName);
```

For more information, see Azure Cosmos DB indexing policies.

Throughput

- **Measure and tune for lower request units/second usage**

  Azure Cosmos DB offers a rich set of database operations including relational and hierarchical queries with UDFs, stored procedures, and triggers – all operating on the documents within a database collection. The cost associated with each of these operations varies based on the CPU, IO, and memory required to complete the operation. Instead of thinking about and managing hardware resources, you can think of a request unit (RU) as a single measure for the resources required to perform various database operations and service an application request.

  Throughput is provisioned based on the number of request units set for each container. Request unit consumption is evaluated as a rate per second. Applications that exceed the provisioned request unit rate for their container are limited until the rate drops below the provisioned level for the container. If your application requires a higher level of throughput, you can increase your throughput by provisioning additional request units.

  The complexity of a query impacts how many request units are consumed for an operation. The number of predicates, nature of the predicates, number of UDFs, and the size of the source data set all influence the cost of query operations.

  To measure the overhead of any operation (create, update, or delete), inspect the x-ms-request-charge header to measure the number of request units consumed by these operations. You can also look at the equivalent RequestCharge property in ResourceResponse<T> or FeedResponse<T>.

  - Async
  - Sync

  Java SDK V4 (Maven com.azure::azure-cosmos) Async API

```
CosmosItemResponse<CustomPOJO> response = asyncContainer.createItem(item).block();

response.getRequestCharge();
```

The request charge returned in this header is a fraction of your provisioned throughput. For example, if you have 2000 RU/s provisioned, and if the preceding query returns 1000 1KB-documents, the cost of the operation is 1000. As such, within one second, the server honors only two such requests before rate limiting subsequent requests. For more information, see Request units and the request unit calculator.

- **Handle rate limiting/request rate too large**

  When a client attempts to exceed the reserved throughput for an account, there is no performance degradation at the server and no use of throughput capacity beyond the reserved level. The server will preemptively end the request with RequestRateTooLarge (HTTP status code 429) and return the x-ms-retry-after-ms header indicating the amount of time, in milliseconds, that the user must wait before reattempting the request.

  ```
  HTTP Status 429,
  Status Line: RequestRateTooLarge
  x-ms-retry-after-ms :100
  ```

  The SDKs all implicitly catch this response, respect the server-specified retry-after header, and retry the request. Unless your account is being accessed concurrently by multiple clients, the next retry will succeed.

  If you have more than one client cumulatively operating consistently above the request rate, the default retry count currently set to 9 internally by the client may not suffice; in this case, the client throws a *CosmosClientException* with status code 429 to the application. The default retry count can be changed by using setRetryOptions on the ConnectionPolicy instance. By default, the *CosmosClientException* with status code 429 is returned after a cumulative wait time of 30 seconds if the request continues to operate above the request rate. This occurs even when the current retry count is less than the max retry count, be it the default of 9 or a user-defined value.

  While the automated retry behavior helps to improve resiliency and usability for the most applications, it might come at odds when doing performance benchmarks, especially when measuring latency. The client-observed latency will spike if the experiment hits the server throttle and causes the client SDK to silently retry. To avoid latency spikes during performance experiments, measure the charge returned by each operation and ensure that requests are operating below the reserved request rate. For more information, see Request units.

- **Design for smaller documents for higher throughput**

  The request charge (the request processing cost) of a given operation is directly correlated to the size of the document. Operations on large documents cost more than operations for small documents. Ideally, architect your application and workflows to have your item size be ~1KB, or similar order or magnitude. For latency-sensitive applications large items should be avoided - multi-MB documents will slow down your application.

Next steps

To learn more about designing your application for scale and high performance, see Partitioning and scaling in Azure Cosmos DB.

Performance tips for Azure Cosmos DB Async Java SDK v2

12/4/2020 • 15 minutes to read • Edit Online

APPLIES TO: SQL API

> **IMPORTANT**
>
> This is *not* the latest Java SDK for Azure Cosmos DB! You should upgrade your project to Azure Cosmos DB Java SDK v4 and then read the Azure Cosmos DB Java SDK v4 performance tips guide. Follow the instructions in the Migrate to Azure Cosmos DB Java SDK v4 guide and Reactor vs RxJava guide to upgrade.
>
> The performance tips in this article are for Azure Cosmos DB Async Java SDK v2 only. See the Azure Cosmos DB Async Java SDK v2 Release notes, Maven repository, and Azure Cosmos DB Async Java SDK v2 troubleshooting guide for more information.

Azure Cosmos DB is a fast and flexible distributed database that scales seamlessly with guaranteed latency and throughput. You do not have to make major architecture changes or write complex code to scale your database with Azure Cosmos DB. Scaling up and down is as easy as making a single API call or SDK method call. However, because Azure Cosmos DB is accessed via network calls there are client-side optimizations you can make to achieve peak performance when using the Azure Cosmos DB Async Java SDK v2.

So if you're asking "How can I improve my database performance?" consider the following options:

Networking

- **Connection mode: Use Direct mode**

  How a client connects to Azure Cosmos DB has important implications on performance, especially in terms of client-side latency. The *ConnectionMode* is a key configuration setting available for configuring the client *ConnectionPolicy*. For Azure Cosmos DB Async Java SDK v2, the two available ConnectionModes are:

  - Gateway (default)
  - Direct

  Gateway mode is supported on all SDK platforms and it is the configured option by default. If your applications run within a corporate network with strict firewall restrictions, Gateway mode is the best choice since it uses the standard HTTPS port and a single endpoint. The performance tradeoff, however, is that Gateway mode involves an additional network hop every time data is read or written to Azure Cosmos DB. Because of this, Direct mode offers better performance due to fewer network hops.

  The *ConnectionMode* is configured during the construction of the *DocumentClient* instance with the *ConnectionPolicy* parameter.

Async Java SDK V2 (Maven com.microsoft.azure::azure-cosmosdb)

```
    public ConnectionPolicy getConnectionPolicy() {
        ConnectionPolicy policy = new ConnectionPolicy();
        policy.setConnectionMode(ConnectionMode.Direct);
        policy.setMaxPoolSize(1000);
        return policy;
    }

    ConnectionPolicy connectionPolicy = new ConnectionPolicy();
    DocumentClient client = new DocumentClient(HOST, MASTER_KEY, connectionPolicy, null);
```

- **Collocate clients in same Azure region for performance**

  When possible, place any applications calling Azure Cosmos DB in the same region as the Azure Cosmos database. For an approximate comparison, calls to Azure Cosmos DB within the same region complete within 1-2 ms, but the latency between the West and East coast of the US is >50 ms. This latency can likely vary from request to request depending on the route taken by the request as it passes from the client to the Azure datacenter boundary. The lowest possible latency is achieved by ensuring the calling application is located within the same Azure region as the provisioned Azure Cosmos DB endpoint. For a list of available regions, see Azure Regions.

SDK Usage

- **Install the most recent SDK**

  The Azure Cosmos DB SDKs are constantly being improved to provide the best performance. See the Azure Cosmos DB Async Java SDK v2 Release Notes pages to determine the most recent SDK and review improvements.

- **Use a singleton Azure Cosmos DB client for the lifetime of your application**

  Each AsyncDocumentClient instance is thread-safe and performs efficient connection management and address caching. To allow efficient connection management and better performance by AsyncDocumentClient, it is recommended to use a single instance of AsyncDocumentClient per AppDomain for the lifetime of the application.

- **Tuning ConnectionPolicy**

  By default, Direct mode Cosmos DB requests are made over TCP when using the Azure Cosmos DB Async Java SDK v2. Internally the SDK uses a special Direct mode architecture to dynamically manage network resources and get the best performance.

  In the Azure Cosmos DB Async Java SDK v2, Direct mode is the best choice to improve database performance with most workloads.

  - *Overview of Direct mode*

The client-side architecture employed in Direct mode enables predictable network utilization and multiplexed access to Azure Cosmos DB replicas. The diagram above shows how Direct mode routes client requests to replicas in the Cosmos DB backend. The Direct mode architecture allocates up to 10 **Channels** on the client side per DB replica. A Channel is a TCP connection preceded by a request buffer, which is 30 requests deep. The Channels belonging to a replica are dynamically allocated as needed by the replica's **Service Endpoint**. When the user issues a request in Direct mode, the **TransportClient** routes the request to the proper service endpoint based on the partition key. The **Request Queue** buffers requests before the Service Endpoint.

- *ConnectionPolicy Configuration options for Direct mode*

  As a first step, use the following recommended configuration settings below. Please contact the Azure Cosmos DB team if you run into issues on this particular topic.

  If you are using Azure Cosmos DB as a reference database (that is, the database is used for many point read operations and few write operations), it may be acceptable to set *idleEndpointTimeout* to 0 (that is, no timeout).

CONFIGURATION OPTION	DEFAULT
bufferPageSize	8192
connectionTimeout	"PT1M"
idleChannelTimeout	"PT0S"
idleEndpointTimeout	"PT1M10S"
maxBufferCapacity	8388608

CONFIGURATION OPTION	DEFAULT
maxChannelsPerEndpoint	10
maxRequestsPerChannel	30
receiveHangDetectionTime	"PT1M5S"
requestExpiryInterval	"PT5S"
requestTimeout	"PT1M"
requestTimerResolution	"PT0.5S"
sendHangDetectionTime	"PT10S"
shutdownTimeout	"PT15S"

- ***Programming tips for Direct mode***

  Review the Azure Cosmos DB Async Java SDK v2 Troubleshooting article as a baseline for resolving any SDK issues.

  Some important programming tips when using Direct mode:

  - **Use multithreading in your application for efficient TCP data transfer** - After making a request, your application should subscribe to receive data on another thread. Not doing so forces unintended "half-duplex" operation and the subsequent requests are blocked waiting for the previous request's reply.
  - **Carry out compute-intensive workloads on a dedicated thread** - For similar reasons to the previous tip, operations such as complex data processing are best placed in a separate thread. A request that pulls in data from another data store (for example if the thread utilizes Azure Cosmos DB and Spark data stores simultaneously) may experience increased latency and it is recommended to spawn an additional thread that awaits a response from the other data store.
    - The underlying network IO in the Azure Cosmos DB Async Java SDK v2 is managed by Netty, see these tips for avoiding coding patterns that block Netty IO threads.
  - **Data modeling** - The Azure Cosmos DB SLA assumes document size to be less than 1KB. Optimizing your data model and programming to favor smaller document size will generally lead to decreased latency. If you are going to need storage and retrieval of docs larger than 1KB, the recommended approach is for documents to link to data in Azure Blob Storage.

- **Tuning parallel queries for partitioned collections**

  Azure Cosmos DB Async Java SDK v2 supports parallel queries, which enable you to query a partitioned collection in parallel. For more information, see code samples related to working with the SDKs. Parallel queries are designed to improve query latency and throughput over their serial counterpart.

  - ***Tuning setMaxDegreeOfParallelism:***

    Parallel queries work by querying multiple partitions in parallel. However, data from an individual partitioned collection is fetched serially with respect to the query. So, use setMaxDegreeOfParallelism to set the number of partitions that has the maximum chance of achieving the most performant query, provided all other system conditions remain the same. If you don't know the number of partitions, you can use setMaxDegreeOfParallelism to set a high number,

and the system chooses the minimum (number of partitions, user provided input) as the maximum degree of parallelism.

It is important to note that parallel queries produce the best benefits if the data is evenly distributed across all partitions with respect to the query. If the partitioned collection is partitioned such a way that all or a majority of the data returned by a query is concentrated in a few partitions (one partition in worst case), then the performance of the query would be bottlenecked by those partitions.

- *Tuning setMaxBufferedItemCount:*

  Parallel query is designed to pre-fetch results while the current batch of results is being processed by the client. The pre-fetching helps in overall latency improvement of a query. setMaxBufferedItemCount limits the number of pre-fetched results. Setting setMaxBufferedItemCount to the expected number of results returned (or a higher number) enables the query to receive maximum benefit from pre-fetching.

  Pre-fetching works the same way irrespective of the MaxDegreeOfParallelism, and there is a single buffer for the data from all partitions.

- **Implement backoff at getRetryAfterInMilliseconds intervals**

  During performance testing, you should increase load until a small rate of requests get throttled. If throttled, the client application should backoff for the server-specified retry interval. Respecting the backoff ensures that you spend minimal amount of time waiting between retries.

- **Scale out your client-workload**

  If you are testing at high throughput levels (>50,000 RU/s), the client application may become the bottleneck due to the machine capping out on CPU or network utilization. If you reach this point, you can continue to push the Azure Cosmos DB account further by scaling out your client applications across multiple servers.

- **Use name based addressing**

  Use name-based addressing, where links have the format `dbs/MyDatabaseId/colls/MyCollectionId/docs/MyDocumentId`, instead of SelfLinks (_self), which have the format `dbs/<database_rid>/colls/<collection_rid>/docs/<document_rid>` to avoid retrieving ResourceIds of all the resources used to construct the link. Also, as these resources get recreated (possibly with same name), caching them may not help.

- **Tune the page size for queries/read feeds for better performance**

  When performing a bulk read of documents by using read feed functionality (for example, readDocuments) or when issuing a SQL query, the results are returned in a segmented fashion if the result set is too large. By default, results are returned in chunks of 100 items or 1 MB, whichever limit is hit first.

  To reduce the number of network round trips required to retrieve all applicable results, you can increase the page size using the x-ms-max-item-count request header to up to 1000. In cases where you need to display only a few results, for example, if your user interface or application API returns only 10 results a time, you can also decrease the page size to 10 to reduce the throughput consumed for reads and queries.

  You may also set the page size using the setMaxItemCount method.

- **Use Appropriate Scheduler (Avoid stealing Event loop IO Netty threads)**

  The Azure Cosmos DB Async Java SDK v2 uses netty for non-blocking IO. The SDK uses a fixed number of IO netty event loop threads (as many CPU cores your machine has) for executing IO operations. The Observable returned by API emits the result on one of the shared IO event loop netty threads. So it is

important to not block the shared IO event loop netty threads. Doing CPU intensive work or blocking operation on the IO event loop netty thread may cause deadlock or significantly reduce SDK throughput.

For example the following code executes a cpu intensive work on the event loop IO netty thread:

Async Java SDK V2 (Maven com.microsoft.azure::azure-cosmosdb)

```
Observable<ResourceResponse<Document>> createDocObs = asyncDocumentClient.createDocument(
  collectionLink, document, null, true);

createDocObs.subscribe(
  resourceResponse -> {
    //this is executed on eventloop IO netty thread.
    //the eventloop thread is shared and is meant to return back quickly.
    //
    // DON'T do this on eventloop IO netty thread.
    veryCpuIntensiveWork();
  });
```

After result is received if you want to do CPU intensive work on the result you should avoid doing so on event loop IO netty thread. You can instead provide your own Scheduler to provide your own thread for running your work.

Async Java SDK V2 (Maven com.microsoft.azure::azure-cosmosdb)

```
import rx.schedulers;

Observable<ResourceResponse<Document>> createDocObs = asyncDocumentClient.createDocument(
  collectionLink, document, null, true);

createDocObs.subscribeOn(Schedulers.computation())
subscribe(
  resourceResponse -> {
    // this is executed on threads provided by Scheduler.computation()
    // Schedulers.computation() should be used only when:
    //   1. The work is cpu intensive
    //   2. You are not doing blocking IO, thread sleep, etc. in this thread against other
resources.
    veryCpuIntensiveWork();
  });
```

Based on the type of your work you should use the appropriate existing RxJava Scheduler for your work. Read here `Schedulers`.

For More Information, Please look at the GitHub page for Azure Cosmos DB Async Java SDK v2.

- **Disable netty's logging**

  Netty library logging is chatty and needs to be turned off (suppressing sign in the configuration may not be enough) to avoid additional CPU costs. If you are not in debugging mode, disable netty's logging altogether. So if you are using log4j to remove the additional CPU costs incurred by `org.apache.log4j.Category.callAppenders()` from netty add the following line to your codebase:

```
org.apache.log4j.Logger.getLogger("io.netty").setLevel(org.apache.log4j.Level.OFF);
```

- **OS Open files Resource Limit**

  Some Linux systems (like Red Hat) have an upper limit on the number of open files and so the total number of connections. Run the following to view the current limits:

```
ulimit -a
```

The number of open files (nofile) needs to be large enough to have enough room for your configured connection pool size and other open files by the OS. It can be modified to allow for a larger connection pool size.

Open the limits.conf file:

```
vim /etc/security/limits.conf
```

Add/modify the following lines:

```
* - nofile 100000
```

Indexing Policy

- **Exclude unused paths from indexing for faster writes**

  Azure Cosmos DB's indexing policy allows you to specify which document paths to include or exclude from indexing by leveraging Indexing Paths (setIncludedPaths and setExcludedPaths). The use of indexing paths can offer improved write performance and lower index storage for scenarios in which the query patterns are known beforehand, as indexing costs are directly correlated to the number of unique paths indexed. For example, the following code shows how to exclude an entire section of the documents (also known as a subtree) from indexing using the "*" wildcard.

  Async Java SDK V2 (Maven com.microsoft.azure::azure-cosmosdb)

```
Index numberIndex = Index.Range(DataType.Number);
numberIndex.set("precision", -1);
indexes.add(numberIndex);
includedPath.setIndexes(indexes);
includedPaths.add(includedPath);
indexingPolicy.setIncludedPaths(includedPaths);
collectionDefinition.setIndexingPolicy(indexingPolicy);
```

  For more information, see Azure Cosmos DB indexing policies.

Throughput

- **Measure and tune for lower request units/second usage**

  Azure Cosmos DB offers a rich set of database operations including relational and hierarchical queries with UDFs, stored procedures, and triggers – all operating on the documents within a database collection. The cost associated with each of these operations varies based on the CPU, IO, and memory required to complete the operation. Instead of thinking about and managing hardware resources, you can think of a request unit (RU) as a single measure for the resources required to perform various database operations and service an application request.

  Throughput is provisioned based on the number of request units set for each container. Request unit consumption is evaluated as a rate per second. Applications that exceed the provisioned request unit rate for their container are limited until the rate drops below the provisioned level for the container. If your application requires a higher level of throughput, you can increase your throughput by provisioning additional request units.

The complexity of a query impacts how many request units are consumed for an operation. The number of predicates, nature of the predicates, number of UDFs, and the size of the source data set all influence the cost of query operations.

To measure the overhead of any operation (create, update, or delete), inspect the x-ms-request-charge header to measure the number of request units consumed by these operations. You can also look at the equivalent RequestCharge property in ResourceResponse<T> or FeedResponse<T>.

Async Java SDK V2 (Maven com.microsoft.azure::azure-cosmosdb)

```
ResourceResponse<Document> response = asyncClient.createDocument(collectionLink, documentDefinition,
null,
                                                    false).toBlocking.single();
response.getRequestCharge();
```

The request charge returned in this header is a fraction of your provisioned throughput. For example, if you have 2000 RU/s provisioned, and if the preceding query returns 1000 1KB-documents, the cost of the operation is 1000. As such, within one second, the server honors only two such requests before rate limiting subsequent requests. For more information, see Request units and the request unit calculator.

- **Handle rate limiting/request rate too large**

  When a client attempts to exceed the reserved throughput for an account, there is no performance degradation at the server and no use of throughput capacity beyond the reserved level. The server will preemptively end the request with RequestRateTooLarge (HTTP status code 429) and return the x-ms-retry-after-ms header indicating the amount of time, in milliseconds, that the user must wait before reattempting the request.

  ```
  HTTP Status 429,
  Status Line: RequestRateTooLarge
  x-ms-retry-after-ms :100
  ```

  The SDKs all implicitly catch this response, respect the server-specified retry-after header, and retry the request. Unless your account is being accessed concurrently by multiple clients, the next retry will succeed.

  If you have more than one client cumulatively operating consistently above the request rate, the default retry count currently set to 9 internally by the client may not suffice; in this case, the client throws a DocumentClientException with status code 429 to the application. The default retry count can be changed by using setRetryOptions on the ConnectionPolicy instance. By default, the DocumentClientException with status code 429 is returned after a cumulative wait time of 30 seconds if the request continues to operate above the request rate. This occurs even when the current retry count is less than the max retry count, be it the default of 9 or a user-defined value.

  While the automated retry behavior helps to improve resiliency and usability for the most applications, it might come at odds when doing performance benchmarks, especially when measuring latency. The client-observed latency will spike if the experiment hits the server throttle and causes the client SDK to silently retry. To avoid latency spikes during performance experiments, measure the charge returned by each operation and ensure that requests are operating below the reserved request rate. For more information, see Request units.

- **Design for smaller documents for higher throughput**

  The request charge (the request processing cost) of a given operation is directly correlated to the size of the document. Operations on large documents cost more than operations for small documents.

Next steps

To learn more about designing your application for scale and high performance, see Partitioning and scaling in Azure Cosmos DB.

Performance tips for Azure Cosmos DB Sync Java SDK v2

12/4/2020 • 10 minutes to read • Edit Online

APPLIES TO: SQL API

> **IMPORTANT**
>
> This is *not* the latest Java SDK for Azure Cosmos DB! You should upgrade your project to Azure Cosmos DB Java SDK v4 and then read the Azure Cosmos DB Java SDK v4 performance tips guide. Follow the instructions in the Migrate to Azure Cosmos DB Java SDK v4 guide and Reactor vs RxJava guide to upgrade.
>
> These performance tips are for Azure Cosmos DB Sync Java SDK v2 only. Please view the Azure Cosmos DB Sync Java SDK v2 Release notes and Maven repository for more information.

Azure Cosmos DB is a fast and flexible distributed database that scales seamlessly with guaranteed latency and throughput. You do not have to make major architecture changes or write complex code to scale your database with Azure Cosmos DB. Scaling up and down is as easy as making a single API call. To learn more, see how to provision container throughput or how to provision database throughput. However, because Azure Cosmos DB is accessed via network calls there are client-side optimizations you can make to achieve peak performance when using Azure Cosmos DB Sync Java SDK v2.

So if you're asking "How can I improve my database performance?" consider the following options:

Networking

1. **Connection mode: Use DirectHttps**

   How a client connects to Azure Cosmos DB has important implications on performance, especially in terms of observed client-side latency. There is one key configuration setting available for configuring the client ConnectionPolicy – the ConnectionMode. The two available ConnectionModes are:

   a. Gateway (default)

   b. DirectHttps

      Gateway mode is supported on all SDK platforms and is the configured default. If your application runs within a corporate network with strict firewall restrictions, Gateway is the best choice since it uses the standard HTTPS port and a single endpoint. The performance tradeoff, however, is that Gateway mode involves an additional network hop every time data is read or written to Azure Cosmos DB. Because of this, DirectHttps mode offers better performance due to fewer network hops.

      The Azure Cosmos DB Sync Java SDK v2 uses HTTPS as a transport protocol. HTTPS uses TLS for initial authentication and encrypting traffic. When using the Azure Cosmos DB Sync Java SDK v2, only HTTPS port 443 needs to be open.

      The ConnectionMode is configured during the construction of the DocumentClient instance with the ConnectionPolicy parameter.

   Sync Java SDK V2 (Maven com.microsoft.azure::azure-documentdb)

```
public ConnectionPolicy getConnectionPolicy() {
  ConnectionPolicy policy = new ConnectionPolicy();
  policy.setConnectionMode(ConnectionMode.DirectHttps);
  policy.setMaxPoolSize(1000);
  return policy;
}

ConnectionPolicy connectionPolicy = new ConnectionPolicy();
DocumentClient client = new DocumentClient(HOST, MASTER_KEY, connectionPolicy, null);
```

2. **Collocate clients in same Azure region for performance**

   When possible, place any applications calling Azure Cosmos DB in the same region as the Azure Cosmos database. For an approximate comparison, calls to Azure Cosmos DB within the same region complete within 1-2 ms, but the latency between the West and East coast of the US is >50 ms. This latency can likely vary from request to request depending on the route taken by the request as it passes from the client to the Azure datacenter boundary. The lowest possible latency is achieved by ensuring the calling application is located within the same Azure region as the provisioned Azure Cosmos DB endpoint. For a list of available regions, see Azure Regions.

SDK Usage

1. **Install the most recent SDK**

   The Azure Cosmos DB SDKs are constantly being improved to provide the best performance. See the Azure Cosmos DB SDK pages to determine the most recent SDK and review improvements.

2. **Use a singleton Azure Cosmos DB client for the lifetime of your application**

Each DocumentClient instance is thread-safe and performs efficient connection management and address caching when operating in Direct Mode. To allow efficient connection management and better performance by DocumentClient, it is recommended to use a single instance of DocumentClient per AppDomain for the lifetime of the application.

3. **Increase MaxPoolSize per host when using Gateway mode**

   Azure Cosmos DB requests are made over HTTPS/REST when using Gateway mode, and are subjected to the default connection limit per hostname or IP address. You may need to set the MaxPoolSize to a higher value (200-1000) so that the client library can utilize multiple simultaneous connections to Azure Cosmos DB. In the Azure Cosmos DB Sync Java SDK v2, the default value for ConnectionPolicy.getMaxPoolSize is 100. Use setMaxPoolSize to change the value.

4. **Tuning parallel queries for partitioned collections**

   Azure Cosmos DB Sync Java SDK version 1.9.0 and above support parallel queries, which enable you to query a partitioned collection in parallel. For more information, see code samples related to working with the SDKs. Parallel queries are designed to improve query latency and throughput over their serial counterpart.

   (a) ***Tuning setMaxDegreeOfParallelism:*** Parallel queries work by querying multiple partitions in parallel. However, data from an individual partitioned collection is fetched serially with respect to the query. So, use setMaxDegreeOfParallelism to set the number of partitions that has the maximum chance of achieving the most performant query, provided all other system conditions remain the same. If you don't know the number of partitions, you can use setMaxDegreeOfParallelism to set a high number, and the system chooses the minimum (number of partitions, user provided input) as the maximum degree of parallelism.

   It is important to note that parallel queries produce the best benefits if the data is evenly distributed across all partitions with respect to the query. If the partitioned collection is partitioned such a way that all or a majority of the data returned by a query is concentrated in a few partitions (one partition in worst case), then the performance of the query would be bottlenecked by those partitions.

   (b) ***Tuning setMaxBufferedItemCount:*** Parallel query is designed to pre-fetch results while the current batch of results is being processed by the client. The pre-fetching helps in overall latency improvement of a query. setMaxBufferedItemCount limits the number of pre-fetched results. By setting setMaxBufferedItemCount to the expected number of results returned (or a higher number), this enables the query to receive maximum benefit from pre-fetching.

   Pre-fetching works the same way irrespective of the MaxDegreeOfParallelism, and there is a single buffer for the data from all partitions.

5. **Implement backoff at getRetryAfterInMilliseconds intervals**

   During performance testing, you should increase load until a small rate of requests get throttled. If throttled, the client application should backoff on throttle for the server-specified retry interval. Respecting the backoff ensures that you spend minimal amount of time waiting between retries. Retry policy support is included in Version 1.8.0 and above of the Azure Cosmos DB Sync Java SDK. For more information, see getRetryAfterInMilliseconds.

6. **Scale out your client-workload**

   If you are testing at high throughput levels (>50,000 RU/s), the client application may become the bottleneck due to the machine capping out on CPU or network utilization. If you reach this point, you can continue to push the Azure Cosmos DB account further by scaling out your client applications across multiple servers.

7. **Use name based addressing**

Use name-based addressing, where links have the format `dbs/MyDatabaseId/colls/MyCollectionId/docs/MyDocumentId`, instead of SelfLinks (_self), which have the format `dbs/<database_rid>/colls/<collection_rid>/docs/<document_rid>` to avoid retrieving ResourceIds of all the resources used to construct the link. Also, as these resources get recreated (possibly with same name), caching these may not help.

8. **Tune the page size for queries/read feeds for better performance**

   When performing a bulk read of documents by using read feed functionality (for example, readDocuments) or when issuing a SQL query, the results are returned in a segmented fashion if the result set is too large. By default, results are returned in chunks of 100 items or 1 MB, whichever limit is hit first.

   To reduce the number of network round trips required to retrieve all applicable results, you can increase the page size using the x-ms-max-item-count request header to up to 1000. In cases where you need to display only a few results, for example, if your user interface or application API returns only 10 results a time, you can also decrease the page size to 10 to reduce the throughput consumed for reads and queries.

   You may also set the page size using the setPageSize method.

Indexing Policy

1. **Exclude unused paths from indexing for faster writes**

   Azure Cosmos DB's indexing policy allows you to specify which document paths to include or exclude from indexing by leveraging Indexing Paths (setIncludedPaths and setExcludedPaths). The use of indexing paths can offer improved write performance and lower index storage for scenarios in which the query patterns are known beforehand, as indexing costs are directly correlated to the number of unique paths indexed. For example, the following code shows how to exclude an entire section (subtree) of the documents from indexing using the "*" wildcard.

   Sync Java SDK V2 (Maven com.microsoft.azure::azure-documentdb)

   ```
   Index numberIndex = Index.Range(DataType.Number);
   numberIndex.set("precision", -1);
   indexes.add(numberIndex);
   includedPath.setIndexes(indexes);
   includedPaths.add(includedPath);
   indexingPolicy.setIncludedPaths(includedPaths);
   collectionDefinition.setIndexingPolicy(indexingPolicy);
   ```

   For more information, see Azure Cosmos DB indexing policies.

Throughput

1. **Measure and tune for lower request units/second usage**

   Azure Cosmos DB offers a rich set of database operations including relational and hierarchical queries with UDFs, stored procedures, and triggers – all operating on the documents within a database collection. The cost associated with each of these operations varies based on the CPU, IO, and memory required to complete the operation. Instead of thinking about and managing hardware resources, you can think of a request unit (RU) as a single measure for the resources required to perform various database operations and service an application request.

   Throughput is provisioned based on the number of request units set for each container. Request unit consumption is evaluated as a rate per second. Applications that exceed the provisioned request unit rate for their container are limited until the rate drops below the provisioned level for the container. If your application requires a higher level of throughput, you can increase your throughput by provisioning

additional request units.

The complexity of a query impacts how many request units are consumed for an operation. The number of predicates, nature of the predicates, number of UDFs, and the size of the source data set all influence the cost of query operations.

To measure the overhead of any operation (create, update, or delete), inspect the x-ms-request-charge header (or the equivalent RequestCharge property in ResourceResponse<T> or FeedResponse<T> to measure the number of request units consumed by these operations.

Sync Java SDK V2 (Maven com.microsoft.azure::azure-documentdb)

```
ResourceResponse<Document> response = client.createDocument(collectionLink, documentDefinition, null,
false);

response.getRequestCharge();
```

The request charge returned in this header is a fraction of your provisioned throughput. For example, if you have 2000 RU/s provisioned, and if the preceding query returns 1000 1KB-documents, the cost of the operation is 1000. As such, within one second, the server honors only two such requests before rate limiting subsequent requests. For more information, see Request units and the request unit calculator.

2. **Handle rate limiting/request rate too large**

   When a client attempts to exceed the reserved throughput for an account, there is no performance degradation at the server and no use of throughput capacity beyond the reserved level. The server will preemptively end the request with RequestRateTooLarge (HTTP status code 429) and return the x-ms-retry-after-ms header indicating the amount of time, in milliseconds, that the user must wait before reattempting the request.

```
HTTP Status 429,
Status Line: RequestRateTooLarge
x-ms-retry-after-ms :100
```

   The SDKs all implicitly catch this response, respect the server-specified retry-after header, and retry the request. Unless your account is being accessed concurrently by multiple clients, the next retry will succeed.

   If you have more than one client cumulatively operating consistently above the request rate, the default retry count currently set to 9 internally by the client may not suffice; in this case, the client throws a DocumentClientException with status code 429 to the application. The default retry count can be changed by using setRetryOptions on the ConnectionPolicy instance. By default, the DocumentClientException with status code 429 is returned after a cumulative wait time of 30 seconds if the request continues to operate above the request rate. This occurs even when the current retry count is less than the max retry count, be it the default of 9 or a user-defined value.

   While the automated retry behavior helps to improve resiliency and usability for the most applications, it might come at odds when doing performance benchmarks, especially when measuring latency. The client-observed latency will spike if the experiment hits the server throttle and causes the client SDK to silently retry. To avoid latency spikes during performance experiments, measure the charge returned by each operation and ensure that requests are operating below the reserved request rate. For more information, see Request units.

3. **Design for smaller documents for higher throughput**

   The request charge (the request processing cost) of a given operation is directly correlated to the size of the document. Operations on large documents cost more than operations for small documents.

Next steps

To learn more about designing your application for scale and high performance, see Partitioning and scaling in Azure Cosmos DB.

Performance and scale testing with Azure Cosmos DB

12/4/2020 • 4 minutes to read • Edit Online

APPLIES TO: SQL API

Performance and scale testing is a key step in application development. For many applications, the database tier has a significant impact on overall performance and scalability. Therefore, it's a critical component of performance testing. Azure Cosmos DB is purpose-built for elastic scale and predictable performance. These capabilities make it a great fit for applications that need a high-performance database tier.

This article is a reference for developers implementing performance test suites for their Azure Cosmos DB workloads. It also can be used to evaluate Azure Cosmos DB for high-performance application scenarios. It focuses primarily on isolated performance testing of the database, but also includes best practices for production applications.

After reading this article, you'll be able to answer the following questions:

- Where can I find a sample .NET client application for performance testing of Azure Cosmos DB?
- How do I achieve high throughput levels with Azure Cosmos DB from my client application?

To get started with code, download the project from Azure Cosmos DB performance testing sample.

> **NOTE**
> The goal of this application is to demonstrate how to get the best performance from Azure Cosmos DB with a small number of client machines. The goal of the sample is not to achieve the peak throughput capacity of Azure Cosmos DB (which can scale without any limits).

If you're looking for client-side configuration options to improve Azure Cosmos DB performance, see Azure Cosmos DB performance tips.

Run the performance testing application

The quickest way to get started is to compile and run the .NET sample, as described in the following steps. You can also review the source code and implement similar configurations on your own client applications.

Step 1: Download the project from Azure Cosmos DB performance testing sample, or fork the GitHub repository.

Step 2: Modify the settings for EndpointUrl, AuthorizationKey, CollectionThroughput, and DocumentTemplate (optional) in App.config.

> **NOTE**
> Before you provision collections with high throughput, refer to the Pricing page to estimate the costs per collection. Azure Cosmos DB bills storage and throughput independently on an hourly basis. You can save costs by deleting or lowering the throughput of your Azure Cosmos containers after testing.

Step 3: Compile and run the console app from the command line. You should see output like the following:

```
C:\Users\cosmosdb\Desktop\Benchmark>DocumentDBBenchmark.exe
Summary:
---------------------------------------------------------------------
Endpoint: https://arramacquerymetrics.documents.azure.com:443/
Collection : db.data at 100000 request units per second
Document Template*: Player.json
Degree of parallelism*: -1
---------------------------------------------------------------------
DocumentDBBenchmark starting...
Found collection data with 100000 RU/s
Starting Inserts with 100 tasks
Inserted 4503 docs @ 4491 writes/s, 47070 RU/s (122B max monthly 1KB reads)
Inserted 17910 docs @ 8862 writes/s, 92878 RU/s (241B max monthly 1KB reads)
Inserted 32339 docs @ 10531 writes/s, 110366 RU/s (286B max monthly 1KB reads)
Inserted 47848 docs @ 11675 writes/s, 122357 RU/s (317B max monthly 1KB reads)
Inserted 58857 docs @ 11545 writes/s, 120992 RU/s (314B max monthly 1KB reads)
Inserted 69547 docs @ 11378 writes/s, 119237 RU/s (309B max monthly 1KB reads)
Inserted 80687 docs @ 11345 writes/s, 118896 RU/s (308B max monthly 1KB reads)
Inserted 91455 docs @ 11272 writes/s, 118131 RU/s (306B max monthly 1KB reads)
Inserted 102129 docs @ 11208 writes/s, 117461 RU/s (304B max monthly 1KB reads)
Inserted 112444 docs @ 11120 writes/s, 116538 RU/s (302B max monthly 1KB reads)
Inserted 122927 docs @ 11063 writes/s, 115936 RU/s (301B max monthly 1KB reads)
Inserted 133157 docs @ 10993 writes/s, 115208 RU/s (299B max monthly 1KB reads)
Inserted 144078 docs @ 10988 writes/s, 115159 RU/s (298B max monthly 1KB reads)
Inserted 155415 docs @ 11013 writes/s, 115415 RU/s (299B max monthly 1KB reads)
Inserted 166126 docs @ 10992 writes/s, 115198 RU/s (299B max monthly 1KB reads)
Inserted 173051 docs @ 10739 writes/s, 112544 RU/s (292B max monthly 1KB reads)
Inserted 180169 docs @ 10527 writes/s, 110324 RU/s (286B max monthly 1KB reads)
Inserted 192469 docs @ 10616 writes/s, 111256 RU/s (288B max monthly 1KB reads)
Inserted 199107 docs @ 10406 writes/s, 109054 RU/s (283B max monthly 1KB reads)
Inserted 200000 docs @ 9930 writes/s, 104065 RU/s (270B max monthly 1KB reads)

Summary:
---------------------------------------------------------------------
Inserted 200000 docs @ 9928 writes/s, 104063 RU/s (270B max monthly 1KB reads)
---------------------------------------------------------------------
DocumentDBBenchmark completed successfully.
Press any key to exit...
```

Step 4 (if necessary): The throughput reported (RU/s) from the tool should be the same or higher than the provisioned throughput of the collection or a set of collections. If it's not, increasing the DegreeOfParallelism in small increments might help you reach the limit. If the throughput from your client app plateaus, start multiple instances of the app on additional client machines. If you need help with this step file a support ticket from the Azure portal.

After you have the app running, you can try different indexing policies and consistency levels to understand their impact on throughput and latency. You can also review the source code and implement similar configurations to your own test suites or production applications.

Next steps

In this article, we looked at how you can perform performance and scale testing with Azure Cosmos DB by using a .NET console app. For more information, see the following articles:

- Azure Cosmos DB performance testing sample
- Client configuration options to improve Azure Cosmos DB performance
- Server-side partitioning in Azure Cosmos DB

Azure Cosmos DB as a key value store – cost overview

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: SQL API

Azure Cosmos DB is a globally distributed, multi-model database service for building highly available, large-scale applications easily. By default, Azure Cosmos DB automatically and efficiently indexes all the data it ingests. This enables fast and consistent SQL (and JavaScript) queries on the data.

This article describes the cost of Azure Cosmos DB for simple write and read operations when it's used as a key/value store. Write operations include inserts, replaces, deletes, and upserts of data items. Besides guaranteeing a 99.999% availability SLA for all multi-region accounts, Azure Cosmos DB offers guaranteed <10-ms latency for reads and for the (indexed) writes, at the 99th percentile.

Why we use Request Units (RUs)

Azure Cosmos DB performance is based on the amount of provisioned throughput expressed in Request Units (RU/s). The provisioning is at a second granularity and is purchased in RU/s (not to be confused with the hourly billing). RUs should be considered as a logical abstraction (a currency) that simplifies the provisioning of required throughput for the application. Users do not have to think of differentiating between read and write throughput. The single currency model of RUs creates efficiencies to share the provisioned capacity between reads and writes. This provisioned capacity model enables the service to provide a **predictable and consistent throughput, guaranteed low latency, and high availability**. Finally, while RU model is used to depict throughput, each provisioned RU also has a defined amount of resources (e.g., memory, cores/CPU and IOPS).

As a globally distributed database system, Cosmos DB is the only Azure service that provides comprehensive SLAs covering latency, throughput, consistency and high availability. The throughput you provision is applied to each of the regions associated with your Cosmos account. For reads, Cosmos DB offers multiple, well-defined consistency levels for you to choose from.

The following table shows the number of RUs required to perform read and write operations based on a data item of size 1 KB and 100 KBs with default automatic indexing turned off.

ITEM SIZE	1 READ	1 WRITE
1 KB	1 RU	5 RUs
100 KB	10 RUs	50 RUs

Cost of reads and writes

If you provision 1,000 RU/s, this amounts to 3.6 million RU/hour and will cost $0.08 for the hour (in the US and Europe). For a 1 KB size data item, this means that you can consume 3.6 million reads or 0.72 million writes (3.6 million RU / 5) using your provisioned throughput. Normalized to million reads and writes, the cost would be $0.022 /million of reads ($0.08 / 3.6) and $0.111/million of writes ($0.08 / 0.72). The cost per million becomes minimal as shown in the table below.

ITEM SIZE	COST OF 1 MILLION READS	COST OF 1 MILLION WRITES
1 KB	$0.022	$0.111
100 KB	$0.222	$1.111

Most of the basic blob or object stores services charge $0.40 per million read transaction and $5 per million write transaction. If used optimally, Cosmos DB can be up to 98% cheaper than these other solutions (for 1 KB transactions).

Next steps

- Use RU calculator to estimate throughput for your workloads.

Migrate your application to use the Azure Cosmos DB Java SDK v4

12/4/2020 • 10 minutes to read • Edit Online

APPLIES TO: SQL API

> **IMPORTANT**
>
> For more information about this SDK, please view the Azure Cosmos DB Java SDK v4 Release notes, Maven repository, Azure Cosmos DB Java SDK v4 performance tips, and Azure Cosmos DB Java SDK v4 troubleshooting guide.

This article explains how to upgrade your existing Java application that is using an older Azure Cosmos DB Java SDK to the newer Azure Cosmos DB Java SDK 4.0 for Core (SQL) API. Azure Cosmos DB Java SDK v4 corresponds to the `com.azure.cosmos` package. You can use the instructions in this doc if you are migrating your application from any of the following Azure Cosmos DB Java SDKs:

- Sync Java SDK 2.x.x
- Async Java SDK 2.x.x
- Java SDK 3.x.x

Azure Cosmos DB Java SDK's and package mappings

The following table lists different Azure Cosmos DB Java SDKs, the package name and the release information:

JAVA SDK	RELEASE DATE	BUNDLED APIS	MAVEN JAR	JAVA PACKAGE NAME	API REFERENCE	RELEASE NOTES
Async 2.x.x	June 2018	Async(RxJava)	`com.microsoft.az` `cosmosdb`	`com.microsoft.azure.cosmosdb.rx`	API	Release Notes
Sync 2.x.x	Sept 2018	Sync	`com.microsoft.az` `documentdb`	`com.microsoft.azure.cosmosdb`	API	Release Notes
3.x.x	July 2019	Async(Reactor)/Sync	`com.microsoft.az` `cosmos`	`com.azure.data.cosmos`	API	-
4.0	June 2020	Async(Reactor)/Sync	`com.azure::azure` `cosmos`	`com.azure.cosmos`	API	API

SDK level implementation changes

The following are the key implementation differences between different SDKs:

RxJava is replaced with reactor in Azure Cosmos DB Java SDK versions 3.x.x and 4.0

If you are unfamiliar with asynchronous programming or Reactive Programming, see the Reactor pattern guide for an introduction to async programming and Project Reactor. This guide may be useful if you have been using Azure Cosmos DB Sync Java SDK 2.x.x or Azure Cosmos DB Java SDK 3.x.x Sync API in the past.

If you have been using Azure Cosmos DB Async Java SDK 2.x.x, and you plan on migrating to the 4.0 SDK, see the

Reactor vs RxJava Guide for guidance on converting RxJava code to use Reactor.

Azure Cosmos DB Java SDK v4 has direct connectivity mode in both Async and Sync APIs

If you have been using Azure Cosmos DB Sync Java SDK 2.x.x, note that the direct connection mode based on TCP (as opposed to HTTP) is implemented in Azure Cosmos DB Java SDK 4.0 for both the Async and Sync APIs.

API level changes

The following are the API level changes in Azure Cosmos DB Java SDK 4.x.x compared to previous SDKs (Java SDK 3.x.x, Async Java SDK 2.x.x, and Sync Java SDK 2.x.x):

- **Java SDK 3.x.x and 4.x.x**

Java SDK Version	Sync API	Async API
Java SDK 4.0	CosmosClient	CosmosAsyncClient
Java SDK 3.x.x	CosmosSyncClient	CosmosClient

- **Async Java SDK 2.x.x and "Legacy" Sync Java SDK 2.x.x**

Java SDK Version	Class Name
Async Java SDK 2.x.x	AsyncDocumentClient
"Legacy" Sync Java SDK 2.x.x	DocumentClient

- The Azure Cosmos DB Java SDK 3.x.x and 4.0 refer the client resources as `Cosmos<resourceName>`. For example, `CosmosClient`, `CosmosDatabase`, `CosmosContainer`. Whereas in version 2.x.x, the Azure Cosmos DB Java SDKs don't have a uniform naming scheme.
- Azure Cosmos DB Java SDK 3.x.x and 4.0 offer both Sync and Async APIs.
  - **Java SDK 4.0** : All the classes belong to the Sync API unless the class name is appended with `Async` after `Cosmos`.
  - **Java SDK 3.x.x**: All the classes belong to the Async API unless the class name is appended with `Async` after `Cosmos`.
  - **Async Java SDK 2.x.x**: The class names are similar to Sync Java SDK 2.x.x, however the name starts with *Async*.

Hierarchical API structure

Azure Cosmos DB Java SDK 4.0 and 3.x.x introduce a hierarchical API structure that organizes the clients, databases, and containers in a nested fashion as shown in the following 4.0 SDK code snippet:

```
CosmosContainer container = client.getDatabase("MyDatabaseName").getContainer("MyContainerName");
```

In version 2.x.x of the Azure Cosmos DB Java SDK, all operations on resources and documents are performed through the client instance.

Representing documents

In Azure Cosmos DB Java SDK 4.0, custom POJO's and `JsonNodes` are the two options to read and write the documents from Azure Cosmos DB.

In the Azure Cosmos DB Java SDK 3.x.x, the `CosmosItemProperties` object is exposed by the public API and served as a document representation. This class is no longer exposed publicly in version 4.0.

Imports

- The Azure Cosmos DB Java SDK 4.0 packages begin with `com.azure.cosmos`
  - Azure Cosmos DB Java SDK 3.x.x packages begin with `com.azure.data.cosmos`
- Azure Cosmos DB Java SDK 4.0 places several classes in a nested package `com.azure.cosmos.models`. Some of these packages include:
  - `CosmosContainerResponse`
  - `CosmosDatabaseResponse`
  - `CosmosItemResponse`
  - The Async API analogs for all of the above packages
  - `CosmosContainerProperties`
  - `FeedOptions`
  - `PartitionKey`
  - `IndexingPolicy`
  - `IndexingMode` ...etc.

Accessors

Azure Cosmos DB Java SDK 4.0 exposes `get` and `set` methods to access the instance members. For example, the `CosmosContainer` instance has `container.getId()` and `container.setId()` methods.

This is different from Azure Cosmos DB Java SDK 3.x.x which exposes a fluent interface. For example, a `CosmosSyncContainer` instance has `container.id()` which is overloaded to get or set the `id` value.

Code snippet comparisons

Create resources

The following code snippet shows the differences in how resources are created between the 4.0 and 3.x.x Async APIs:

- Java SDK 4.0 Async API
- Java SDK 3.x.x Async API

```
// Create Async client.
// Building an async client is still a sync operation.
CosmosAsyncClient client = new CosmosClientBuilder()
        .endpoint("your.hostname")
        .key("yourmasterkey")
        .consistencyLevel(ConsistencyLevel.EVENTUAL)
        .buildAsyncClient();

// Create database with specified name
client.createDatabaseIfNotExists("YourDatabaseName")
        .flatMap(databaseResponse -> {
            testDatabaseAsync = client.getDatabase("YourDatabaseName");
            // Container properties - name and partition key
            CosmosContainerProperties containerProperties =
                    new CosmosContainerProperties("YourContainerName", "/id");

            // Provision manual throughput
            ThroughputProperties throughputProperties = ThroughputProperties.createManualThroughput(400);

            // Create container
            return database.createContainerIfNotExists(containerProperties, throughputProperties);
        }).flatMap(containerResponse -> {
    testContainerAsync = database.getContainer("YourContainerName");
    return Mono.empty();
}).subscribe();
```

Item operations

The following code snippet shows the differences in how item operations are performed between the 4.0 and 3.x.x Async APIs:

- Java SDK 4.0 Async API
- Java SDK 3.x.x Async API

```
// Container is created. Generate many docs to insert.
int number_of_docs = 50000;
ArrayList<JsonNode> docs = generateManyDocs(number_of_docs);

// Insert many docs into container...
Flux.fromIterable(docs)
        .flatMap(doc -> testContainerAsync.createItem(doc))
        .subscribe(); // ...Subscribing triggers stream execution.
```

Indexing

The following code snippet shows the differences in how indexing is created between the 4.0 and 3.x.x Async APIs:

- Java SDK 4.0 Async API
- Java SDK 3.x.x Async API

```
CosmosContainerProperties containerProperties = new CosmosContainerProperties(containerName, "/lastName");

// Custom indexing policy
IndexingPolicy indexingPolicy = new IndexingPolicy();
indexingPolicy.setIndexingMode(IndexingMode.CONSISTENT);

// Included paths
List<IncludedPath> includedPaths = new ArrayList<>();
includedPaths.add(new IncludedPath("/*"));
indexingPolicy.setIncludedPaths(includedPaths);

// Excluded paths
List<ExcludedPath> excludedPaths = new ArrayList<>();
excludedPaths.add(new ExcludedPath("/name/*"));
indexingPolicy.setExcludedPaths(excludedPaths);

containerProperties.setIndexingPolicy(indexingPolicy);

ThroughputProperties throughputProperties = ThroughputProperties.createManualThroughput(400);

database.createContainerIfNotExists(containerProperties, throughputProperties);
CosmosAsyncContainer containerIfNotExists = database.getContainer(containerName);
```

Stored procedures

The following code snippet shows the differences in how stored procedures are created between the 4.0 and 3.x.x Async APIs:

- Java SDK 4.0 Async API
- Java SDK 3.x.x Async API

```
logger.info("Creating stored procedure...\n");

String sprocId = "createMyDocument";

String sprocBody = "function createMyDocument() {\n" +
        "var documentToCreate = {\"id\":\"test_doc\"}\n" +
        "var context = getContext();\n" +
        "var collection = context.getCollection();\n" +
        "var accepted = collection.createDocument(collection.getSelfLink(), documentToCreate,\n" +
        "    function (err, documentCreated) {\n" +
        "if (err) throw new Error('Error' + err.message);\n" +
        "context.getResponse().setBody(documentCreated.id)\n" +
        "});\n" +
        "if (!accepted) return;\n" +
        "}";

CosmosStoredProcedureProperties storedProcedureDef = new CosmosStoredProcedureProperties(sprocId,
sprocBody);
container.getScripts()
        .createStoredProcedure(storedProcedureDef,
                new CosmosStoredProcedureRequestOptions()).block();

// ...

logger.info(String.format("Executing stored procedure %s...\n\n", sprocId));

CosmosStoredProcedureRequestOptions options = new CosmosStoredProcedureRequestOptions();
options.setPartitionKey(new PartitionKey("test_doc"));

container.getScripts()
        .getStoredProcedure(sprocId)
        .execute(null, options)
        .flatMap(executeResponse -> {
            logger.info(String.format("Stored procedure %s returned %s (HTTP %d), at cost %.3f RU.\n",
                    sprocId,
                    executeResponse.getResponseAsString(),
                    executeResponse.getStatusCode(),
                    executeResponse.getRequestCharge()));
            return Mono.empty();
        }).block();
```

Change feed

The following code snippet shows the differences in how change feed operations are executed between the 4.0 and 3.x.x Async APIs:

- Java SDK 4.0 Async API
- Java SDK 3.x.x Async API

```
ChangeFeedProcessor changeFeedProcessorInstance =
        new ChangeFeedProcessorBuilder()
                .hostName(hostName)
                .feedContainer(feedContainer)
                .leaseContainer(leaseContainer)
                .handleChanges((List<JsonNode> docs) -> {
                    logger.info("--->setHandleChanges() START");

                    for (JsonNode document : docs) {
                        try {
                            //Change Feed hands the document to you in the form of a JsonNode
                            //As a developer you have two options for handling the JsonNode document
provided to you by Change Feed
                            //One option is to operate on the document in the form of a JsonNode, as shown
below. This is great
                            //especially if you do not have a single uniform data model for all documents.
                            logger.info("---->DOCUMENT RECEIVED: " +
OBJECT_MAPPER.writerWithDefaultPrettyPrinter()
                                    .writeValueAsString(document));

                            //You can also transform the JsonNode to a POJO having the same structure as the
JsonNode,
                            //as shown below. Then you can operate on the POJO.
                            CustomPOJO pojo_doc = OBJECT_MAPPER.treeToValue(document, CustomPOJO.class);
                            logger.info("---->id: " + pojo_doc.getId());

                        } catch (JsonProcessingException e) {
                            e.printStackTrace();
                        }
                    }
                    logger.info("--->handleChanges() END");

                })
                .buildChangeFeedProcessor();

// ...

changeFeedProcessorInstance.start()
        .subscribeOn(Schedulers.elastic())
        .subscribe();
```

Container level Time-To-Live(TTL)

The following code snippet shows the differences in how to create time to live for data in the container using the 4.0 and 3.x.x Async APIs:

- Java SDK 4.0 Async API
- Java SDK 3.x.x Async API

```
CosmosAsyncContainer container;

// Create a new container with TTL enabled with default expiration value
CosmosContainerProperties containerProperties = new CosmosContainerProperties("myContainer",
"/myPartitionKey");
containerProperties.setDefaultTimeToLiveInSeconds(90 * 60 * 60 * 24);
ThroughputProperties throughputProperties = ThroughputProperties.createManualThroughput(400);
database.createContainerIfNotExists(containerProperties, throughputProperties).block();
container = database.getContainer("myContainer");
```

Item level Time-To-Live(TTL)

The following code snippet shows the differences in how to create time to live for an item using the 4.0 and 3.x.x Async APIs:

- Java SDK 4.0 Async API
- Java SDK 3.x.x Async API

```
// Include a property that serializes to "ttl" in JSON
class SalesOrder
{
    private String id;
    private String customerId;
    private Integer ttl;

    public SalesOrder(String id, String customerId, Integer ttl) {
        this.id = id;
        this.customerId = customerId;
        this.ttl = ttl;
    }

    public String getId() {return this.id;}
    public void setId(String new_id) {this.id = new_id;}
    public String getCustomerId() {return this.customerId;}
    public void setCustomerId(String new_cid) {this.customerId = new_cid;}
    public Integer getTtl() {return this.ttl;}
    public void setTtl(Integer new_ttl) {this.ttl = new_ttl;}

    //...
}
```

```
// Set the value to the expiration in seconds
SalesOrder salesOrder = new SalesOrder(
        "SO05",
        "CO18009186470",
        60 * 60 * 24 * 30  // Expire sales orders in 30 days
);
```

Next steps

- Build a Java app to manage Azure Cosmos DB SQL API data using the V4 SDK
- Learn about the Reactor-based Java SDKs
- Learn about converting RxJava async code to Reactor async code with the Reactor vs RxJava Guide

Migrate your application to use the Azure Cosmos DB .NET SDK v3

12/4/2020 • 11 minutes to read • Edit Online

APPLIES TO: SQL API

> **IMPORTANT**
>
> To learn about the Azure Cosmos DB .NET SDK v3, see the Release notes, the .NET GitHub repository, .NET SDK v3 Performance Tips, and the Troubleshooting guide.

This article highlights some of the considerations of upgrading your existing .NET application to the newer Azure Cosmos DB .NET SDK v3 for Core (SQL) API. Azure Cosmos DB .NET SDK v3 corresponds to the Microsoft.Azure.Cosmos namespace. You can use the information provided in this doc if you are migrating your application from any of the following Azure Cosmos DB .NET SDKs:

- Azure Cosmos DB .NET Framework SDK v2 for SQL API
- Azure Cosmos DB .NET Core SDK v2 for SQL API

The instructions in this article also help you to migrate the following external libraries that are now part of the Azure Cosmos DB .NET SDK v3 for Core (SQL) API:

- .NET change feed processor library 2.0
- .NET bulk executor library 1.1 or greater

What's new in the .NET V3 SDK

The v3 SDK contains many usability and performance improvements, including:

- Intuitive programming model naming
- .NET Standard 2.0 **
- Increased performance through stream API support
- Fluent hierarchy that replaces the need for URI factory
- Built-in support for change feed processor library
- Built-in support for bulk operations
- Mockabale APIs for easier unit testing
- Transactional batch and Blazor support
- Pluggable serializers
- Scale non-partitioned and autoscale containers

** The SDK targets .NET Standard 2.0 that unifies the existing Azure Cosmos DB .NET Framework and .NET Core SDKs into a single .NET SDK. You can use the .NET SDK in any platform that implements .NET Standard 2.0, including your .NET Framework 4.6.1+ and .NET Core 2.0+ applications.

Most of the networking, retry logic, and lower levels of the SDK remain largely unchanged.

The Azure Cosmos DB .NET SDK v3 is now open source. We welcome any pull requests and will be logging issues and tracking feedback on GitHub. We'll work on taking on any features that will improve customer experience.

Why migrate to the .NET v3 SDK

In addition to the numerous usability and performance improvements, new feature investments made in the latest SDK will not be back ported to older versions.

While there are no immediate plans to retire support for the 2.0 SDKs, the SDKs will be replaced by newer versions in the future and the SDK will go into maintenance mode. For the best development experience, we recommend always starting with the latest supported version of SDK.

Major name changes from v2 SDK to v3 SDK

The following name changes have been applied throughout the .NET 3.0 SDK to align with the API naming conventions for the Core (SQL) API:

- `DocumentClient` is renamed to `CosmosClient`
- `Collection` is renamed to `Container`
- `Document` is renamed to `Item`

All the resource objects are renamed with additional properties, which, includes the resource name for clarity.

The following are some of the main class name changes:

.NET V2 SDK	.NET V3 SDK
`Microsoft.Azure.Documents.Client.DocumentClient`	`Microsoft.Azure.CosmosClient`
`Microsoft.Azure.Documents.Client.ConnectionPolicy`	`Microsoft.Azure.Cosmos.CosmosClientOptions`
`Microsoft.Azure.Documents.Client.DocumentClientException`	`Microsoft.Azure.Cosmos.CosmosException`
`Microsoft.Azure.Documents.Client.Database`	`Microsoft.Azure.Cosmos.DatabaseProperties`
`Microsoft.Azure.Documents.Client.DocumentCollection`	`Microsoft.Azure.Cosmos.ContainerProperties`
`Microsoft.Azure.Documents.Client.RequestOptions`	`Microsoft.Azure.Cosmos.ItemRequestOptions`
`Microsoft.Azure.Documents.Client.FeedOptions`	`Microsoft.Azure.Cosmos.QueryRequestOptions`
`Microsoft.Azure.Documents.Client.StoredProcedure`	`Microsoft.Azure.Cosmos.StoredProcedureProperties`
`Microsoft.Azure.Documents.Client.Trigger`	`Microsoft.Azure.Cosmos.TriggerProperties`

Classes replaced on .NET v3 SDK

The following classes have been replaced on the 3.0 SDK:

- `Microsoft.Azure.Documents.UriFactory`
- `Microsoft.Azure.Documents.Document`
- `Microsoft.Azure.Documents.Resource`

The Microsoft.Azure.Documents.UriFactory class has been replaced by the fluent design. The fluent design builds URLs internally and allows a single `Container` object to be passed around instead of a `DocumentClient`, `DatabaseName`, and `DocumentCollection`.

Changes to item ID generation

Item ID is no longer auto populated in the .NET v3 SDK. Therefore, the Item ID must specifically include a generated ID. View the following example:

```
[JsonProperty(PropertyName = "id")]
public Guid Id { get; set; }
```

Changed default behavior for connection mode

The SDK v3 now defaults to Direct + TCP connection modes compared to the previous v2 SDK, which defaulted to Gateway + HTTPS connections modes. This change provides enhanced performance and scalability.

Changes to FeedOptions (QueryRequestOptions in v3.0 SDK)

The `FeedOptions` class in SDK v2 has now been renamed to `QueryRequestOptions` in the SDK v3 and within the class, several properties have had changes in name and/or default value or been removed completely.

`FeedOptions.MaxDegreeOfParallelism` has been renamed to `QueryRequestOptions.MaxConcurrency` and default value and associated behavior remains the same, operations run client side during parallel query executionwill be executed serially with no-parallelism.

`FeedOptions.EnableCrossPartitionQuery` has been removed and the default behavior in SDK 3.0 is that cross-partition queries will be executed without the need to enable the property specifically.

`FeedOptions.PopulateQueryMetrics` is enabled by default with the results being present in the diagnostics property of the response.

`FeedOptions.RequestContinuation` has now been promoted to the query methods themselves.

The following properties have been removed:

- `FeedOptions.DisableRUPerMinuteUsage`
- `FeedOptions.EnableCrossPartitionQuery`
- `FeedOptions.JsonSerializerSettings`
- `FeedOptions.PartitionKeyRangeId`
- `FeedOptions.PopulateQueryMetrics`

Constructing a client

The .NET SDK v3 provides a fluent `CosmosClientBuilder` class that replaces the need for the SDK v2 URI Factory.

The following example creates a new `CosmosClientBuilder` with a strong ConsistencyLevel and a list of preferred locations:

```
CosmosClientBuilder cosmosClientBuilder = new CosmosClientBuilder(
    accountEndpoint: "https://testcosmos.documents.azure.com:443/",
    authKeyOrResourceToken: "SuperSecretKey")
.WithConsistencyLevel(ConsistencyLevel.Strong)
.WithApplicationRegion(Regions.EastUS);
CosmosClient client = cosmosClientBuilder.Build();
```

Using the change feed processor APIs directly from the v3 SDK

The v3 SDK has built-in support for the Change Feed Processor APIs, allowing you use the same SDK for building your application and change feed processor implementation. Previously, you had to use a separate change feed processor library.

For more information, see how to migrate from the change feed processor library to the Azure Cosmos DB .NET v3 SDK

Using the bulk executor library directly from the V3 SDK

The v3 SDK has built-in support for the bulk executor library, allowing you to use the same SDK for building your application and performing bulk operations. Previously, you were required to use a separate bulk executor library.

For more information, see how to migrate from the bulk executor library to bulk support in Azure Cosmos DB .NET V3 SDK

Code snippet comparisons

The following code snippet shows the differences in how resources are created between the .NET v2 and v3 SDKs:

Database operations

Create a database

- .NET SDK v3
- .NET SDK v2

```
// Create database with no shared provisioned throughput
DatabaseResponse databaseResponse = await client.CreateDatabaseIfNotExistsAsync(DatabaseName);
Database database = databaseResponse;
DatabaseProperties databaseProperties = databaseResponse;

// Create a database with a shared manual provisioned throughput
string databaseIdManual = new string(DatabaseName + "_SharedManualThroughput");
database = await client.CreateDatabaseIfNotExistsAsync(databaseIdManual,
ThroughputProperties.CreateManualThroughput(400));

// Create a database with shared autoscale provisioned throughput
string databaseIdAutoscale = new string(DatabaseName + "_SharedAutoscaleThroughput");
database = await client.CreateDatabaseIfNotExistsAsync(databaseIdAutoscale,
ThroughputProperties.CreateAutoscaleThroughput(4000));
```

Read a database by ID

- .NET SDK v3
- .NET SDK v2

```
// Read a database
Console.WriteLine($"{Environment.NewLine} Read database resource: {DatabaseName}");
database = client.GetDatabase(DatabaseName);
Console.WriteLine($"{Environment.NewLine} database { database.Id.ToString()}");

// Read all databases
string findQueryText = "SELECT * FROM c";
using (FeedIterator<DatabaseProperties> feedIterator = client.GetDatabaseQueryIterator<DatabaseProperties>
(findQueryText))
{
    while (feedIterator.HasMoreResults)
    {
        FeedResponse<DatabaseProperties> databaseResponses = await feedIterator.ReadNextAsync();
        foreach (DatabaseProperties _database in databaseResponses)
        {
            Console.WriteLine($"{ Environment.NewLine} database {_database.Id.ToString()}");
        }
    }
}
```

Delete a database

- .NET SDK v3
- .NET SDK v2

```
// Delete a database
await client.GetDatabase(DatabaseName).DeleteAsync();
Console.WriteLine($"{ Environment.NewLine} database {DatabaseName} deleted.");

// Delete all databases in an account
string deleteQueryText = "SELECT * FROM c";
using (FeedIterator<DatabaseProperties> feedIterator = client.GetDatabaseQueryIterator<DatabaseProperties>
(deleteQueryText))
{
    while (feedIterator.HasMoreResults)
    {
        FeedResponse<DatabaseProperties> databaseResponses = await feedIterator.ReadNextAsync();
        foreach (DatabaseProperties _database in databaseResponses)
        {
            await client.GetDatabase(_database.Id).DeleteAsync();
            Console.WriteLine($"{ Environment.NewLine} database {_database.Id} deleted");
        }
    }
}
```

Container operations

Create a container (Autoscale + Time to live with expiration)

- .NET SDK v3
- .NET SDK v2

```
private static async Task CreateManualThroughputContainer(Database database)
{
    // Set throughput to the minimum value of 400 RU/s manually configured throughput
    string containerIdManual = ContainerName + "_Manual";
    ContainerResponse container = await database.CreateContainerIfNotExistsAsync(
        id: containerIdManual,
        partitionKeyPath: partitionKeyPath,
        throughput: 400);
}

// Create container with autoscale
private static async Task CreateAutoscaleThroughputContainer(Database database)
{
    string autoscaleContainerId = ContainerName + "_Autoscale";
    ContainerProperties containerProperties = new ContainerProperties(autoscaleContainerId, partitionKeyPath);

    Container container = await database.CreateContainerIfNotExistsAsync(
        containerProperties: containerProperties,
        throughputProperties: ThroughputProperties.CreateAutoscaleThroughput(autoscaleMaxThroughput: 4000);
}

// Create a container with TTL Expiration
private static async Task CreateContainerWithTtlExpiration(Database database)
{
    string containerIdManualwithTTL = ContainerName + "_ManualTTL";

    ContainerProperties properties = new ContainerProperties
        (id: containerIdManualwithTTL,
        partitionKeyPath: partitionKeyPath);

    properties.DefaultTimeToLive = (int)TimeSpan.FromDays(1).TotalSeconds; //expire in 1 day

    ContainerResponse containerResponse = await database.CreateContainerIfNotExistsAsync(containerProperties:
properties);
    ContainerProperties returnedProperties = containerResponse;
}
```

Read container properties

- .NET SDK v3
- .NET SDK v2

```
private static async Task ReadContainerProperties(Database database)
{
    string containerIdManual = ContainerName + "_Manual";
    Container container = database.GetContainer(containerIdManual);
    ContainerProperties containerProperties = await container.ReadContainerAsync();
}
```

Delete a container

- .NET SDK v3
- .NET SDK v2

```
private static async Task DeleteContainers(Database database)
{
    string containerIdManual = ContainerName + "_Manual";

    // Delete a container
    await database.GetContainer(containerIdManual).DeleteContainerAsync();

    // Delete all CosmosContainer resources for a database
    using (FeedIterator<ContainerProperties> feedIterator =
database.GetContainerQueryIterator<ContainerProperties>())
    {
        while (feedIterator.HasMoreResults)
        {
            foreach (ContainerProperties _container in await feedIterator.ReadNextAsync())
            {
                await database.GetContainer(_container.Id).DeleteContainerAsync();
                Console.WriteLine($"{Environment.NewLine}  deleted container {_container.Id}");
            }
        }
    }
}
```

Item and query operations

Create an item

- .NET SDK v3
- .NET SDK v2

```
private static async Task CreateItemAsync(Container container)
{
    // Create a SalesOrder POCO object
    SalesOrder salesOrder1 = GetSalesOrderSample("Account1", "SalesOrder1");
    ItemResponse<SalesOrder> response = await container.CreateItemAsync(salesOrder1,
        new PartitionKey(salesOrder1.AccountNumber));
}

private static async Task RunBasicOperationsOnDynamicObjects(Container container)
{
    // Dynamic Object
    dynamic salesOrder = new
    {
        id = "SalesOrder5",
        AccountNumber = "Account1",
        PurchaseOrderNumber = "PO18009186470",
        OrderDate = DateTime.UtcNow,
        Total = 5.95,
    };
    Console.WriteLine("\nCreating item");
    ItemResponse<dynamic> response = await container.CreateItemAsync<dynamic>(
        salesOrder, new PartitionKey(salesOrder.AccountNumber));
    dynamic createdSalesOrder = response.Resource;
}
```

Read all the items in a container

- .NET SDK v3
- .NET SDK v2

```
private static async Task ReadAllItems(Container container)
{
    // Read all items in a container
    List<SalesOrder> allSalesForAccount1 = new List<SalesOrder>();

    using (FeedIterator<SalesOrder> resultSet = container.GetItemQueryIterator<SalesOrder>(
        queryDefinition: null,
        requestOptions: new QueryRequestOptions()
        {
            PartitionKey = new PartitionKey("Account1"),
            MaxItemCount = 5
        }))
    {
        while (resultSet.HasMoreResults)
        {
            FeedResponse<SalesOrder> response = await resultSet.ReadNextAsync();
            SalesOrder salesOrder = response.First();
            Console.WriteLine($"\n1.3.1 Account Number: {salesOrder.AccountNumber}; Id: {salesOrder.Id}");
            allSalesForAccount1.AddRange(response);
        }
    }
}
```

Query items

- .NET SDK v3
- .NET SDK v2

```
private static async Task QueryItems(Container container)
{
    // Query for items by a property other than Id
    QueryDefinition queryDefinition = new QueryDefinition(
        "select * from sales s where s.AccountNumber = @AccountInput")
        .WithParameter("@AccountInput", "Account1");

    List<SalesOrder> allSalesForAccount1 = new List<SalesOrder>();
    using (FeedIterator<SalesOrder> resultSet = container.GetItemQueryIterator<SalesOrder>(
        queryDefinition,
        requestOptions: new QueryRequestOptions()
        {
            PartitionKey = new PartitionKey("Account1"),
            MaxItemCount = 1
        }))
    {
        while (resultSet.HasMoreResults)
        {
            FeedResponse<SalesOrder> response = await resultSet.ReadNextAsync();
            SalesOrder sale = response.First();
            Console.WriteLine($"\n Account Number: {sale.AccountNumber}; Id: {sale.Id};");
            allSalesForAccount1.AddRange(response);
        }
    }
}
```

Delete an item

- .NET SDK v3
- .NET SDK v2

```
private static async Task DeleteItemAsync(Container container)
{
    ItemResponse<SalesOrder> response = await container.DeleteItemAsync<SalesOrder>(
        partitionKey: new PartitionKey("Account1"), id: "SalesOrder3");
}
```

Next steps

- Build a Console app to manage Azure Cosmos DB SQL API data using the v3 SDK
- Learn more about what you can do with the v3 SDK

Migrate your application to use the Azure Cosmos DB .NET SDK v2

12/4/2020 • 2 minutes to read • Edit Online

> **IMPORTANT**
>
> It is important to note that the v3 of the .NET SDK is currently available and a migration plan from v2 to v3 is available here. To learn about the Azure Cosmos DB .NET SDK v2, see the Release notes, the .NET GitHub repository, .NET SDK v2 Performance Tips, and the Troubleshooting guide.

This article highlights some of the considerations to upgrade your existing v1 .NET application to Azure Cosmos DB .NET SDK v2 for Core (SQL) API. Azure Cosmos DB .NET SDK v2 corresponds to the `Microsoft.Azure.DocumentDB` namespace. You can use the information provided in this document if you are migrating your application from any of the following Azure Cosmos DB .NET Platforms to use the v2 SDK `Microsoft.Azure.Cosmos`:

- Azure Cosmos DB .NET Framework v1 SDK for SQL API
- Azure Cosmos DB .NET Core SDK v1 for SQL API

What's available in the .NET v2 SDK

The v2 SDK contains many usability and performance improvements, including:

- Support for TCP direct mode for non-Windows clients
- Multi-region write support
- Improvements on query performance
- Support for geospatial/geometry collections and indexing
- Increased improvements for direct/TCP transport diagnostics
- Updates on direct TCP transport stack to reduce the number of connections established
- Improvements in latency reduction in the RequestTimeout

Most of the retry logic and lower levels of the SDK remains largely unchanged.

Why migrate to the .NET v2 SDK

In addition to the numerous performance improvements, new feature investments made in the latest SDK will not be back ported to older versions.

Additionally, the older SDKs will be replaced by newer versions and the v1 SDK will go into maintenance mode. For the best development experience, we recommend migrating your application to a later version of the SDK.

Major changes from v1 SDK to v2 SDK

Direct mode + TCP

The .NET v2 SDK now supports both direct and gateway mode. Direct mode supports connectivity through TCP protocol and offers better performance as it connects directly to the backend replicas with fewer network hops.

For more details, read through the Azure Cosmos DB SQL SDK connectivity modes guide.

Session token formatting

The v2 SDK no longer uses the *simple* session token format as used in v1, instead the SDK uses *vector* formatting. The format should be converted when passing to the client application with different versions, since the formats are not interchangeable.

For more information, see converting session token formats in the .NET SDK.

Using the .NET change feed processor SDK

The .NET change feed processor library 2.1.x requires `Microsoft.Azure.DocumentDB` 2.0 or later.

The 2.1.x library has the following key changes:

- Stability and diagnosability improvements
- Improved handling of errors and exceptions
- Additional support for partitioned lease collections
- Advanced extensions to implement the `ChangeFeedDocument` interface and class for additional error handling and tracing
- Added support for using a custom store to persist continuation tokens per partition

For more information, see the change feed processor library release notes.

Using the bulk executor library

The v2 bulk executor library currently has a dependency on the Azure Cosmos DB .NET SDK 2.5.1 or later.

For more information, see the Azure Cosmos DB bulk executor library overview and the .NET bulk executor library release notes.

Next steps

- Read about additional performance tips using Azure Cosmos DB for SQL API v2 for optimization your application to achieve max performance
- Learn more about what you can do with the v2 SDK

Troubleshoot query issues when using Azure Cosmos DB

12/4/2020 • 14 minutes to read • Edit Online

APPLIES TO: SQL API

This article walks through a general recommended approach for troubleshooting queries in Azure Cosmos DB. Although you shouldn't consider the steps outlined in this article a complete defense against potential query issues, we've included the most common performance tips here. You should use this article as a starting place for troubleshooting slow or expensive queries in the Azure Cosmos DB core (SQL) API. You can also use diagnostics logs to identify queries that are slow or that consume significant amounts of throughput. If you are using Azure Cosmos DB's API for MongoDB, you should use Azure Cosmos DB's API for MongoDB query troubleshooting guide

Query optimizations in Azure Cosmos DB are broadly categorized as follows:

- Optimizations that reduce the Request Unit (RU) charge of the query
- Optimizations that just reduce latency

If you reduce the RU charge of a query, you'll typically decrease latency as well.

This article provides examples that you can re-create by using the nutrition dataset.

Common SDK issues

Before reading this guide, it is helpful to consider common SDK issues that aren't related to the query engine.

- Follow these SDK Performance tips.
  - .NET SDK troubleshooting guide
  - Java SDK troubleshooting guide
- The SDK allows setting a `MaxItemCount` for your queries but you can't specify a minimum item count.
  - Code should handle any page size, from zero to the `MaxItemCount`.
- Sometimes queries may have empty pages even when there are results on a future page. Reasons for this could be:
  - The SDK could be doing multiple network calls.
  - The query might be taking a long time to retrieve the documents.
- All queries have a continuation token that will allow the query to continue. Be sure to drain the query completely. Learn more about handling multiple pages of results

Get query metrics

When you optimize a query in Azure Cosmos DB, the first step is always to get the query metrics for your query. These metrics are also available through the Azure portal. Once you run your query in the Data Explorer, the query metrics are visible next to the **Results** tab:

After you get the query metrics, compare the **Retrieved Document Count** with the **Output Document Count** for your query. Use this comparison to identify the relevant sections to review in this article.

The **Retrieved Document Count** is the number of documents that the query engine needed to load. The **Output Document Count** is the number of documents that were needed for the results of the query. If the **Retrieved Document Count** is significantly higher than the **Output Document Count**, there was at least one part of your query that was unable to use an index and needed to do a scan.

Refer to the following sections to understand the relevant query optimizations for your scenario.

Query's RU charge is too high

Retrieved Document Count is significantly higher than Output Document Count

- Include necessary paths in the indexing policy.
- Understand which system functions use the index.
- Understand which aggregate queries use the index.
- Optimize queries that have both a filter and an ORDER BY clause.
- Optimize JOIN expressions by using a subquery.

Retrieved Document Count is approximately equal to Output Document Count

- Minimize cross partition queries.
- Optimize queries that have filters on multiple properties.
- Optimize queries that have both a filter and an ORDER BY clause.

Query's RU charge is acceptable but latency is still too high

- Improve proximity.
- Increase provisioned throughput.
- Increase MaxConcurrency.
- Increase MaxBufferedItemCount.

Queries where Retrieved Document Count exceeds Output Document

Count

The **Retrieved Document Count** is the number of documents that the query engine needed to load. The **Output Document Count** is the number of documents returned by the query. If the **Retrieved Document Count** is significantly higher than the **Output Document Count**, there was at least one part of your query that was unable to use an index and needed to do a scan.

Here's an example of scan query that wasn't entirely served by the index:

Query:

```
SELECT VALUE c.description
FROM c
WHERE UPPER(c.description) = "BABYFOOD, DESSERT, FRUIT DESSERT, WITHOUT ASCORBIC ACID, JUNIOR"
```

Query metrics:

```
Retrieved Document Count                  :          60,951
Retrieved Document Size                   :     399,998,938 bytes
Output Document Count                     :               7
Output Document Size                      :             510 bytes
Index Utilization                         :            0.00 %
Total Query Execution Time                :        4,500.34 milliseconds
  Query Preparation Times
    Query Compilation Time                :            0.09 milliseconds
    Logical Plan Build Time               :            0.05 milliseconds
    Physical Plan Build Time              :            0.04 milliseconds
    Query Optimization Time               :            0.01 milliseconds
  Index Lookup Time                       :            0.01 milliseconds
  Document Load Time                      :        4,177.66 milliseconds
  Runtime Execution Times
    Query Engine Times                    :          322.16 milliseconds
    System Function Execution Time        :           85.74 milliseconds
    User-defined Function Execution Time  :            0.00 milliseconds
  Document Write Time                     :            0.01 milliseconds
Client Side Metrics
  Retry Count                             :               0
  Request Charge                          :        4,059.95 RUs
```

The **Retrieved Document Count** (60,951) is significantly higher than the **Output Document Count** (7), implying that this query resulted in a document scan. In this case, the system function UPPER() doesn't use an index.

Include necessary paths in the indexing policy

Your indexing policy should cover any properties included in `WHERE` clauses, `ORDER BY` clauses, `JOIN`, and most system functions. The desired paths specified in the index policy should match the properties in the JSON documents.

> **NOTE**
>
> Properties in Azure Cosmos DB indexing policy are case-sensitive

If you run the following simple query on the nutrition dataset, you will observe a much lower RU charge when the property in the `WHERE` clause is indexed:

Original

Query:

```
SELECT *
FROM c
WHERE c.description = "Malabar spinach, cooked"
```

Indexing policy:

```
{
    "indexingMode": "consistent",
    "automatic": true,
    "includedPaths": [
        {
            "path": "/*"
        }
    ],
    "excludedPaths": [
        {
            "path": "/description/*"
        }
    ]
}
```

RU charge: 409.51 RUs

Optimized

Updated indexing policy:

```
{
    "indexingMode": "consistent",
    "automatic": true,
    "includedPaths": [
        {
            "path": "/*"
        }
    ],
    "excludedPaths": []
}
```

RU charge: 2.98 RUs

You can add properties to the indexing policy at any time, with no effect on write or read availability. You can track index transformation progress.

Understand which system functions use the index

If an expression can be translated into a range of string values, it can use the index. Otherwise, it can't.

Here's the list of some common string functions that can use the index:

- STARTSWITH(str_expr1, str_expr2, bool_expr)
- CONTAINS(str_expr, str_expr, bool_expr)
- LEFT(str_expr, num_expr) = str_expr
- SUBSTRING(str_expr, num_expr, num_expr) = str_expr, but only if the first num_expr is 0

Following are some common system functions that don't use the index and must load each document:

SYSTEM FUNCTION	IDEAS FOR OPTIMIZATION

SYSTEM FUNCTION	IDEAS FOR OPTIMIZATION
UPPER/LOWER	Instead of using the system function to normalize data for comparisons, normalize the casing upon insertion. A query like `SELECT * FROM c WHERE UPPER(c.name) = 'BOB'` becomes `SELECT * FROM c WHERE c.name = 'BOB'`.
Mathematical functions (non-aggregates)	If you need to compute a value frequently in your query, consider storing the value as a property in your JSON document.

Other parts of the query might still use the index even though the system functions don't.

Understand which aggregate queries use the index

In most cases, aggregate system functions in Azure Cosmos DB will use the index. However, depending on the filters or additional clauses in an aggregate query, the query engine may be required to load a high number of documents. Typically, the query engine will apply equality and range filters first. After applying these filters, the query engine can evaluate additional filters and resort to loading remaining documents to compute the aggregate, if needed.

For example, given these two sample queries, the query with both an equality and `CONTAINS` system function filter will generally be more efficient than a query with just a `CONTAINS` system function filter. This is because the equality filter is applied first and uses the index before documents need to be loaded for the more expensive `CONTAINS` filter.

Query with only `CONTAINS` filter - higher RU charge:

```
SELECT COUNT(1)
FROM c
WHERE CONTAINS(c.description, "spinach")
```

Query with both equality filter and `CONTAINS` filter - lower RU charge:

```
SELECT AVG(c._ts)
FROM c
WHERE c.foodGroup = "Sausages and Luncheon Meats" AND CONTAINS(c.description, "spinach")
```

Here are additional examples of aggregate queries that will not fully use the index:

Queries with system functions that don't use the index

You should refer to the relevant system function's page to see if it uses the index.

```
SELECT MAX(c._ts)
FROM c
WHERE CONTAINS(c.description, "spinach")
```

Aggregate queries with user-defined functions(UDF's)

```
SELECT AVG(c._ts)
FROM c
WHERE udf.MyUDF("Sausages and Luncheon Meats")
```

Queries with GROUP BY

The RU charge of queries with `GROUP BY` will increase as the cardinality of the properties in the `GROUP BY` clause increases. In the below query, for example, the RU charge of the query will increase as the number unique

descriptions increases.

The RU charge of an aggregate function with a `GROUP BY` clause will be higher than the RU charge of an aggregate function alone. In this example, the query engine must load every document that matches the `c.foodGroup = "Sausages and Luncheon Meats"` filter so the RU charge is expected to be high.

```
SELECT COUNT(1)
FROM c
WHERE c.foodGroup = "Sausages and Luncheon Meats"
GROUP BY c.description
```

If you plan to frequently run the same aggregate queries, it may be more efficient to build a real-time materialized view with the Azure Cosmos DB change feed than running individual queries.

Optimize queries that have both a filter and an ORDER BY clause

Although queries that have a filter and an `ORDER BY` clause will normally use a range index, they'll be more efficient if they can be served from a composite index. In addition to modifying the indexing policy, you should add all properties in the composite index to the `ORDER BY` clause. This change to the query will ensure that it uses the composite index. You can observe the impact by running a query on the nutrition dataset:

Original

Query:

```
SELECT *
FROM c
WHERE c.foodGroup = "Soups, Sauces, and Gravies"
ORDER BY c._ts ASC
```

Indexing policy:

```
{

        "automatic":true,
        "indexingMode":"Consistent",
        "includedPaths":[
            {
                "path":"/*"
            }
        ],
        "excludedPaths":[]
}
```

RU charge: 44.28 RUs

Optimized

Updated query (includes both properties in the `ORDER BY` clause):

```
SELECT *
FROM c
WHERE c.foodGroup = "Soups, Sauces, and Gravies"
ORDER BY c.foodGroup, c._ts ASC
```

Updated indexing policy:

```
{
        "automatic":true,
        "indexingMode":"Consistent",
        "includedPaths":[
            {
                "path":"/*"
            }
        ],
        "excludedPaths":[],
        "compositeIndexes":[
            [
                {
                    "path":"/foodGroup",
                    "order":"ascending"
        },
                {
                    "path":"/_ts",
                    "order":"ascending"
                }
            ]
        ]
    }
```

RU charge: 8.86 RUs

Optimize JOIN expressions by using a subquery

Multi-value subqueries can optimize `JOIN` expressions by pushing predicates after each select-many expression rather than after all cross joins in the `WHERE` clause.

Consider this query:

```
SELECT Count(1) AS Count
FROM c
JOIN t IN c.tags
JOIN n IN c.nutrients
JOIN s IN c.servings
WHERE t.name = 'infant formula' AND (n.nutritionValue > 0
AND n.nutritionValue < 10) AND s.amount > 1
```

RU charge: 167.62 RUs

For this query, the index will match any document that has a tag with the name `infant formula`, `nutritionValue` greater than 0, and `amount` greater than 1. The `JOIN` expression here will perform the cross-product of all items of tags, nutrients, and servings arrays for each matching document before any filter is applied. The `WHERE` clause will then apply the filter predicate on each `<c, t, n, s>` tuple.

For example, if a matching document has 10 items in each of the three arrays, it will expand to 1 x 10 x 10 x 10 (that is, 1,000) tuples. The use of subqueries here can help to filter out joined array items before joining with the next expression.

This query is equivalent to the preceding one but uses subqueries:

```
SELECT Count(1) AS Count
FROM c
JOIN (SELECT VALUE t FROM t IN c.tags WHERE t.name = 'infant formula')
JOIN (SELECT VALUE n FROM n IN c.nutrients WHERE n.nutritionValue > 0 AND n.nutritionValue < 10)
JOIN (SELECT VALUE s FROM s IN c.servings WHERE s.amount > 1)
```

RU charge: 22.17 RUs

Assume that only one item in the tags array matches the filter and that there are five items for both the nutrients and servings arrays. The `JOIN` expressions will expand to 1 x 1 x 5 x 5 = 25 items, as opposed to 1,000 items in the first query.

Queries where Retrieved Document Count is equal to Output Document Count

If the **Retrieved Document Count** is approximately equal to the **Output Document Count**, the query engine didn't have to scan many unnecessary documents. For many queries, like those that use the `TOP` keyword, **Retrieved Document Count** might exceed **Output Document Count** by 1. You don't need to be concerned about this.

Minimize cross partition queries

Azure Cosmos DB uses partitioning to scale individual containers as Request Unit and data storage needs increase. Each physical partition has a separate and independent index. If your query has an equality filter that matches your container's partition key, you'll need to check only the relevant partition's index. This optimization reduces the total number of RUs that the query requires.

If you have a large number of provisioned RUs (more than 30,000) or a large amount of data stored (more than approximately 100 GB), you probably have a large enough container to see a significant reduction in query RU charges.

For example, if you create a container with the partition key foodGroup, the following queries will need to check only a single physical partition:

```
SELECT *
FROM c
WHERE c.foodGroup = "Soups, Sauces, and Gravies" and c.description = "Mushroom, oyster, raw"
```

Queries that have an `IN` filter with the partition key will only check the relevant physical partition(s) and will not "fan-out":

```
SELECT *
FROM c
WHERE c.foodGroup IN("Soups, Sauces, and Gravies", "Vegetables and Vegetable Products") and c.description =
"Mushroom, oyster, raw"
```

Queries that have range filters on the partition key, or that don't have any filters on the partition key, will need to "fan-out" and check every physical partition's index for results:

```
SELECT *
FROM c
WHERE c.description = "Mushroom, oyster, raw"
```

```
SELECT *
FROM c
WHERE c.foodGroup > "Soups, Sauces, and Gravies" and c.description = "Mushroom, oyster, raw"
```

Optimize queries that have filters on multiple properties

Although queries that have filters on multiple properties will normally use a range index, they'll be more efficient if they can be served from a composite index. For small amounts of data, this optimization won't have a significant impact. It could be useful, however, for large amounts of data. You can only optimize, at most, one non-equality filter per composite index. If your query has multiple non-equality filters, pick one of them that will use the

composite index. The rest will continue to use range indexes. The non-equality filter must be defined last in the composite index. Learn more about composite indexes.

Here are some examples of queries that could be optimized with a composite index:

```
SELECT *
FROM c
WHERE c.foodGroup = "Vegetables and Vegetable Products" AND c._ts = 1575503264
```

```
SELECT *
FROM c
WHERE c.foodGroup = "Vegetables and Vegetable Products" AND c._ts > 1575503264
```

Here's the relevant composite index:

```
{
        "automatic":true,
        "indexingMode":"Consistent",
        "includedPaths":[
            {
                "path":"/*"
            }
        ],
        "excludedPaths":[],
        "compositeIndexes":[
            [
                {
                    "path":"/foodGroup",
                    "order":"ascending"
                },
                {
                    "path":"/_ts",
                    "order":"ascending"
                }
            ]
        ]
}
```

Optimizations that reduce query latency

In many cases, the RU charge might be acceptable when query latency is still too high. The following sections give an overview of tips for reducing query latency. If you run the same query multiple times on the same dataset, it will typically have the same RU charge each time. But query latency might vary between query executions.

Improve proximity

Queries that are run from a different region than the Azure Cosmos DB account will have higher latency than if they were run inside the same region. For example, if you're running code on your desktop computer, you should expect latency to be tens or hundreds of milliseconds higher (or more) than if the query came from a virtual machine within the same Azure region as Azure Cosmos DB. It's simple to globally distribute data in Azure Cosmos DB to ensure you can bring your data closer to your app.

Increase provisioned throughput

In Azure Cosmos DB, your provisioned throughput is measured in Request Units (RUs). Imagine you have a query that consumes 5 RUs of throughput. For example, if you provision 1,000 RUs, you would be able to run that query 200 times per second. If you tried to run the query when there wasn't enough throughput available, Azure Cosmos DB would return an HTTP 429 error. Any of the current Core (SQL) API SDKs will automatically retry this query after waiting for a short time. Throttled requests take longer, so increasing provisioned throughput can improve

query latency. You can observe the total number of throttled requests on the **Metrics** blade of the Azure portal.

Increase MaxConcurrency

Parallel queries work by querying multiple partitions in parallel. But data from an individual partitioned collection is fetched serially with respect to the query. So, if you set MaxConcurrency to the number of partitions, you have the best chance of achieving the most performant query, provided all other system conditions remain the same. If you don't know the number of partitions, you can set MaxConcurrency (or MaxDegreesOfParallelism in older SDK versions) to a high number. The system will choose the minimum (number of partitions, user provided input) as the maximum degree of parallelism.

Increase MaxBufferedItemCount

Queries are designed to pre-fetch results while the current batch of results is being processed by the client. Pre-fetching helps to improve the overall latency of a query. Setting MaxBufferedItemCount limits the number of pre-fetched results. If you set this value to the expected number of results returned (or a higher number), the query can get the most benefit from pre-fetching. If you set this value to -1, the system will automatically determine the number of items to buffer.

Next steps

See the following articles for information on how to measure RUs per query, get execution statistics to tune your queries, and more:

- Get SQL query execution metrics by using .NET SDK
- Tuning query performance with Azure Cosmos DB
- Performance tips for .NET SDK
- Performance tips for Java v4 SDK

Diagnose and troubleshoot Azure Cosmos DB request timeout exceptions

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: SQL API

Azure Cosmos DB returned an HTTP 408 request timeout.

Troubleshooting steps

The following list contains known causes and solutions for request timeout exceptions.

Check the SLA

Check Azure Cosmos DB monitoring to see if the number of 408 exceptions violates the Azure Cosmos DB SLA.

Solution 1: It didn't violate the Azure Cosmos DB SLA

The application should handle this scenario and retry on these transient failures.

Solution 2: It did violate the Azure Cosmos DB SLA

Contact Azure Support.

Hot partition key

Azure Cosmos DB distributes the overall provisioned throughput evenly across physical partitions. When there's a hot partition, one or more logical partition keys on a physical partition are consuming all the physical partition's Request Units per second (RU/s). At the same time, the RU/s on other physical partitions are going unused. As a symptom, the total RU/s consumed will be less than the overall provisioned RU/s at the database or container. You'll still see throttling (429s) on the requests against the hot logical partition key. Use the Normalized RU Consumption metric to see if the workload is encountering a hot partition.

Solution:

Choose a good partition key that evenly distributes request volume and storage. Learn how to change your partition key.

Next steps

- Diagnose and troubleshoot issues when you use the Azure Cosmos DB .NET SDK.
- Learn about performance guidelines for .NET v3 and .NET v2.
- Diagnose and troubleshoot issues when you use the Azure Cosmos DB Java v4 SDK.
- Learn about performance guidelines for Java v4 SDK.

Diagnose and troubleshoot Azure Cosmos DB request rate too large exceptions

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: SQL API

A "Request rate too large" message or error code 429 indicates that your requests are being throttled.

Troubleshooting steps

The following section contains known causes and solutions for too many requests.

Check the metrics

Check Azure Cosmos DB monitoring to see the number of 429 exceptions.

Cause:

The consumed throughput (Request Units per second) has exceeded the provisioned throughput. The SDK automatically retries requests based on the specified retry policy. If you get this failure often, consider increasing the throughput on the collection. Check the portal's metrics to see if you're getting 429 errors. Review your partition key to ensure it results in an even distribution of storage and request volume.

Solution:

1. Use the portal or the SDK to increase the provisioned throughput.
2. Switch the database or container to Autoscale.

Next steps

- Diagnose and troubleshoot issues when you use the Azure Cosmos DB .NET SDK.
- Learn about performance guidelines for .NET v3 and .NET v2.
- Diagnose and troubleshoot issues when you use the Azure Cosmos DB Java v4 SDK.
- Learn about performance guidelines for Java v4 SDK.

Diagnose and troubleshoot Azure Cosmos DB not found exceptions

12/4/2020 • 3 minutes to read • Edit Online

APPLIES TO: SQL API

The HTTP status code 404 represents that the resource no longer exists.

Expected behavior

There are many valid scenarios where an application expects a code 404 and correctly handles the scenario.

A not found exception was returned for an item that should exist or does exist

Here are the possible reasons for a status code 404 to be returned if the item should exist or does exist.

The read session is not available for the input session token

Solution:

1. Update your current SDK to the latest version available. The most common causes for this particular error have been fixed in the newest SDK versions.

Race condition

There are multiple SDK client instances and the read happened before the write.

Solution:

1. The default account consistency for Azure Cosmos DB is session consistency. When an item is created or updated, the response returns a session token that can be passed between SDK instances to guarantee that the read request is reading from a replica with that change.
2. Change the consistency level to a stronger level.

Invalid partition key and ID combination

The partition key and ID combination aren't valid.

Solution:

Fix the application logic that's causing the incorrect combination.

Invalid character in an item ID

An item is inserted into Azure Cosmos DB with an invalid character in the item ID.

Solution:

Change the ID to a different value that doesn't contain the special characters. If changing the ID isn't an option, you can Base64 encode the ID to escape the special characters. Base64 can still produce a name with a invalid character '/' which needs to be replaced.

Items already inserted in the container for the ID can be replaced by using RID values instead of name-based references.

```
// Get a container reference that uses RID values.
ContainerProperties containerProperties = await this.Container.ReadContainerAsync();
string[] selfLinkSegments = containerProperties.SelfLink.Split('/');
string databaseRid = selfLinkSegments[1];
string containerRid = selfLinkSegments[3];
Container containerByRid = this.cosmosClient.GetContainer(databaseRid, containerRid);

// Invalid characters are listed here.
//https://docs.microsoft.com/dotnet/api/microsoft.azure.documents.resource.id?view=azure-dotnet&preserve-
view=true#remarks
FeedIterator<JObject> invalidItemsIterator = this.Container.GetItemQueryIterator<JObject>(
    @"select * from t where CONTAINS(t.id, ""/"") or CONTAINS(t.id, ""#"") or CONTAINS(t.id, ""?"") or
CONTAINS(t.id, ""\\"") ");
while (invalidItemsIterator.HasMoreResults)
{
    foreach (JObject itemWithInvalidId in await invalidItemsIterator.ReadNextAsync())
    {
        // Choose a new ID that doesn't contain special characters.
        // If that isn't possible, then Base64 encode the ID to escape the special characters.
        byte[] plainTextBytes = Encoding.UTF8.GetBytes(itemWithInvalidId["id"].ToString());
        itemWithInvalidId["id"] = Convert.ToBase64String(plainTextBytes).Replace('/', '!');

        // Update the item with the new ID value by using the RID-based container reference.
        JObject item = await containerByRid.ReplaceItemAsync<JObject>(
            item: itemWithInvalidId,
            ID: itemWithInvalidId["_rid"].ToString(),
            partitionKey: new Cosmos.PartitionKey(itemWithInvalidId["status"].ToString()));

        // Validating the new ID can be read by using the original name-based container reference.
        await this.Container.ReadItemAsync<ToDoActivity>(
            item["id"].ToString(),
            new Cosmos.PartitionKey(item["status"].ToString())); ;
    }
}
```

Time to Live purge

The item had the Time to Live (TTL) property set. The item was purged because the TTL property expired.

Solution:

Change the TTL property to prevent the item from being purged.

Lazy indexing

The lazy indexing hasn't caught up.

Solution:

Wait for the indexing to catch up or change the indexing policy.

Parent resource deleted

The database or container that the item exists in was deleted.

Solution:

1. Restore the parent resource, or re-create the resources.
2. Create a new resource to replace the deleted resource.

7. Container/Collection names are case-sensitive

Container/Collection names are case-sensitive in Cosmos DB.

Solution:

Make sure to use the exact name while connecting to Cosmos DB.

Next steps

- Diagnose and troubleshoot issues when you use the Azure Cosmos DB .NET SDK.
- Learn about performance guidelines for .NET v3 and .NET v2.
- Diagnose and troubleshoot issues when you use the Azure Cosmos DB Java v4 SDK.
- Learn about performance guidelines for Java v4 SDK.

Diagnose and troubleshoot Azure Cosmos DB unauthorized exceptions

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: SQL API

HTTP 401: The MAC signature found in the HTTP request isn't the same as the computed signature. If you received the 401 error message "The MAC signature found in the HTTP request is not the same as the computed signature," it can be caused by the following scenarios.

For older SDKs, the exception can appear as an invalid JSON exception instead of the correct 401 unauthorized exception. Newer SDKs properly handle this scenario and give a valid error message.

Troubleshooting steps

The following list contains known causes and solutions for unauthorized exceptions.

The key wasn't properly rotated is the most common scenario

The 401 MAC signature is seen shortly after a key rotation and eventually stops without any changes.

Solution:

The key was rotated and didn't follow the best practices. The Azure Cosmos DB account key rotation can take anywhere from a few seconds to possibly days depending on the Azure Cosmos DB account size.

The key is misconfigured

The 401 MAC signature issue will be consistent and happens for all calls using that key.

Solution:

The key is misconfigured on the application and is using the wrong key for the account, or the entire key wasn't copied.

The application is using the read-only keys for write operations

The 401 MAC signature issue only occurs for write operations like create or replace, but read requests succeed.

Solution:

Switch the application to use a read/write key to allow the operations to complete successfully.

Race condition with create container

The 401 MAC signature issue is seen shortly after a container creation. This issue occurs only until the container creation is completed.

Solution:

There's a race condition with container creation. An application instance is trying to access the container before the container creation is complete. The most common scenario for this race condition is if the application is running and the container is deleted and re-created with the same name. The SDK attempts to use the new container, but the container creation is still in progress so it doesn't have the keys.

Next steps

- Diagnose and troubleshoot issues when you use the Azure Cosmos DB .NET SDK.
- Learn about performance guidelines for .NET v3 and .NET v2.
- Diagnose and troubleshoot issues when you use the Azure Cosmos DB Java v4 SDK.

- Learn about performance guidelines for Java v4 SDK.

Diagnose and troubleshoot the availability of Azure Cosmos SDKs in multiregional environments

11/2/2020 • 4 minutes to read • Edit Online

APPLIES TO: SQL API

This article describes the behavior of the latest version of Azure Cosmos SDKs when you see a connectivity issue to a particular region or when a region failover occurs.

All the Azure Cosmos SDKs give you an option to customize the regional preference. The following properties are used in different SDKs:

- The ConnectionPolicy.PreferredLocations property in .NET V2 SDK.
- The CosmosClientOptions.ApplicationRegion or CosmosClientOptions.ApplicationPreferredRegions properties in .NET V3 SDK.
- The CosmosClientBuilder.preferredRegions method in Java V4 SDK.
- The CosmosClient.preferred_locations parameter in Python SDK.
- The CosmosClientOptions.ConnectionPolicy.preferredLocations parameter in JS SDK.

When you set the regional preference, the client will connect to a region as mentioned in the following table:

ACCOUNT TYPE	READS	WRITES
Single write region	Preferred region	Primary region
Multiple write regions	Preferred region	Preferred region

If you **don't set a preferred region**, the SDK client defaults to the primary region:

ACCOUNT TYPE	READS	WRITES
Single write region	Primary region	Primary region
Multiple write regions	Primary region	Primary region

> **NOTE**
> Primary region refers to the first region in the Azure Cosmos account region list

Under normal circumstances, the SDK client will connect to the preferred region (if a regional preference is set) or to the primary region (if no preference is set), and the operations will be limited to that region, unless any of the below scenarios occur.

In these cases, the client using the Azure Cosmos SDK exposes logs and includes the retry information as part of the **operation diagnostic information**:

- The *RequestDiagnosticsString* property on responses in .NET V2 SDK.
- The *Diagnostics* property on responses and exceptions in .NET V3 SDK.
- The *getDiagnostics()* method on responses and exceptions in Java V4 SDK.

When determining the next region in order of preference, the SDK client will use the account region list, prioritizing the preferred regions (if any).

For a comprehensive detail on SLA guarantees during these events, see the SLAs for availability.

Removing a region from the account

When you remove a region from an Azure Cosmos account, any SDK client that actively uses the account will detect the region removal through a backend response code. The client then marks the regional endpoint as unavailable. The client retries the current operation and all the future operations are permanently routed to the next region in order of preference.

Adding a region to an account

Every 5 minutes, the Azure Cosmos SDK client reads the account configuration and refreshes the regions that it's aware of.

If you remove a region and later add it back to the account, if the added region has a higher regional preference order in the SDK configuration than the current connected region, the SDK will switch back to use this region permanently. After the added region is detected, all the future requests are directed to it.

If you configure the client to preferably connect to a region that the Azure Cosmos account does not have, the preferred region is ignored. If you add that region later, the client detects it and will switch permanently to that region.

Fail over the write region in a single write region account

If you initiate a failover of the current write region, the next write request will fail with a known backend response. When this response is detected, the client will query the account to learn the new write region and proceeds to retry the current operation and permanently route all future write operations to the new region.

Regional outage

If the account is single write region and the regional outage occurs during a write operation, the behavior is similar to a manual failover. For read requests or multiple write regions accounts, the behavior is similar to removing a region.

Session consistency guarantees

When using session consistency, the client needs to guarantee that it can read its own writes. In single write region accounts where the read region preference is different from the write region, there could be cases where the user issues a write and when doing a read from a local region, the local region has not yet received the data replication (speed of light constraint). In such cases, the SDK detects the specific failure on the read operation and retries the read on the primary region to ensure session consistency.

Transient connectivity issues on TCP protocol

In scenarios where the Azure Cosmos SDK client is configured to use the TCP protocol, for a given request, there might be situations where the network conditions are temporarily affecting the communication with a particular endpoint. These temporary network conditions can surface as TCP timeouts. The client will retry the request locally on the same endpoint for some seconds.

If the user has configured a preferred region list with more than one region and the Azure Cosmos account is multiple write regions or single write region and the operation is a read request, the client will retry that single operation in the next region from the preference list.

Next steps

- Review the Availability SLAs.
- Use the latest .NET SDK
- Use the latest Java SDK
- Use the latest Python SDK
- Use the latest Node SDK

Diagnose and troubleshoot issues when using Azure Cosmos DB .NET SDK

12/4/2020 • 5 minutes to read • Edit Online

APPLIES TO: SQL API

This article covers common issues, workarounds, diagnostic steps, and tools when you use the .NET SDK with Azure Cosmos DB SQL API accounts. The .NET SDK provides client-side logical representation to access the Azure Cosmos DB SQL API. This article describes tools and approaches to help you if you run into any issues.

Checklist for troubleshooting issues

Consider the following checklist before you move your application to production. Using the checklist will prevent several common issues you might see. You can also quickly diagnose when an issue occurs:

- Use the latest SDK. Preview SDKs should not be used for production. This will prevent hitting known issues that are already fixed.
- Review the performance tips, and follow the suggested practices. This will help prevent scaling, latency, and other performance issues.
- Enable the SDK logging to help you troubleshoot an issue. Enabling the logging may affect performance so it's best to enable it only when troubleshooting issues. You can enable the following logs:
- Log metrics by using the Azure portal. Portal metrics show the Azure Cosmos DB telemetry, which is helpful to determine if the issue corresponds to Azure Cosmos DB or if it's from the client side.
- Log the diagnostics string in the V2 SDK or diagnostics in V3 SDK from the point operation responses.
- Log the SQL Query Metrics from all the query responses
- Follow the setup for SDK logging

Take a look at the Common issues and workarounds section in this article.

Check the GitHub issues section that's actively monitored. Check to see if any similar issue with a workaround is already filed. If you didn't find a solution, then file a GitHub issue. You can open a support tick for urgent issues.

Common issues and workarounds

General suggestions

- Run your app in the same Azure region as your Azure Cosmos DB account, whenever possible.
- You may run into connectivity/availability issues due to lack of resources on your client machine. We recommend monitoring your CPU utilization on nodes running the Azure Cosmos DB client, and scaling up/out if they're running at high load.

Check the portal metrics

Checking the portal metrics will help determine if it's a client-side issue or if there is an issue with the service. For example, if the metrics contain a high rate of rate-limited requests (HTTP status code 429) which means the request is getting throttled then check the Request rate too large section.

Common error status codes

STATUS CODE	DESCRIPTION
400	Bad request (Depends on the error message)
401	Not authorized
404	Resource is not found
408	Request timed out
409	Conflict failure is when the ID provided for a resource on a write operation has been taken by an existing resource. Use another ID for the resource to resolve this issue as ID must be unique within all documents with the same partition key value.
410	Gone exceptions (Transient failure that should not violate SLA)
412	Precondition failure is where the operation specified an eTag that is different from the version available at the server. It's optimistic concurrency error. Retry the request after reading the latest version of the resource and updating the eTag on the request.
413	Request Entity Too Large
429	Too many requests
449	Transient error that only occurs on write operations, and is safe to retry
500	The operation failed due to an unexpected service error. Contact support. See Filing an Azure support issue.
503	Service unavailable

Azure SNAT (PAT) port exhaustion

If your app is deployed on Azure Virtual Machines without a public IP address, by default Azure SNAT ports establish connections to any endpoint outside of your VM. The number of connections allowed from the VM to the Azure Cosmos DB endpoint is limited by the Azure SNAT configuration. This situation can lead to connection throttling, connection closure, or the above mentioned Request timeouts.

Azure SNAT ports are used only when your VM has a private IP address is connecting to a public IP address. There are two workarounds to avoid Azure SNAT limitation (provided you already are using a single client instance across the entire application):

- Add your Azure Cosmos DB service endpoint to the subnet of your Azure Virtual Machines virtual network. For more information, see Azure Virtual Network service endpoints.

  When the service endpoint is enabled, the requests are no longer sent from a public IP to Azure Cosmos DB. Instead, the virtual network and subnet identity are sent. This change might result in firewall drops if only public IPs are allowed. If you use a firewall, when you enable the service endpoint, add a subnet to the firewall by using Virtual Network ACLs.

- Assign a public IP to your Azure VM.

High network latency

High network latency can be identified by using the diagnostics string in the V2 SDK or diagnostics in V3 SDK.

If no timeouts are present and the diagnostics show single requests where the high latency is evident on the difference between `ResponseTime` and `RequestStartTime`, like so (>300 milliseconds in this example):

```
RequestStartTime: 2020-03-09T22:44:49.5373624Z, RequestEndTime: 2020-03-09T22:44:49.9279906Z,  Number of
regions attempted:1
ResponseTime: 2020-03-09T22:44:49.9279906Z, StoreResult: StorePhysicalAddress: rntbd://..., ...
```

This latency can have multiple causes:

- Your application is not running in the same region as your Azure Cosmos DB account.
- Your PreferredLocations or ApplicationRegion configuration is incorrect and is trying to connect to a different region to where your application is currently running on.
- There might be a bottleneck on the Network interface because of high traffic. If the application is running on Azure Virtual Machines, there are possible workarounds:
  - Consider using a Virtual Machine with Accelerated Networking enabled.
  - Enable Accelerated Networking on an existing Virtual Machine.
  - Consider using a higher end Virtual Machine.

Common query issues

The query metrics will help determine where the query is spending most of the time. From the query metrics, you can see how much of it is being spent on the back-end vs the client. Learn more about troubleshooting query performance.

- If the back-end query returns quickly, and spends a large time on the client check the load on the machine. It's likely that there are not enough resource and the SDK is waiting for resources to be available to handle the response.

- If the back-end query is slow, try optimizing the query and looking at the current indexing policy

> **NOTE**
>
> For improved performance, we recommend Windows 64-bit host processing. The SQL SDK includes a native ServiceInterop.dll to parse and optimize queries locally. ServiceInterop.dll is supported only on the Windows x64 platform. For Linux and other unsupported platforms where ServiceInterop.dll isn't available, an additional network call will be made to the gateway to get the optimized query.

If you encounter the following error:
`Unable to load DLL 'Microsoft.Azure.Cosmos.ServiceInterop.dll' or one of its dependencies:` and are using Windows, you should upgrade to the latest Windows version.

Next steps

- Learn about Performance guidelines for .NET V3 and .NET V2
- Learn about the Reactor-based Java SDKs

Diagnose and troubleshoot Azure Cosmos DB "Request header too large" message

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: SQL API

The "Request header too large" message is thrown with an HTTP error code 400. This error occurs if the size of the request header has grown so large that it exceeds the maximum-allowed size. We recommend that you use the latest version of the SDK. Use at least version 3.x or 2.x, because these versions add header size tracing to the exception message.

Troubleshooting steps

The "Request header too large" message occurs if the session or the continuation token is too large. The following sections describe the cause of the issue and its solution in each category.

Session token is too large

Cause:

A 400 bad request most likely occurs because the session token is too large. If the following statements are true, the session token is too large:

- The error occurs on point operations like create, read, and update where there isn't a continuation token.
- The exception started without making any changes to the application. The session token grows as the number of partitions increases in the container. The number of partitions increases as the amount of data increases or if the throughput is increased.

Temporary mitigation:

Restart your client application to reset all the session tokens. Eventually, the session token will grow back to the previous size that caused the issue. To avoid this issue completely, use the solution in the next section.

Solution:

1. Follow the guidance in the .NET v3 or .NET v2 performance tips articles. Convert the application to use the direct connection mode with the Transmission Control Protocol (TCP). The direct connection mode with the TCP protocol doesn't have the header size restriction like the HTTP protocol, so it avoids this issue. Make sure to use the latest version of the SDK, which has a fix for query operations when the service interop isn't available.
2. If the direct connection mode with the TCP protocol isn't an option for your workload, mitigate it by changing the client consistency level. The session token is only used for session consistency, which is the default consistency level for Azure Cosmos DB. Other consistency levels don't use the session token.

Continuation token is too large

Cause:

The 400 bad request occurs on query operations where the continuation token is used if the continuation token has grown too large or if different queries have different continuation token sizes.

Solution:

1. Follow the guidance in the .NET v3 or .NET v2 performance tips articles. Convert the application to use the direct connection mode with the TCP protocol. The direct connection mode with the TCP protocol doesn't have the header size restriction like the HTTP protocol, so it avoids this issue.
2. If the direct connection mode with the TCP protocol isn't an option for your workload, set the `ResponseContinuationTokenLimitInKb` option. You can find this option in `FeedOptions` in v2 or

`QueryRequestOptions` in v3.

Next steps

- Diagnose and troubleshoot issues when you use the Azure Cosmos DB .NET SDK.
- Learn about performance guidelines for .NET v3 and .NET v2.

Diagnose and troubleshoot Azure Cosmos DB .NET SDK request timeout exceptions

12/4/2020 • 3 minutes to read • Edit Online

APPLIES TO: SQL API

The HTTP 408 error occurs if the SDK was unable to complete the request before the timeout limit occurred.

Customize the timeout on the Azure Cosmos DB .NET SDK

The SDK has two distinct alternatives to control timeouts, each with a different scope.

RequestTimeout

The `CosmosClientOptions.RequestTimeout` (or `ConnectionPolicy.RequestTimeout` for SDK v2) configuration allows you to set a timeout that affects each individual network request. An operation started by a user can span multiple network requests (for example, there could be throttling). This configuration would apply for each network request on the retry. This timeout isn't an end-to-end operation request timeout.

CancellationToken

All the async operations in the SDK have an optional CancellationToken parameter. This CancellationToken parameter is used throughout the entire operation, across all network requests. In between network requests, the cancellation token might be checked and an operation canceled if the related token is expired. The cancellation token should be used to define an approximate expected timeout on the operation scope.

> **NOTE**
>
> The `CancellationToken` parameter is a mechanism where the library will check the cancellation when it won't cause an invalid state. The operation might not cancel exactly when the time defined in the cancellation is up. Instead, after the time is up, it cancels when it's safe to do so.

Troubleshooting steps

The following list contains known causes and solutions for request timeout exceptions.

High CPU utilization

High CPU utilization is the most common case. For optimal latency, CPU usage should be roughly 40 percent. Use 10 seconds as the interval to monitor maximum (not average) CPU utilization. CPU spikes are more common with cross-partition queries where it might do multiple connections for a single query.

Solution:

The client application that uses the SDK should be scaled up or out.

Socket or port availability might be low

When running in Azure, clients using the .NET SDK can hit Azure SNAT (PAT) port exhaustion.

Solution 1:

If you're running on Azure VMs, follow the SNAT port exhaustion guide.

Solution 2:

If you're running on Azure App Service, follow the connection errors troubleshooting guide and use App Service diagnostics.

Solution 3:

If you're running on Azure Functions, verify you're following the Azure Functions recommendation of maintaining singleton or static clients for all of the involved services (including Azure Cosmos DB). Check the service limits based on the type and size of your Function App hosting.

Solution 4:

If you use an HTTP proxy, make sure it can support the number of connections configured in the SDK `ConnectionPolicy`. Otherwise, you'll face connection issues.

Create multiple client instances

Creating multiple client instances might lead to connection contention and timeout issues.

Solution:

Follow the performance tips, and use a single CosmosClient instance across an entire process.

Hot partition key

Azure Cosmos DB distributes the overall provisioned throughput evenly across physical partitions. When there's a hot partition, one or more logical partition keys on a physical partition are consuming all the physical partition's Request Units per second (RU/s). At the same time, the RU/s on other physical partitions are going unused. As a symptom, the total RU/s consumed will be less than the overall provisioned RU/s at the database or container, but you'll still see throttling (429s) on the requests against the hot logical partition key. Use the Normalized RU Consumption metric to see if the workload is encountering a hot partition.

Solution:

Choose a good partition key that evenly distributes request volume and storage. Learn how to change your partition key.

High degree of concurrency

The application is doing a high level of concurrency, which can lead to contention on the channel.

Solution:

The client application that uses the SDK should be scaled up or out.

Large requests or responses

Large requests or responses can lead to head-of-line blocking on the channel and exacerbate contention, even with a relatively low degree of concurrency.

Solution:

The client application that uses the SDK should be scaled up or out.

Failure rate is within the Azure Cosmos DB SLA

The application should be able to handle transient failures and retry when necessary. Any 408 exceptions aren't retried because on create paths it's impossible to know if the service created the item or not. Sending the same item again for create will cause a conflict exception. User applications business logic might have custom logic to handle conflicts, which would break from the ambiguity of an existing item versus conflict from a create retry.

Failure rate violates the Azure Cosmos DB SLA

Contact Azure Support.

Next steps

- Diagnose and troubleshoot issues when you use the Azure Cosmos DB .NET SDK.
- Learn about performance guidelines for .NET v3 and .NET v2.

Diagnose and troubleshoot Azure Cosmos DB service unavailable exceptions

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: SQL API

The SDK wasn't able to connect to Azure Cosmos DB.

Troubleshooting steps

The following list contains known causes and solutions for service unavailable exceptions.

The required ports are being blocked

Verify that all the required ports are enabled.

Client-side transient connectivity issues

Service unavailable exceptions can surface when there are transient connectivity problems that are causing timeouts. Typically, the stack trace related to this scenario will contain a `TransportException` error. For example:

```
TransportException: A client transport error occurred: The request timed out while waiting for a server
response.
(Time: xxx, activity ID: xxx, error code: ReceiveTimeout [0x0010], base error: HRESULT 0x80131500
```

Follow the request timeout troubleshooting steps to resolve it.

Service outage

Check the Azure status to see if there's an ongoing issue.

Next steps

- Diagnose and troubleshoot issues when you use the Azure Cosmos DB .NET SDK.
- Learn about performance guidelines for .NET v3 and .NET v2.

Troubleshoot issues when you use Azure Cosmos DB Java SDK v4 with SQL API accounts

12/4/2020 • 9 minutes to read • Edit Online

APPLIES TO: SQL API

> **IMPORTANT**
>
> This article covers troubleshooting for Azure Cosmos DB Java SDK v4 only. Please see the Azure Cosmos DB Java SDK v4 Release notes, Maven repository, and performance tips for more information. If you are currently using an older version than v4, see the Migrate to Azure Cosmos DB Java SDK v4 guide for help upgrading to v4.

This article covers common issues, workarounds, diagnostic steps, and tools when you use Azure Cosmos DB Java SDK v4 with Azure Cosmos DB SQL API accounts. Azure Cosmos DB Java SDK v4 provides client-side logical representation to access the Azure Cosmos DB SQL API. This article describes tools and approaches to help you if you run into any issues.

Start with this list:

- Take a look at the Common issues and workarounds section in this article.
- Look at the Java SDK in the Azure Cosmos DB central repo, which is available open source on GitHub. It has an issues section that's actively monitored. Check to see if any similar issue with a workaround is already filed. One helpful tip is to filter issues by the *cosmos:v4-item* tag.
- Review the performance tips for Azure Cosmos DB Java SDK v4, and follow the suggested practices.
- Read the rest of this article, if you didn't find a solution. Then file a GitHub issue. If there is an option to add tags to your GitHub issue, add a *cosmos:v4-item* tag.

Common issues and workarounds

Network issues, Netty read timeout failure, low throughput, high latency

General suggestions

For best performance:

- Make sure the app is running on the same region as your Azure Cosmos DB account.
- Check the CPU usage on the host where the app is running. If CPU usage is 50 percent or more, run your app on a host with a higher configuration. Or you can distribute the load on more machines.
  - If you are running your application on Azure Kubernetes Service, you can use Azure Monitor to monitor CPU utilization.

Connection throttling

Connection throttling can happen because of either a connection limit on a host machine or Azure SNAT (PAT) port exhaustion.

Connection limit on a host machine

Some Linux systems, such as Red Hat, have an upper limit on the total number of open files. Sockets in Linux are implemented as files, so this number limits the total number of connections, too. Run the following command.

```
ulimit -a
```

The number of max allowed open files, which are identified as "nofile," needs to be at least double your connection pool size. For more information, see the Azure Cosmos DB Java SDK v4 performance tips.

Azure SNAT (PAT) port exhaustion

If your app is deployed on Azure Virtual Machines without a public IP address, by default Azure SNAT ports establish connections to any endpoint outside of your VM. The number of connections allowed from the VM to the Azure Cosmos DB endpoint is limited by the Azure SNAT configuration.

Azure SNAT ports are used only when your VM has a private IP address and a process from the VM tries to connect to a public IP address. There are two workarounds to avoid Azure SNAT limitation:

- Add your Azure Cosmos DB service endpoint to the subnet of your Azure Virtual Machines virtual network. For more information, see Azure Virtual Network service endpoints.

  When the service endpoint is enabled, the requests are no longer sent from a public IP to Azure Cosmos DB. Instead, the virtual network and subnet identity are sent. This change might result in firewall drops if only public IPs are allowed. If you use a firewall, when you enable the service endpoint, add a subnet to the firewall by using Virtual Network ACLs.

- Assign a public IP to your Azure VM.

Can't reach the Service - firewall

`ConnectTimeoutException` indicates that the SDK cannot reach the service. You may get a failure similar to the following when using the direct mode:

```
GoneException{error=null, resourceAddress='https://cdb-ms-prod-westus-
fd4.documents.azure.com:14940/apps/e41242a5-2d71-5acb-2e00-5e5f744b12de/services/d8aa21a5-340b-21d4-b1a2-
4a5333e7ed8a/partitions/ed028254-b613-4c2a-bf3c-14bd5eb64500/replicas/131298754052060051p//',
statusCode=410, message=Message: The requested resource is no longer available at the server.,
getCauseInfo=[class: class io.netty.channel.ConnectTimeoutException, message: connection timed out: cdb-ms-
prod-westus-fd4.documents.azure.com/101.13.12.5:14940]
```

If you have a firewall running on your app machine, open port range 10,000 to 20,000 which are used by the direct mode. Also follow the Connection limit on a host machine.

HTTP proxy

If you use an HTTP proxy, make sure it can support the number of connections configured in the SDK `ConnectionPolicy`. Otherwise, you face connection issues.

Invalid coding pattern: Blocking Netty IO thread

The SDK uses the Netty IO library to communicate with Azure Cosmos DB. The SDK has an Async API and uses non-blocking IO APIs of Netty. The SDK's IO work is performed on IO Netty threads. The number of IO Netty threads is configured to be the same as the number of CPU cores of the app machine.

The Netty IO threads are meant to be used only for non-blocking Netty IO work. The SDK returns the API invocation result on one of the Netty IO threads to the app's code. If the app performs a long-lasting operation after it receives results on the Netty thread, the SDK might not have enough IO threads to perform its internal IO work. Such app coding might result in low throughput, high latency, and `io.netty.handler.timeout.ReadTimeoutException` failures. The workaround is to switch the thread when you know the operation takes time.

For example, take a look at the following code snippet which adds items to a container (look here for guidance on setting up the database and container.) You might perform long-lasting work that takes more than a few milliseconds on the Netty thread. If so, you eventually can get into a state where no Netty IO thread is present to process IO work. As a result, you get a ReadTimeoutException failure.

Java SDK V4 (Maven com.azure::azure-cosmos) Async API

```
//Bad code with read timeout exception

int requestTimeoutInSeconds = 10;

/* ... */

AtomicInteger failureCount = new AtomicInteger();
// Max number of concurrent item inserts is # CPU cores + 1
Flux<Family> familyPub =
        Flux.just(Families.getAndersenFamilyItem(), Families.getAndersenFamilyItem(),
Families.getJohnsonFamilyItem());
familyPub.flatMap(family -> {
    return container.createItem(family);
}).flatMap(r -> {
    try {
        // Time-consuming work is, for example,
        // writing to a file, computationally heavy work, or just sleep.
        // Basically, it's anything that takes more than a few milliseconds.
        // Doing such operations on the IO Netty thread
        // without a proper scheduler will cause problems.
        // The subscriber will get a ReadTimeoutException failure.
        TimeUnit.SECONDS.sleep(2 * requestTimeoutInSeconds);
    } catch (Exception e) {
    }
    return Mono.empty();
}).doOnError(Exception.class, exception -> {
    failureCount.incrementAndGet();
}).blockLast();
assert(failureCount.get() > 0);
```

The workaround is to change the thread on which you perform work that takes time. Define a singleton instance of the scheduler for your app.

Java SDK V4 (Maven com.azure::azure-cosmos) Async API

```
// Have a singleton instance of an executor and a scheduler.
ExecutorService ex  = Executors.newFixedThreadPool(30);
Scheduler customScheduler = Schedulers.fromExecutor(ex);
```

You might need to do work that takes time, for example, computationally heavy work or blocking IO. In this case, switch the thread to a worker provided by your `customScheduler` by using the `.publishOn(customScheduler)` API.

Java SDK V4 (Maven com.azure::azure-cosmos) Async API

```
container.createItem(family)
        .publishOn(customScheduler) // Switches the thread.
        .subscribe(
                // ...
        );
```

By using `publishOn(customScheduler)`, you release the Netty IO thread and switch to your own custom thread provided by the custom scheduler. This modification solves the problem. You won't get a `io.netty.handler.timeout.ReadTimeoutException` failure anymore.

Request rate too large

This failure is a server-side failure. It indicates that you consumed your provisioned throughput. Retry later. If you get this failure often, consider an increase in the collection throughput.

- **Implement backoff at getRetryAfterInMilliseconds intervals**

  During performance testing, you should increase load until a small rate of requests get throttled. If throttled, the client application should backoff for the server-specified retry interval. Respecting the backoff ensures that you spend minimal amount of time waiting between retries.

Failure connecting to Azure Cosmos DB Emulator

The Azure Cosmos DB Emulator HTTPS certificate is self-signed. For the SDK to work with the emulator, import the emulator certificate to a Java TrustStore. For more information, see Export Azure Cosmos DB Emulator certificates.

Dependency Conflict Issues

The Azure Cosmos DB Java SDK pulls in a number of dependencies; generally speaking, if your project dependency tree includes an older version of an artifact that Azure Cosmos DB Java SDK depends on, this may result in unexpected errors being generated when you run your application. If you are debugging why your application unexpectedly throws an exception, it is a good idea to double-check that your dependency tree is not accidentally pulling in an older version of one or more of the Azure Cosmos DB Java SDK dependencies.

The workaround for such an issue is to identify which of your project dependencies brings in the old version and exclude the transitive dependency on that older version, and allow Azure Cosmos DB Java SDK to bring in the newer version.

To identify which of your project dependencies brings in an older version of something that Azure Cosmos DB Java SDK depends on, run the following command against your project pom.xml file:

```
mvn dependency:tree
```

For more information, see the maven dependency tree guide.

Once you know which dependency of your project depends on an older version, you can modify the dependency on that lib in your pom file and exclude the transitive dependency, following the example below (which assumes that *reactor-core* is the outdated dependency):

```
<dependency>
  <groupId>${groupid-of-lib-which-brings-in-reactor}</groupId>
  <artifactId>${artifactId-of-lib-which-brings-in-reactor}</artifactId>
  <version>${version-of-lib-which-brings-in-reactor}</version>
  <exclusions>
    <exclusion>
      <groupId>io.projectreactor</groupId>
      <artifactId>reactor-core</artifactId>
    </exclusion>
  </exclusions>
</dependency>
```

For more information, see the exclude transitive dependency guide.

Enable client SDK logging

Azure Cosmos DB Java SDK v4 uses SLF4j as the logging facade that supports logging into popular logging frameworks such as log4j and logback.

For example, if you want to use log4j as the logging framework, add the following libs in your Java classpath.

```
<dependency>
  <groupId>org.slf4j</groupId>
  <artifactId>slf4j-log4j12</artifactId>
  <version>${slf4j.version}</version>
</dependency>
<dependency>
  <groupId>log4j</groupId>
  <artifactId>log4j</artifactId>
  <version>${log4j.version}</version>
</dependency>
```

Also add a log4j config.

```
# this is a sample log4j configuration

# Set root logger level to INFO and its only appender to A1.
log4j.rootLogger=INFO, A1

log4j.category.com.azure.cosmos=INFO
#log4j.category.io.netty=OFF
#log4j.category.io.projectreactor=OFF
# A1 is set to be a ConsoleAppender.
log4j.appender.A1=org.apache.log4j.ConsoleAppender

# A1 uses PatternLayout.
log4j.appender.A1.layout=org.apache.log4j.PatternLayout
log4j.appender.A1.layout.ConversionPattern=%d %5X{pid} [%t] %-5p %c - %m%n
```

For more information, see the sfl4j logging manual.

OS network statistics

Run the netstat command to get a sense of how many connections are in states such as `ESTABLISHED` and `CLOSE_WAIT`.

On Linux, you can run the following command.

```
netstat -nap
```

On Windows, you can run the same command with different argument flags:

```
netstat -abn
```

Filter the result to only connections to the Azure Cosmos DB endpoint.

The number of connections to the Azure Cosmos DB endpoint in the `ESTABLISHED` state can't be greater than your configured connection pool size.

Many connections to the Azure Cosmos DB endpoint might be in the `CLOSE_WAIT` state. There might be more than 1,000. A number that high indicates that connections are established and torn down quickly. This situation potentially causes problems. For more information, see the Common issues and workarounds section.

Diagnose and troubleshoot Azure Cosmos DB Java v4 SDK request timeout exceptions

12/4/2020 • 4 minutes to read • Edit Online

APPLIES TO: SQL API

The HTTP 408 error occurs if the SDK was unable to complete the request before the timeout limit occurred.

Troubleshooting steps

The following list contains known causes and solutions for request timeout exceptions.

Existing issues

If you are seeing requests getting stuck for longer duration or timing out more frequently, please upgrade the Java v4 SDK to the latest version. NOTE: We strongly recommend to use the version 4.7.0 and above. Checkout the Java v4 SDK release notes for more details.

High CPU utilization

High CPU utilization is the most common case. For optimal latency, CPU usage should be roughly 40 percent. Use 10 seconds as the interval to monitor maximum (not average) CPU utilization. CPU spikes are more common with cross-partition queries where it might do multiple connections for a single query.

Solution:

The client application that uses the SDK should be scaled up or out.

Connection throttling

Connection throttling can happen because of either a connection limit on a host machine or Azure SNAT (PAT) port exhaustion.

Connection limit on a host machine

Some Linux systems, such as Red Hat, have an upper limit on the total number of open files. Sockets in Linux are implemented as files, so this number limits the total number of connections, too. Run the following command.

```
ulimit -a
```

Solution:

The number of max allowed open files, which are identified as "nofile," needs to be at least 10,000 or more. For more information, see the Azure Cosmos DB Java SDK v4 performance tips.

Socket or port availability might be low

When running in Azure, clients using the Java SDK can hit Azure SNAT (PAT) port exhaustion.

Solution 1:

If you're running on Azure VMs, follow the SNAT port exhaustion guide.

Solution 2:

If you're running on Azure App Service, follow the connection errors troubleshooting guide and use App Service diagnostics.

Solution 3:

If you're running on Azure Functions, verify you're following the Azure Functions recommendation of maintaining

singleton or static clients for all of the involved services (including Azure Cosmos DB). Check the service limits based on the type and size of your Function App hosting.

Solution 4:

If you use an HTTP proxy, make sure it can support the number of connections configured in the SDK `GatewayConnectionConfig`. Otherwise, you'll face connection issues.

Create multiple client instances

Creating multiple client instances might lead to connection contention and timeout issues.

Solution 1:

Follow the performance tips, and use a single CosmosClient instance across an entire application.

Solution 2:

If singleton CosmosClient is not possible to have in an application, we recommend using connection sharing across multiple Cosmos Clients through this API `connectionSharingAcrossClientsEnabled(true)` in CosmosClient. When you have multiple instances of Cosmos Client in the same JVM interacting to multiple Cosmos accounts, enabling this allows connection sharing in Direct mode if possible between instances of Cosmos Client. Please note, when setting this option, the connection configuration (e.g., socket timeout config, idle timeout config) of the first instantiated client will be used for all other client instances.

Hot partition key

Azure Cosmos DB distributes the overall provisioned throughput evenly across physical partitions. When there's a hot partition, one or more logical partition keys on a physical partition are consuming all the physical partition's Request Units per second (RU/s). At the same time, the RU/s on other physical partitions are going unused. As a symptom, the total RU/s consumed will be less than the overall provisioned RU/s at the database or container, but you'll still see throttling (429s) on the requests against the hot logical partition key. Use the Normalized RU Consumption metric to see if the workload is encountering a hot partition.

Solution:

Choose a good partition key that evenly distributes request volume and storage. Learn how to change your partition key.

High degree of concurrency

The application is doing a high level of concurrency, which can lead to contention on the channel.

Solution:

The client application that uses the SDK should be scaled up or out.

Large requests or responses

Large requests or responses can lead to head-of-line blocking on the channel and exacerbate contention, even with a relatively low degree of concurrency.

Solution:

The client application that uses the SDK should be scaled up or out.

Failure rate is within the Azure Cosmos DB SLA

The application should be able to handle transient failures and retry when necessary. Any 408 exceptions aren't retried because on create paths it's impossible to know if the service created the item or not. Sending the same item again for create will cause a conflict exception. User applications business logic might have custom logic to handle conflicts, which would break from the ambiguity of an existing item versus conflict from a create retry.

Failure rate violates the Azure Cosmos DB SLA

Contact Azure Support.

Next steps

- Diagnose and troubleshoot issues when you use the Azure Cosmos DB Java v4 SDK.
- Learn about performance guidelines for Java v4.

Diagnose and troubleshoot Azure Cosmos DB Java v4 SDK service unavailable exceptions

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: SQL API

The Java v4 SDK wasn't able to connect to Azure Cosmos DB.

Troubleshooting steps

The following list contains known causes and solutions for service unavailable exceptions.

The required ports are being blocked

Verify that all the required ports are enabled.

Client-side transient connectivity issues

Service unavailable exceptions can surface when there are transient connectivity problems that are causing timeouts. Typically, the stack trace related to this scenario will contain a `ServiceUnavailableException` error with diagnostic details. For example:

```
Exception in thread "main" ServiceUnavailableException{userAgent=azsdk-java-cosmos/4.6.0 Linux/4.15.0-1096-azure JRE/11.0.8, error=null, resourceAddress='null', requestUri='null', statusCode=503, message=Service is currently unavailable, please retry after a while. If this problem persists please contact support.: Message: "" {"diagnostics"}
```

Follow the request timeout troubleshooting steps to resolve it.

Service outage

Check the Azure status to see if there's an ongoing issue.

Next steps

- Diagnose and troubleshoot issues when you use the Azure Cosmos DB Java v4 SDK.
- Learn about performance guidelines for Java v4 SDK.

Troubleshoot issues when you use the Azure Cosmos DB Async Java SDK v2 with SQL API accounts

12/4/2020 • 8 minutes to read • Edit Online

APPLIES TO: SQL API

> **IMPORTANT**
>
> This is *not* the latest Java SDK for Azure Cosmos DB! You should upgrade your project to Azure Cosmos DB Java SDK v4 and then read the Azure Cosmos DB Java SDK v4 troubleshooting guide. Follow the instructions in the Migrate to Azure Cosmos DB Java SDK v4 guide and Reactor vs RxJava guide to upgrade.
>
> This article covers troubleshooting for Azure Cosmos DB Async Java SDK v2 only. See the Azure Cosmos DB Async Java SDK v2 Release Notes, Maven repository and performance tips for more information.

This article covers common issues, workarounds, diagnostic steps, and tools when you use the Java Async SDK with Azure Cosmos DB SQL API accounts. The Java Async SDK provides client-side logical representation to access the Azure Cosmos DB SQL API. This article describes tools and approaches to help you if you run into any issues.

Start with this list:

- Take a look at the Common issues and workarounds section in this article.
- Look at the SDK, which is available open source on GitHub. It has an issues section that's actively monitored. Check to see if any similar issue with a workaround is already filed.
- Review the performance tips, and follow the suggested practices.
- Read the rest of this article, if you didn't find a solution. Then file a GitHub issue.

Common issues and workarounds

Network issues, Netty read timeout failure, low throughput, high latency

General suggestions

- Make sure the app is running on the same region as your Azure Cosmos DB account.
- Check the CPU usage on the host where the app is running. If CPU usage is 90 percent or more, run your app on a host with a higher configuration. Or you can distribute the load on more machines.

Connection throttling

Connection throttling can happen because of either a connection limit on a host machine or Azure SNAT (PAT) port exhaustion.

Connection limit on a host machine

Some Linux systems, such as Red Hat, have an upper limit on the total number of open files. Sockets in Linux are implemented as files, so this number limits the total number of connections, too. Run the following command.

```
ulimit -a
```

The number of max allowed open files, which are identified as "nofile," needs to be at least double your connection pool size. For more information, see Performance tips.

Azure SNAT (PAT) port exhaustion

If your app is deployed on Azure Virtual Machines without a public IP address, by default Azure SNAT ports

establish connections to any endpoint outside of your VM. The number of connections allowed from the VM to the Azure Cosmos DB endpoint is limited by the Azure SNAT configuration.

Azure SNAT ports are used only when your VM has a private IP address and a process from the VM tries to connect to a public IP address. There are two workarounds to avoid Azure SNAT limitation:

- Add your Azure Cosmos DB service endpoint to the subnet of your Azure Virtual Machines virtual network. For more information, see Azure Virtual Network service endpoints.

  When the service endpoint is enabled, the requests are no longer sent from a public IP to Azure Cosmos DB. Instead, the virtual network and subnet identity are sent. This change might result in firewall drops if only public IPs are allowed. If you use a firewall, when you enable the service endpoint, add a subnet to the firewall by using Virtual Network ACLs.

- Assign a public IP to your Azure VM.

Can't reach the Service - firewall

`ConnectTimeoutException` indicates that the SDK cannot reach the service. You may get a failure similar to the following when using the direct mode:

```
GoneException{error=null, resourceAddress='https://cdb-ms-prod-westus-
fd4.documents.azure.com:14940/apps/e41242a5-2d71-5acb-2e00-5e5f744b12de/services/d8aa21a5-340b-21d4-b1a2-
4a5333e7ed8a/partitions/ed028254-b613-4c2a-bf3c-14bd5eb64500/replicas/131298754052060051p//', statusCode=410,
message=Message: The requested resource is no longer available at the server., getCauseInfo=[class: class
io.netty.channel.ConnectTimeoutException, message: connection timed out: cdb-ms-prod-westus-
fd4.documents.azure.com/101.13.12.5:14940]
```

If you have a firewall running on your app machine, open port range 10,000 to 20,000 which are used by the direct mode. Also follow the Connection limit on a host machine.

HTTP proxy

If you use an HTTP proxy, make sure it can support the number of connections configured in the SDK `ConnectionPolicy`. Otherwise, you face connection issues.

Invalid coding pattern: Blocking Netty IO thread

The SDK uses the Netty IO library to communicate with Azure Cosmos DB. The SDK has Async APIs and uses non-blocking IO APIs of Netty. The SDK's IO work is performed on IO Netty threads. The number of IO Netty threads is configured to be the same as the number of CPU cores of the app machine.

The Netty IO threads are meant to be used only for non-blocking Netty IO work. The SDK returns the API invocation result on one of the Netty IO threads to the app's code. If the app performs a long-lasting operation after it receives results on the Netty thread, the SDK might not have enough IO threads to perform its internal IO work. Such app coding might result in low throughput, high latency, and `io.netty.handler.timeout.ReadTimeoutException` failures. The workaround is to switch the thread when you know the operation takes time.

For example, take a look at the following code snippet. You might perform long-lasting work that takes more than a few milliseconds on the Netty thread. If so, you eventually can get into a state where no Netty IO thread is present to process IO work. As a result, you get a ReadTimeoutException failure.

Async Java SDK V2 (Maven com.microsoft.azure::azure-cosmosdb)

```
@Test
public void badCodeWithReadTimeoutException() throws Exception {
    int requestTimeoutInSeconds = 10;

    ConnectionPolicy policy = new ConnectionPolicy();
    policy.setRequestTimeoutInMillis(requestTimeoutInSeconds * 1000);

    AsyncDocumentClient testClient = new AsyncDocumentClient.Builder()
            .withServiceEndpoint(TestConfigurations.HOST)
            .withMasterKeyOrResourceToken(TestConfigurations.MASTER_KEY)
            .withConnectionPolicy(policy)
            .build();

    int numberOfCpuCores = Runtime.getRuntime().availableProcessors();
    int numberOfConcurrentWork = numberOfCpuCores + 1;
    CountDownLatch latch = new CountDownLatch(numberOfConcurrentWork);
    AtomicInteger failureCount = new AtomicInteger();

    for (int i = 0; i < numberOfConcurrentWork; i++) {
        Document docDefinition = getDocumentDefinition();
        Observable<ResourceResponse<Document>> createObservable = testClient
                .createDocument(getCollectionLink(), docDefinition, null, false);
        createObservable.subscribe(r -> {
                    try {
                        // Time-consuming work is, for example,
                        // writing to a file, computationally heavy work, or just sleep.
                        // Basically, it's anything that takes more than a few milliseconds.
                        // Doing such operations on the IO Netty thread
                        // without a proper scheduler will cause problems.
                        // The subscriber will get a ReadTimeoutException failure.
                        TimeUnit.SECONDS.sleep(2 * requestTimeoutInSeconds);
                    } catch (Exception e) {
                    }
                },

                exception -> {
                    //It will be io.netty.handler.timeout.ReadTimeoutException.
                    exception.printStackTrace();
                    failureCount.incrementAndGet();
                    latch.countDown();
                },
                () -> {
                    latch.countDown();
                });
    }

    latch.await();
    assertThat(failureCount.get()).isGreaterThan(0);
}
```

The workaround is to change the thread on which you perform work that takes time. Define a singleton instance of the scheduler for your app.

Async Java SDK V2 (Maven com.microsoft.azure::azure-cosmosdb)

```
// Have a singleton instance of an executor and a scheduler.
ExecutorService ex  = Executors.newFixedThreadPool(30);
Scheduler customScheduler = rx.schedulers.Schedulers.from(ex);
```

You might need to do work that takes time, for example, computationally heavy work or blocking IO. In this case, switch the thread to a worker provided by your `customScheduler` by using the `.observeOn(customScheduler)` API.

Async Java SDK V2 (Maven com.microsoft.azure::azure-cosmosdb)

```
Observable<ResourceResponse<Document>> createObservable = client
        .createDocument(getCollectionLink(), docDefinition, null, false);

createObservable
        .observeOn(customScheduler) // Switches the thread.
        .subscribe(
            // ...
        );
```

By using `observeOn(customScheduler)`, you release the Netty IO thread and switch to your own custom thread provided by the custom scheduler. This modification solves the problem. You won't get a `io.netty.handler.timeout.ReadTimeoutException` failure anymore.

Connection pool exhausted issue

`PoolExhaustedException` is a client-side failure. This failure indicates that your app workload is higher than what the SDK connection pool can serve. Increase the connection pool size or distribute the load on multiple apps.

Request rate too large

This failure is a server-side failure. It indicates that you consumed your provisioned throughput. Retry later. If you get this failure often, consider an increase in the collection throughput.

Failure connecting to Azure Cosmos DB Emulator

The Azure Cosmos DB Emulator HTTPS certificate is self-signed. For the SDK to work with the emulator, import the emulator certificate to a Java TrustStore. For more information, see Export Azure Cosmos DB Emulator certificates.

Dependency Conflict Issues

```
Exception in thread "main" java.lang.NoSuchMethodError: rx.Observable.toSingle()Lrx/Single;
```

The above exception suggests you have a dependency on an older version of RxJava lib (e.g., 1.2.2). Our SDK relies on RxJava 1.3.8 which has APIs not available in earlier version of RxJava.

The workaround for such issues is to identify which other dependency brings in RxJava-1.2.2 and exclude the transitive dependency on RxJava-1.2.2, and allow CosmosDB SDK bring the newer version.

To identify which library brings in RxJava-1.2.2 run the following command next to your project pom.xml file:

```
mvn dependency:tree
```

For more information, see the maven dependency tree guide.

Once you identify RxJava-1.2.2 is transitive dependency of which other dependency of your project, you can modify the dependency on that lib in your pom file and exclude RxJava transitive dependency it:

```
<dependency>
  <groupId>${groupid-of-lib-which-brings-in-rxjava1.2.2}</groupId>
  <artifactId>${artifactId-of-lib-which-brings-in-rxjava1.2.2}</artifactId>
  <version>${version-of-lib-which-brings-in-rxjava1.2.2}</version>
  <exclusions>
    <exclusion>
      <groupId>io.reactivex</groupId>
      <artifactId>rxjava</artifactId>
    </exclusion>
  </exclusions>
</dependency>
```

For more information, see the exclude transitive dependency guide.

Enable client SDK logging

The Java Async SDK uses SLF4j as the logging facade that supports logging into popular logging frameworks such as log4j and logback.

For example, if you want to use log4j as the logging framework, add the following libs in your Java classpath.

```
<dependency>
  <groupId>org.slf4j</groupId>
  <artifactId>slf4j-log4j12</artifactId>
  <version>${slf4j.version}</version>
</dependency>
<dependency>
  <groupId>log4j</groupId>
  <artifactId>log4j</artifactId>
  <version>${log4j.version}</version>
</dependency>
```

Also add a log4j config.

```
# this is a sample log4j configuration

# Set root logger level to DEBUG and its only appender to A1.
log4j.rootLogger=INFO, A1

log4j.category.com.microsoft.azure.cosmosdb=DEBUG
#log4j.category.io.netty=INFO
#log4j.category.io.reactivex=INFO
# A1 is set to be a ConsoleAppender.
log4j.appender.A1=org.apache.log4j.ConsoleAppender

# A1 uses PatternLayout.
log4j.appender.A1.layout=org.apache.log4j.PatternLayout
log4j.appender.A1.layout.ConversionPattern=%d %5X{pid} [%t] %-5p %c - %m%n
```

For more information, see the sfl4j logging manual.

OS network statistics

Run the netstat command to get a sense of how many connections are in states such as `ESTABLISHED` and `CLOSE_WAIT`.

On Linux, you can run the following command.

```
netstat -nap
```

Filter the result to only connections to the Azure Cosmos DB endpoint.

The number of connections to the Azure Cosmos DB endpoint in the `ESTABLISHED` state can't be greater than your configured connection pool size.

Many connections to the Azure Cosmos DB endpoint might be in the `CLOSE_WAIT` state. There might be more than 1,000. A number that high indicates that connections are established and torn down quickly. This situation potentially causes problems. For more information, see the Common issues and workarounds section.

How to configure logging and connectivity with the Azure Functions trigger for Cosmos DB

11/2/2020 • 2 minutes to read • Edit Online

APPLIES TO: SQL API

This article describes advanced configuration options you can set when using the Azure Functions trigger for Cosmos DB.

Enabling trigger specific logs

The Azure Functions trigger for Cosmos DB uses the Change Feed Processor Library internally, and the library generates a set of health logs that can be used to monitor internal operations for troubleshooting purposes.

The health logs describe how the Azure Functions trigger for Cosmos DB behaves when attempting operations during load-balancing scenarios or initialization.

Enabling logging

To enable logging when using Azure Functions trigger for Cosmos DB, locate the `host.json` file in your Azure Functions project or Azure Functions App and configure the level of required logging. Enable the traces for `Host.Triggers.CosmosDB` as shown in the following sample:

```
{
  "version": "2.0",
  "logging": {
    "fileLoggingMode": "always",
    "logLevel": {
      "Host.Triggers.CosmosDB": "Trace"
    }
  }
}
```

After the Azure Function is deployed with the updated configuration, you will see the Azure Functions trigger for Cosmos DB logs as part of your traces. You can view the logs in your configured logging provider under the *Category* `Host.Triggers.CosmosDB`.

Query the logs

Run the following query to query the logs generated by the Azure Functions trigger for Cosmos DB in Azure Application Insights' Analytics:

```
traces
| where customDimensions.Category == "Host.Triggers.CosmosDB"
```

Configuring the connection policy

There are two connection modes - Direct mode and Gateway mode. To learn more about these connection modes, see the connection modes article. By default, **Gateway** is used to establish all connections on the Azure Functions trigger for Cosmos DB. However, it might not be the best option for performance-driven scenarios.

Changing the connection mode and protocol

There are two key configuration settings available to configure the client connection policy – the **connection mode** and the **connection protocol**. You can change the default connection mode and protocol used by the Azure Functions trigger for Cosmos DB and all the Azure Cosmos DB bindings). To change the default settings, you need to locate the `host.json` file in your Azure Functions project or Azure Functions App and add the following extra setting:

```
{
  "cosmosDB": {
    "connectionMode": "Direct",
    "protocol": "Tcp"
  }
}
```

Where `connectionMode` must have the desired connection mode (Direct or Gateway) and `protocol` the desired connection protocol (Tcp or Https).

If your Azure Functions project is working with Azure Functions V1 runtime, the configuration has a slight name difference, you should use `documentDB` instead of `cosmosDB`:

```
{
  "documentDB": {
    "connectionMode": "Direct",
    "protocol": "Tcp"
  }
}
```

> **NOTE**
> When working with Azure Functions Consumption Plan Hosting plan, each instance has a limit in the amount of Socket Connections that it can maintain. When working with Direct / TCP mode, by design more connections are created and can hit the Consumption Plan limit, in which case you can either use Gateway mode or run your Azure Functions in App Service Mode.

Next steps

- Connection limits in Azure Functions
- Enable monitoring in your Azure Functions applications.
- Learn how to Diagnose and troubleshoot common issues when using the Azure Functions trigger for Cosmos DB.

Create multiple Azure Functions triggers for Cosmos DB

12/4/2020 • 3 minutes to read • Edit Online

APPLIES TO: SQL API

This article describes how you can configure multiple Azure Functions triggers for Cosmos DB to work in parallel and independently react to changes.

Event-based architecture requirements

When building serverless architectures with Azure Functions, it's recommended to create small function sets that work together instead of large long running functions.

As you build event-based serverless flows using the Azure Functions trigger for Cosmos DB, you'll run into the scenario where you want to do multiple things whenever there is a new event in a particular Azure Cosmos container. If actions you want to trigger, are independent from one another, the ideal solution would be to **create one Azure Functions triggers for Cosmos DB per action** you want to do, all listening for changes on the same Azure Cosmos container.

Optimizing containers for multiple Triggers

Given the *requirements* of the Azure Functions trigger for Cosmos DB, we need a second container to store state, also called, the *leases container*. Does this mean that you need a separate leases container for each Azure Function?

Here, you have two options:

- Create **one leases container per Function**: This approach can translate into additional costs, unless you're using a shared throughput database. Remember, that the minimum throughput at the container level is 400 Request Units, and in the case of the leases container, it is only being used to checkpoint the progress and maintain state.

- Have **one lease container and share it** for all your Functions: This second option makes better use of the provisioned Request Units on the container, as it enables multiple Azure Functions to share and use the same provisioned throughput.

The goal of this article is to guide you to accomplish the second option.

Configuring a shared leases container

To configure the shared leases container, the only extra configuration you need to make on your triggers is to add the `LeaseCollectionPrefix` attribute if you are using C# or `leaseCollectionPrefix` attribute if you are using JavaScript. The value of the attribute should be a logical descriptor of what that particular trigger.

For example, if you have three Triggers: one that sends emails, one that does an aggregation to create a materialized view, and one that sends the changes to another storage, for later analysis, you could assign the `LeaseCollectionPrefix` of "emails" to the first one, "materialized" to the second one, and "analytics" to the third one.

The important part is that all three Triggers **can use the same leases container configuration** (account, database, and container name).

A very simple code samples using the `LeaseCollectionPrefix` attribute in C#, would look like this:

```
using Microsoft.Azure.Documents;
using Microsoft.Azure.WebJobs;
using Microsoft.Azure.WebJobs.Host;
using System.Collections.Generic;
using Microsoft.Extensions.Logging;

[FunctionName("SendEmails")]
public static void SendEmails([CosmosDBTrigger(
    databaseName: "ToDoItems",
    collectionName: "Items",
    ConnectionStringSetting = "CosmosDBConnection",
    LeaseCollectionName = "leases",
    LeaseCollectionPrefix = "emails")]IReadOnlyList<Document> documents,
    ILogger log)
{
    ...
}

[FunctionName("MaterializedViews")]
public static void MaterializedViews([CosmosDBTrigger(
    databaseName: "ToDoItems",
    collectionName: "Items",
    ConnectionStringSetting = "CosmosDBConnection",
    LeaseCollectionName = "leases",
    LeaseCollectionPrefix = "materialized")]IReadOnlyList<Document> documents,
    ILogger log)
{
    ...
}
```

And for JavaScript, you can apply the configuration on the `function.json` file, with the `leaseCollectionPrefix` attribute:

```
{
    "type": "cosmosDBTrigger",
    "name": "documents",
    "direction": "in",
    "leaseCollectionName": "leases",
    "connectionStringSetting": "CosmosDBConnection",
    "databaseName": "ToDoItems",
    "collectionName": "Items",
    "leaseCollectionPrefix": "emails"
},
{
    "type": "cosmosDBTrigger",
    "name": "documents",
    "direction": "in",
    "leaseCollectionName": "leases",
    "connectionStringSetting": "CosmosDBConnection",
    "databaseName": "ToDoItems",
    "collectionName": "Items",
    "leaseCollectionPrefix": "materialized"
}
```

NOTE

Always monitor on the Request Units provisioned on your shared leases container. Each Trigger that shares it, will increase the throughput average consumption, so you might need to increase the provisioned throughput as you increase the number of Azure Functions that are using it.

Next steps

- See the full configuration for the Azure Functions trigger for Cosmos DB
- Check the extended list of samples for all the languages.
- Visit the Serverless recipes with Azure Cosmos DB and Azure Functions GitHub repository for more samples.

Diagnose and troubleshoot issues when using Azure Functions trigger for Cosmos DB

12/4/2020 • 9 minutes to read • Edit Online

APPLIES TO: SQL API

This article covers common issues, workarounds, and diagnostic steps, when you use the Azure Functions trigger for Cosmos DB.

Dependencies

The Azure Functions trigger and bindings for Cosmos DB depend on the extension packages over the base Azure Functions runtime. Always keep these packages updated, as they might include fixes and new features that might address any potential issues you may encounter:

- For Azure Functions V2, see Microsoft.Azure.WebJobs.Extensions.CosmosDB.
- For Azure Functions V1, see Microsoft.Azure.WebJobs.Extensions.DocumentDB.

This article will always refer to Azure Functions V2 whenever the runtime is mentioned, unless explicitly specified.

Consume the Azure Cosmos DB SDK independently

The key functionality of the extension package is to provide support for the Azure Functions trigger and bindings for Cosmos DB. It also includes the Azure Cosmos DB .NET SDK, which is helpful if you want to interact with Azure Cosmos DB programmatically without using the trigger and bindings.

If want to use the Azure Cosmos DB SDK, make sure that you don't add to your project another NuGet package reference. Instead, **let the SDK reference resolve through the Azure Functions' Extension package**. Consume the Azure Cosmos DB SDK separately from the trigger and bindings

Additionally, if you are manually creating your own instance of the Azure Cosmos DB SDK client, you should follow the pattern of having only one instance of the client using a Singleton pattern approach. This process will avoid the potential socket issues in your operations.

Common scenarios and workarounds

Azure Function fails with error message collection doesn't exist

Azure Function fails with error message "Either the source collection 'collection-name' (in database 'database-name') or the lease collection 'collection2-name' (in database 'database2-name') does not exist. Both collections must exist before the listener starts. To automatically create the lease collection, set 'CreateLeaseCollectionIfNotExists' to 'true'"

This means that either one or both of the Azure Cosmos containers required for the trigger to work do not exist or are not reachable to the Azure Function. **The error itself will tell you which Azure Cosmos database and container is the trigger looking for** based on your configuration.

1. Verify the `ConnectionStringSetting` attribute and that it **references a setting that exists in your Azure Function App**. The value on this attribute shouldn't be the Connection String itself, but the name of the Configuration Setting.
2. Verify that the `databaseName` and `collectionName` exist in your Azure Cosmos account. If you are using

automatic value replacement (using `%settingName%` patterns), make sure the name of the setting exists in your Azure Function App.

3. If you don't specify a `LeaseCollectionName/leaseCollectionName`, the default is "leases". Verify that such container exists. Optionally you can set the `CreateLeaseCollectionIfNotExists` attribute in your Trigger to `true` to automatically create it.
4. Verify your Azure Cosmos account's Firewall configuration to see to see that it's not it's not blocking the Azure Function.

Azure Function fails to start with "Shared throughput collection should have a partition key"

The previous versions of the Azure Cosmos DB Extension did not support using a leases container that was created within a shared throughput database. To resolve this issue, update the Microsoft.Azure.WebJobs.Extensions.CosmosDB extension to get the latest version.

Azure Function fails to start with "PartitionKey must be supplied for this operation."

This error means that you are currently using a partitioned lease collection with an old extension dependency. Upgrade to the latest available version. If you are currently running on Azure Functions V1, you will need to upgrade to Azure Functions V2.

Azure Function fails to start with "The lease collection, if partitioned, must have partition key equal to id."

This error means that your current leases container is partitioned, but the partition key path is not `/id`. To resolve this issue, you need to recreate the leases container with `/id` as the partition key.

You see a "Value cannot be null. Parameter name: o" in your Azure Functions logs when you try to Run the Trigger

This issue appears if you are using the Azure portal and you try to select the **Run** button on the screen when inspecting an Azure Function that uses the trigger. The trigger does not require for you to select Run to start, it will automatically start when the Azure Function is deployed. If you want to check the Azure Function's log stream on the Azure portal, just go to your monitored container and insert some new items, you will automatically see the Trigger executing.

My changes take too long to be received

This scenario can have multiple causes and all of them should be checked:

1. Is your Azure Function deployed in the same region as your Azure Cosmos account? For optimal network latency, both the Azure Function and your Azure Cosmos account should be colocated in the same Azure region.
2. Are the changes happening in your Azure Cosmos container continuous or sporadic? If it's the latter, there could be some delay between the changes being stored and the Azure Function picking them up. This is because internally, when the trigger checks for changes in your Azure Cosmos container and finds none pending to be read, it will sleep for a configurable amount of time (5 seconds, by default) before checking for new changes (to avoid high RU consumption). You can configure this sleep time through the `FeedPollDelay/feedPollDelay` setting in the configuration of your trigger (the value is expected to be in milliseconds).
3. Your Azure Cosmos container might be rate-limited.
4. You can use the `PreferredLocations` attribute in your trigger to specify a comma-separated list of Azure regions to define a custom preferred connection order.

Some changes are repeated in my Trigger

The concept of a "change" is an operation on a document. The most common scenarios where events for the same document is received are:

- The account is using Eventual consistency. While consuming the change feed in an Eventual consistency level, there could be duplicate events in-between subsequent change feed read operations (the last event of one read operation appears as the first of the next).
- The document is being updated. The Change Feed can contain multiple operations for the same documents, if that document is receiving updates, it can pick up multiple events (one for each update). One easy way to

distinguish among different operations for the same document is to track the `_lsn` property for each change. If they don't match, these are different changes over the same document.

- If you are identifying documents just by `id`, remember that the unique identifier for a document is the `id` and its partition key (there can be two documents with the same `id` but different partition key).

Some changes are missing in my Trigger

If you find that some of the changes that happened in your Azure Cosmos container are not being picked up by the Azure Function, there is an initial investigation step that needs to take place.

When your Azure Function receives the changes, it often processes them, and could optionally, send the result to another destination. When you are investigating missing changes, make sure you **measure which changes are being received at the ingestion point** (when the Azure Function starts), not on the destination.

If some changes are missing on the destination, this could mean that is some error happening during the Azure Function execution after the changes were received.

In this scenario, the best course of action is to add `try/catch` blocks in your code and inside the loops that might be processing the changes, to detect any failure for a particular subset of items and handle them accordingly (send them to another storage for further analysis or retry).

> **NOTE**
> The Azure Functions trigger for Cosmos DB, by default, won't retry a batch of changes if there was an unhandled exception during your code execution. This means that the reason that the changes did not arrive at the destination is because that you are failing to process them.

If you find that some changes were not received at all by your trigger, the most common scenario is that there is **another Azure Function running**. It could be another Azure Function deployed in Azure or an Azure Function running locally on a developer's machine that has **exactly the same configuration** (same monitored and lease containers), and this Azure Function is stealing a subset of the changes you would expect your Azure Function to process.

Additionally, the scenario can be validated, if you know how many Azure Function App instances you have running. If you inspect your leases container and count the number of lease items within, the distinct values of the `Owner` property in them should be equal to the number of instances of your Function App. If there are more owners than the known Azure Function App instances, it means that these extra owners are the ones "stealing" the changes.

One easy way to work around this situation, is to apply a `LeaseCollectionPrefix/leaseCollectionPrefix` to your Function with a new/different value or, alternatively, test with a new leases container.

Need to restart and reprocess all the items in my container from the beginning

To reprocess all the items in a container from the beginning:

1. Stop your Azure function if it is currently running.
2. Delete the documents in the lease collection (or delete and re-create the lease collection so it is empty)
3. Set the StartFromBeginning CosmosDBTrigger attribute in your function to true.
4. Restart the Azure function. It will now read and process all changes from the beginning.

Setting StartFromBeginning to true will tell the Azure function to start reading changes from the beginning of the history of the collection instead of the current time. This only works when there are no already created leases (that is, documents in the leases collection). Setting this property to true when there are leases already created has no effect; in this scenario, when a function is stopped and restarted, it will begin reading from the last checkpoint, as defined in the leases collection. To reprocess from the beginning, follow the above steps 1-4.

Binding can only be done with IReadOnlyList<Document> or JArray

This error happens if your Azure Functions project (or any referenced project) contains a manual NuGet reference to the Azure Cosmos DB SDK with a different version than the one provided by the Azure Functions Cosmos DB Extension.

To work around this situation, remove the manual NuGet reference that was added and let the Azure Cosmos DB SDK reference resolve through the Azure Functions Cosmos DB Extension package.

Changing Azure Function's polling interval for the detecting changes

As explained earlier for My changes take too long to be received, Azure function will sleep for a configurable amount of time (5 seconds, by default) before checking for new changes (to avoid high RU consumption). You can configure this sleep time through the `FeedPollDelay/feedPollDelay` setting in the configuration of your trigger (the value is expected to be in milliseconds).

Next steps

- Enable monitoring for your Azure Functions
- Azure Cosmos DB .NET SDK Troubleshooting

Use the change feed estimator

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: SQL API

This article describes how you can monitor the progress of your change feed processor instances as they read the change feed.

Why is monitoring progress important?

The change feed processor acts as a pointer that moves forward across your change feed and delivers the changes to a delegate implementation.

Your change feed processor deployment can process changes at a particular rate based on its available resources like CPU, memory, network, and so on.

If this rate is slower than the rate at which your changes happen in your Azure Cosmos container, your processor will start to lag behind.

Identifying this scenario helps understand if we need to scale our change feed processor deployment.

Implement the change feed estimator

Like the change feed processor, the change feed estimator works as a push model. The estimator will measure the difference between the last processed item (defined by the state of the leases container) and the latest change in the container, and push this value to a delegate. The interval at which the measurement is taken can also be customized with a default value of 5 seconds.

As an example, if your change feed processor is defined like this:

```
Container leaseContainer = client.GetContainer(databaseId, Program.leasesContainer);
Container monitoredContainer = client.GetContainer(databaseId, Program.monitoredContainer);
ChangeFeedProcessor changeFeedProcessor = monitoredContainer
    .GetChangeFeedProcessorBuilder<ToDoItem>("changeFeedEstimator", Program.HandleChangesAsync)
        .WithInstanceName("consoleHost")
        .WithLeaseContainer(leaseContainer)
        .Build();
```

The correct way to initialize an estimator to measure that processor would be using `GetChangeFeedEstimatorBuilder` like so:

```
ChangeFeedProcessor changeFeedEstimator = monitoredContainer
    .GetChangeFeedEstimatorBuilder("changeFeedEstimator", Program.HandleEstimationAsync,
TimeSpan.FromMilliseconds(1000))
    .WithLeaseContainer(leaseContainer)
    .Build();
```

Where both the processor and the estimator share the same `leaseContainer` and the same name.

The other two parameters are the delegate, which will receive a number that represents **how many changes are pending to be read** by the processor, and the time interval at which you want this measurement to be taken.

An example of a delegate that receives the estimation is:

```
static async Task HandleEstimationAsync(long estimation, CancellationToken cancellationToken)
{
    if (estimation > 0)
    {
        Console.WriteLine($"\tEstimator detected {estimation} items pending to be read by the Processor.");
    }

    await Task.Delay(0);
}
```

You can send this estimation to your monitoring solution and use it to understand how your progress is behaving over time.

> **NOTE**
>
> The change feed estimator does not need to be deployed as part of your change feed processor, nor be part of the same project. It can be independent and run in a completely different instance. It just needs to use the same name and lease configuration.

Additional resources

- Azure Cosmos DB SDK
- Usage samples on GitHub
- Additional samples on GitHub

Next steps

You can now proceed to learn more about change feed processor in the following articles:

- Overview of change feed processor
- Change feed processor start time

Migrate from the change feed processor library to the Azure Cosmos DB .NET V3 SDK

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: SQL API

This article describes the required steps to migrate an existing application's code that uses the change feed processor library to the change feed feature in the latest version of the .NET SDK (also referred as .NET V3 SDK).

Required code changes

The .NET V3 SDK has several breaking changes, the following are the key steps to migrate your application:

1. Convert the `DocumentCollectionInfo` instances into `Container` references for the monitored and leases containers.
2. Customizations that use `WithProcessorOptions` should be updated to use `WithLeaseConfiguration` and `WithPollInterval` for intervals, `WithStartTime` for start time, and `WithMaxItems` to define the maximum item count.
3. Set the `processorName` on `GetChangeFeedProcessorBuilder` to match the value configured on `ChangeFeedProcessorOptions.LeasePrefix`, or use `string.Empty` otherwise.
4. The changes are no longer delivered as a `IReadOnlyList<Document>`, instead, it's a `IReadOnlyCollection<T>` where `T` is a type you need to define, there is no base item class anymore.
5. To handle the changes, you no longer need an implementation, instead you need to define a delegate. The delegate can be a static Function or, if you need to maintain state across executions, you can create your own class and pass an instance method as delegate.

For example, if the original code to build the change feed processor looks as follows:

```
ChangeFeedProcessorLibrary.DocumentCollectionInfo monitoredCollectionInfo = new
ChangeFeedProcessorLibrary.DocumentCollectionInfo()
{
    DatabaseName = databaseId,
    CollectionName = Program.monitoredContainer,
    Uri = new Uri(configuration["EndPointUrl"]),
    MasterKey = configuration["AuthorizationKey"]
};

ChangeFeedProcessorLibrary.DocumentCollectionInfo leaseCollectionInfo = new
ChangeFeedProcessorLibrary.DocumentCollectionInfo()
{
    DatabaseName = databaseId,
    CollectionName = Program.leasesContainer,
    Uri = new Uri(configuration["EndPointUrl"]),
    MasterKey = configuration["AuthorizationKey"]
};

ChangeFeedProcessorLibrary.ChangeFeedProcessorBuilder builder = new
ChangeFeedProcessorLibrary.ChangeFeedProcessorBuilder();
var oldChangeFeedProcessor = await builder
    .WithHostName("consoleHost")
    .WithProcessorOptions(new ChangeFeedProcessorLibrary.ChangeFeedProcessorOptions
    {
        StartFromBeginning = true,
        LeasePrefix = "MyLeasePrefix",
        MaxItemCount = 10,
        FeedPollDelay = TimeSpan.FromSeconds(1)
    })
    .WithFeedCollection(monitoredCollectionInfo)
    .WithLeaseCollection(leaseCollectionInfo)
    .WithObserver<ChangeFeedObserver>()
    .BuildAsync();
```

The migrated code will look like:

```
Container leaseContainer = client.GetContainer(databaseId, Program.leasesContainer);
Container monitoredContainer = client.GetContainer(databaseId, Program.monitoredContainer);
ChangeFeedProcessor changeFeedProcessor = monitoredContainer
    .GetChangeFeedProcessorBuilder<ToDoItem>("MyLeasePrefix", Program.HandleChangesAsync)
        .WithInstanceName("consoleHost")
        .WithLeaseContainer(leaseContainer)
        .WithMaxItems(10)
        .WithPollInterval(TimeSpan.FromSeconds(1))
        .WithStartTime(DateTime.MinValue.ToUniversalTime())
        .Build();
```

And the delegate, can be a static method:

```
static async Task HandleChangesAsync(IReadOnlyCollection<ToDoItem> changes, CancellationToken
cancellationToken)
{
    foreach (ToDoItem item in changes)
    {
        Console.WriteLine($"\tDetected operation for item with id {item.id}, created at
{item.creationTime}.");
        // Simulate work
        await Task.Delay(1);
    }
}
```

State and lease container

Similar to the change feed processor library, the change feed feature in .NET V3 SDK uses a lease container to store the state. However, the schemas are different.

The SDK V3 change feed processor will detect any old library state and migrate it to the new schema automatically upon the first execution of the migrated application code.

You can safely stop the application using the old code, migrate the code to the new version, start the migrated application, and any changes that happened while the application was stopped, will be picked up and processed by the new version.

Additional resources

- Azure Cosmos DB SDK
- Usage samples on GitHub
- Additional samples on GitHub

Next steps

You can now proceed to learn more about change feed processor in the following articles:

- Overview of change feed processor
- Using the change feed estimator
- Change feed processor start time

How to create a Java application that uses Azure Cosmos DB SQL API and change feed processor

11/2/2020 • 5 minutes to read • Edit Online

APPLIES TO: SQL API

This how-to guide walks you through a simple Java application which uses the Azure Cosmos DB SQL API to insert documents into an Azure Cosmos DB container, while maintaining a materialized view of the container using Change Feed and Change Feed Processor. The Java application communicates with the Azure Cosmos DB SQL API using Azure Cosmos DB Java SDK v4.

> **IMPORTANT**
>
> This tutorial is for Azure Cosmos DB Java SDK v4 only. Please view the Azure Cosmos DB Java SDK v4 Release notes, Maven repository, Azure Cosmos DB Java SDK v4 performance tips, and Azure Cosmos DB Java SDK v4 troubleshooting guide for more information. If you are currently using an older version than v4, see the Migrate to Azure Cosmos DB Java SDK v4 guide for help upgrading to v4.

Prerequisites

- The URI and key for your Azure Cosmos DB account
- Maven
- Java 8

Background

The Azure Cosmos DB change feed provides an event-driven interface to trigger actions in response to document insertion. This has many uses. For example in applications which are both read and write heavy, a chief use of change feed is to create a real-time **materialized view** of a container as it is ingesting documents. The materialized view container will hold the same data but partitioned for efficient reads, making the application both read and write efficient.

The work of managing change feed events is largely taken care of by the change feed Processor library built into the SDK. This library is powerful enough to distribute change feed events among multiple workers, if that is desired. All you have to do is provide the change feed library a callback.

This simple example demonstrates change feed Processor library with a single worker creating and deleting documents from a materialized view.

Setup

If you have not already done so, clone the app example repo:

```
git clone https://github.com/Azure-Samples/azure-cosmos-java-sql-app-example.git
```

Open a terminal in the repo directory. Build the app by running

```
mvn clean package
```

Walkthrough

1. As a first check, you should have an Azure Cosmos DB account. Open the **Azure portal** in your browser, go to your Azure Cosmos DB account, and in the left pane navigate to **Data Explorer**.

2. Run the app in the terminal using the following command:

```
mvn exec:java -Dexec.mainClass="com.azure.cosmos.workedappexample.SampleGroceryStore" -
DACCOUNT_HOST="your-account-uri" -DACCOUNT_KEY="your-account-key" -Dexec.cleanupDaemonThreads=false
```

3. Press enter when you see

```
Press enter to create the grocery store inventory system...
```

then return to the Azure portal Data Explorer in your browser. You will see a database **GroceryStoreDatabase** has been added with three empty containers:

- **InventoryContainer** - The inventory record for our example grocery store, partitioned on item `id` which is a UUID.
- **InventoryContainer-pktype** - A materialized view of the inventory record, optimized for queries over item `type`
- **InventoryContainer-leases** - A leases container is always needed for change feed; leases track the app's progress in reading the change feed.

4. In the terminal, you should now see a prompt

```
Press enter to start creating the materialized view...
```

Press enter. Now the following block of code will execute and initialize the change feed processor on another thread:

Java SDK V4 (Maven com.azure::azure-cosmos) Async API

```
changeFeedProcessorInstance = getChangeFeedProcessor("SampleHost_1", feedContainer, leaseContainer);
changeFeedProcessorInstance.start()
    .subscribeOn(Schedulers.elastic())
    .doOnSuccess(aVoid -> {
        isProcessorRunning.set(true);
    })
    .subscribe();

while (!isProcessorRunning.get()); //Wait for Change Feed processor start
```

`"SampleHost_1"` is the name of the Change Feed processor worker. `changeFeedProcessorInstance.start()` is what actually starts the Change Feed processor.

Return to the Azure portal Data Explorer in your browser. Under the **InventoryContainer-leases** container, click **items** to see its contents. You will see that Change Feed Processor has populated the lease container, i.e. the processor has assigned the `SampleHost_1` worker a lease on some partitions of the **InventoryContainer**.

5. Press enter again in the terminal. This will trigger 10 documents to be inserted into **InventoryContainer**. Each document insertion appears in the change feed as JSON; the following callback code handles these events by mirroring the JSON documents into a materialized view:

Java SDK V4 (Maven com.azure::azure-cosmos) Async API

```
public static ChangeFeedProcessor getChangeFeedProcessor(String hostName, CosmosAsyncContainer
feedContainer, CosmosAsyncContainer leaseContainer) {
    ChangeFeedProcessorOptions cfOptions = new ChangeFeedProcessorOptions();
    cfOptions.setFeedPollDelay(Duration.ofMillis(100));
    cfOptions.setStartFromBeginning(true);
    return new ChangeFeedProcessorBuilder()
        .options(cfOptions)
        .hostName(hostName)
        .feedContainer(feedContainer)
        .leaseContainer(leaseContainer)
        .handleChanges((List<JsonNode> docs) -> {
            for (JsonNode document : docs) {
                    //Duplicate each document update from the feed container into the materialized view
container
                    updateInventoryTypeMaterializedView(document);
            }

        })
        .buildChangeFeedProcessor();
}

private static void updateInventoryTypeMaterializedView(JsonNode document) {
    typeContainer.upsertItem(document).subscribe();
}
```

6. Allow the code to run 5-10sec. Then return to the Azure portal Data Explorer and navigate to **InventoryContainer > items**. You should see that items are being inserted into the inventory container; note the partition key (`id`).

7. Now, in Data Explorer navigate to **InventoryContainer-pktype > items**. This is the materialized view - the items in this container mirror **InventoryContainer** because they were inserted programmatically by change feed. Note the partition key (`type`). So this materialized view is optimized for queries filtering over `type` , which would be inefficient on **InventoryContainer** because it is partitioned on `id` .

8. We're going to delete a document from both **InventoryContainer** and **InventoryContainer-pktype** using just a single `upsertItem()` call. First, take a look at Azure portal Data Explorer. We'll delete the document for which `/type == "plums"` ; it is encircled in red below

Hit enter again to call the function `deleteDocument()` in the example code. This function, shown below, upserts a new version of the document with `/ttl == 5`, which sets document Time-To-Live (TTL) to 5sec.

Java SDK V4 (Maven com.azure::azure-cosmos) Async API

```
public static void deleteDocument() {

    String jsonString =    "{\"id\" : \"" + idToDelete + "\""
            + ","
            + "\"brand\" : \"Jerry's\""
            + ","
            + "\"type\" : \"plums\""
            + ","
            + "\"quantity\" : \"50\""
            + ","
            + "\"ttl\" : 5"
            + "}";

    ObjectMapper mapper = new ObjectMapper();
    JsonNode document = null;

    try {
        document = mapper.readTree(jsonString);
    } catch (Exception e) {
        e.printStackTrace();
    }

    feedContainer.upsertItem(document,new CosmosItemRequestOptions()).block();
}
```

The change feed `feedPollDelay` is set to 100ms; therefore, change feed responds to this update almost instantly and calls `updateInventoryTypeMaterializedView()` shown above. That last function call will upsert the new document with TTL of 5sec into **InventoryContainer-pktype**.

The effect is that after about 5 seconds, the document will expire and be deleted from both containers.

This procedure is necessary because change feed only issues events on item insertion or update, not on item deletion.

9. Press enter one more time to close the program and clean up its resources.

How to write stored procedures, triggers, and user-defined functions in Azure Cosmos DB

12/4/2020 • 11 minutes to read • Edit Online

APPLIES TO: SQL API

Azure Cosmos DB provides language-integrated, transactional execution of JavaScript that lets you write **stored procedures**, **triggers**, and **user-defined functions (UDFs)**. When using the SQL API in Azure Cosmos DB, you can define the stored procedures, triggers, and UDFs in JavaScript language. You can write your logic in JavaScript and execute it inside the database engine. You can create and execute triggers, stored procedures, and UDFs by using Azure portal, the JavaScript language integrated query API in Azure Cosmos DB and the Cosmos DB SQL API client SDKs.

To call a stored procedure, trigger, and user-defined function, you need to register it. For more information, see How to work with stored procedures, triggers, user-defined functions in Azure Cosmos DB.

> **NOTE**
>
> For partitioned containers, when executing a stored procedure, a partition key value must be provided in the request options. Stored procedures are always scoped to a partition key. Items that have a different partition key value will not be visible to the stored procedure. This also applied to triggers as well.

> **TIP**
>
> Cosmos supports deploying containers with stored procedures, triggers and user-defined functions. For more information see Create an Azure Cosmos DB container with server-side functionality.

How to write stored procedures

Stored procedures are written using JavaScript, they can create, update, read, query, and delete items inside an Azure Cosmos container. Stored procedures are registered per collection, and can operate on any document or an attachment present in that collection.

Here is a simple stored procedure that returns a "Hello World" response.

```
var helloWorldStoredProc = {
    id: "helloWorld",
    serverScript: function () {
        var context = getContext();
        var response = context.getResponse();

        response.setBody("Hello, World");
    }
}
```

The context object provides access to all operations that can be performed in Azure Cosmos DB, as well as access to the request and response objects. In this case, you use the response object to set the body of the response to be sent back to the client.

Once written, the stored procedure must be registered with a collection. To learn more, see How to use stored

procedures in Azure Cosmos DB article.

Create an item using stored procedure

When you create an item by using stored procedure, the item is inserted into the Azure Cosmos container and an ID for the newly created item is returned. Creating an item is an asynchronous operation and depends on the JavaScript callback functions. The callback function has two parameters - one for the error object in case the operation fails and another for a return value; in this case, the created object. Inside the callback, you can either handle the exception or throw an error. In case a callback is not provided and there is an error, the Azure Cosmos DB runtime will throw an error.

The stored procedure also includes a parameter to set the description, it's a boolean value. When the parameter is set to true and the description is missing, the stored procedure will throw an exception. Otherwise, the rest of the stored procedure continues to run.

The following example stored procedure takes an array of new Azure Cosmos items as input, inserts it into the Azure Cosmos container and returns the count of the items inserted. In this example, we are leveraging the ToDoList sample from the Quickstart .NET SQL API

```
function createToDoItems(items) {
    var collection = getContext().getCollection();
    var collectionLink = collection.getSelfLink();
    var count = 0;

    if (!items) throw new Error("The array is undefined or null.");

    var numItems = items.length;

    if (numItems == 0) {
        getContext().getResponse().setBody(0);
        return;
    }

    tryCreate(items[count], callback);

    function tryCreate(item, callback) {
        var options = { disableAutomaticIdGeneration: false };

        var isAccepted = collection.createDocument(collectionLink, item, options, callback);

        if (!isAccepted) getContext().getResponse().setBody(count);
    }

    function callback(err, item, options) {
        if (err) throw err;
        count++;
        if (count >= numItems) {
            getContext().getResponse().setBody(count);
        } else {
            tryCreate(items[count], callback);
        }
    }
}
```

Arrays as input parameters for stored procedures

When defining a stored procedure in Azure portal, input parameters are always sent as a string to the stored procedure. Even if you pass an array of strings as an input, the array is converted to string and sent to the stored procedure. To work around this, you can define a function within your stored procedure to parse the string as an array. The following code shows how to parse a string input parameter as an array:

```
function sample(arr) {
    if (typeof arr === "string") arr = JSON.parse(arr);

    arr.forEach(function(a) {
        // do something here
        console.log(a);
    });
}
```

Transactions within stored procedures

You can implement transactions on items within a container by using a stored procedure. The following example uses transactions within a fantasy football gaming app to trade players between two teams in a single operation. The stored procedure attempts to read the two Azure Cosmos items each corresponding to the player IDs passed in as an argument. If both players are found, then the stored procedure updates the items by swapping their teams. If any errors are encountered along the way, the stored procedure throws a JavaScript exception that implicitly aborts the transaction.

```
// JavaScript source code
function tradePlayers(playerId1, playerId2) {
    var context = getContext();
    var container = context.getCollection();
    var response = context.getResponse();

    var player1Document, player2Document;

    // query for players
    var filterQuery =
    {
        'query' : 'SELECT * FROM Players p where p.id = @playerId1',
        'parameters' : [{'name':'@playerId1', 'value':playerId1}]
    };

    var accept = container.queryDocuments(container.getSelfLink(), filterQuery, {},
        function (err, items, responseOptions) {
            if (err) throw new Error("Error" + err.message);

            if (items.length != 1) throw "Unable to find both names";
            player1Item = items[0];

            var filterQuery2 =
            {
                'query' : 'SELECT * FROM Players p where p.id = @playerId2',
                'parameters' : [{'name':'@playerId2', 'value':playerId2}]
            };
            var accept2 = container.queryDocuments(container.getSelfLink(), filterQuery2, {},
                function (err2, items2, responseOptions2) {
                    if (err2) throw new Error("Error" + err2.message);
                    if (items2.length != 1) throw "Unable to find both names";
                    player2Item = items2[0];
                    swapTeams(player1Item, player2Item);
                    return;
                });
            if (!accept2) throw "Unable to read player details, abort ";
        });

    if (!accept) throw "Unable to read player details, abort ";

    // swap the two players' teams
    function swapTeams(player1, player2) {
        var player2NewTeam = player1.team;
        player1.team = player2.team;
        player2.team = player2NewTeam;

        var accept = container.replaceDocument(player1._self, player1,
            function (err, itemReplaced) {
                if (err) throw "Unable to update player 1, abort ";

                var accept2 = container.replaceDocument(player2._self, player2,
                    function (err2, itemReplaced2) {
                        if (err) throw "Unable to update player 2, abort"
                    });

                if (!accept2) throw "Unable to update player 2, abort";
            });

        if (!accept) throw "Unable to update player 1, abort";
    }
}
```

Bounded execution within stored procedures

The following is an example of a stored procedure that bulk-imports items into an Azure Cosmos container. The stored procedure handles bounded execution by checking the boolean return value from `createDocument`, and then

uses the count of items inserted in each invocation of the stored procedure to track and resume progress across batches.

```
function bulkImport(items) {
    var container = getContext().getCollection();
    var containerLink = container.getSelfLink();

    // The count of imported items, also used as current item index.
    var count = 0;

    // Validate input.
    if (!items) throw new Error("The array is undefined or null.");

    var itemsLength = items.length;
    if (itemsLength == 0) {
        getContext().getResponse().setBody(0);
    }

    // Call the create API to create an item.
    tryCreate(items[count], callback);

    // Note that there are 2 exit conditions:
    // 1) The createDocument request was not accepted.
    //    In this case the callback will not be called, we just call setBody and we are done.
    // 2) The callback was called items.length times.
    //    In this case all items were created and we don't need to call tryCreate anymore. Just call setBody
and we are done.
    function tryCreate(item, callback) {
        var isAccepted = container.createDocument(containerLink, item, callback);

        // If the request was accepted, callback will be called.
        // Otherwise report current count back to the client,
        // which will call the script again with remaining set of items.
        if (!isAccepted) getContext().getResponse().setBody(count);
    }

    // This is called when container.createDocument is done in order to process the result.
    function callback(err, item, options) {
        if (err) throw err;

        // One more item has been inserted, increment the count.
        count++;

        if (count >= itemsLength) {
            // If we created all items, we are done. Just set the response.
            getContext().getResponse().setBody(count);
        } else {
            // Create next document.
            tryCreate(items[count], callback);
        }
    }
}
```

Async await with stored procedures

The following is an example of a stored procedure that uses async-await with Promises using a helper function. The stored procedure queries for an item and replaces it.

```
function async_sample() {
    const ERROR_CODE = {
        NotAccepted: 429
    };

    const asyncHelper = {
        queryDocuments(sqlQuery, options) {
            return new Promise((resolve, reject) => {
                const isAccepted = __.queryDocuments(__.getSelfLink(), sqlQuery, options, (err, feed, options)
=> {
                    if (err) reject(err);
                    resolve({ feed, options });
                });
                if (!isAccepted) reject(new Error(ERROR_CODE.NotAccepted, "replaceDocument was not
accepted."));
            });
        },

        replaceDocument(doc) {
            return new Promise((resolve, reject) => {
                const isAccepted = __.replaceDocument(doc._self, doc, (err, result, options) => {
                    if (err) reject(err);
                    resolve({ result, options });
                });
                if (!isAccepted) reject(new Error(ERROR_CODE.NotAccepted, "replaceDocument was not
accepted."));
            });
        }
    };

    async function main() {
        let continuation;
        do {
            let { feed, options } = await asyncHelper.queryDocuments("SELECT * from c", { continuation });

            for (let doc of feed) {
                doc.newProp = 1;
                await asyncHelper.replaceDocument(doc);
            }

            continuation = options.continuation;
        } while (continuation);
    }

    main().catch(err => getContext().abort(err));
}
```

How to write triggers

Azure Cosmos DB supports pre-triggers and post-triggers. Pre-triggers are executed before modifying a database item and post-triggers are executed after modifying a database item.

Pre-triggers

The following example shows how a pre-trigger is used to validate the properties of an Azure Cosmos item that is being created. In this example, we are leveraging the ToDoList sample from the Quickstart .NET SQL API, to add a timestamp property to a newly added item if it doesn't contain one.

```
function validateToDoItemTimestamp() {
    var context = getContext();
    var request = context.getRequest();

    // item to be created in the current operation
    var itemToCreate = request.getBody();

    // validate properties
    if (!("timestamp" in itemToCreate)) {
        var ts = new Date();
        itemToCreate["timestamp"] = ts.getTime();
    }

    // update the item that will be created
    request.setBody(itemToCreate);
}
```

Pre-triggers cannot have any input parameters. The request object in the trigger is used to manipulate the request message associated with the operation. In the previous example, the pre-trigger is run when creating an Azure Cosmos item, and the request message body contains the item to be created in JSON format.

When triggers are registered, you can specify the operations that it can run with. This trigger should be created with a `TriggerOperation` value of `TriggerOperation.Create`, which means using the trigger in a replace operation as shown in the following code is not permitted.

For examples of how to register and call a pre-trigger, see pre-triggers and post-triggers articles.

Post-triggers

The following example shows a post-trigger. This trigger queries for the metadata item and updates it with details about the newly created item.

```
function updateMetadata() {
var context = getContext();
var container = context.getCollection();
var response = context.getResponse();

// item that was created
var createdItem = response.getBody();

// query for metadata document
var filterQuery = 'SELECT * FROM root r WHERE r.id = "_metadata"';
var accept = container.queryDocuments(container.getSelfLink(), filterQuery,
    updateMetadataCallback);
if(!accept) throw "Unable to update metadata, abort";

function updateMetadataCallback(err, items, responseOptions) {
    if(err) throw new Error("Error" + err.message);
        if(items.length != 1) throw 'Unable to find metadata document';

        var metadataItem = items[0];

        // update metadata
        metadataItem.createdItems += 1;
        metadataItem.createdNames += " " + createdItem.id;
        var accept = container.replaceDocument(metadataItem._self,
            metadataItem, function(err, itemReplaced) {
                    if(err) throw "Unable to update metadata, abort";
            });
        if(!accept) throw "Unable to update metadata, abort";
        return;
}
```

One thing that is important to note is the transactional execution of triggers in Azure Cosmos DB. The post-trigger runs as part of the same transaction for the underlying item itself. An exception during the post-trigger execution will fail the whole transaction. Anything committed will be rolled back and an exception returned.

For examples of how to register and call a pre-trigger, see pre-triggers and post-triggers articles.

How to write user-defined functions

The following sample creates a UDF to calculate income tax for various income brackets. This user-defined function would then be used inside a query. For the purposes of this example assume there is a container called "Incomes" with properties as follows:

```
{
   "name": "Satya Nadella",
   "country": "USA",
   "income": 70000
}
```

The following is a function definition to calculate income tax for various income brackets:

```
function tax(income) {

        if(income == undefined)
            throw 'no input';

        if (income < 1000)
            return income * 0.1;
        else if (income < 10000)
            return income * 0.2;
        else
            return income * 0.4;
    }
```

For examples of how to register and use a user-defined function, see How to use user-defined functions in Azure Cosmos DB article.

Logging

When using stored procedure, triggers or user-defined functions, you can log the steps using the `console.log()` command. This command will concentrate a string for debugging when `EnableScriptLogging` is set to true as shown in the following example:

```
var response = await client.ExecuteStoredProcedureAsync(
document.SelfLink,
new RequestOptions { EnableScriptLogging = true } );
Console.WriteLine(response.ScriptLog);
```

Next steps

Learn more concepts and how-to write or use stored procedures, triggers, and user-defined functions in Azure Cosmos DB:

- How to register and use stored procedures, triggers, and user-defined functions in Azure Cosmos DB
- How to write stored procedures and triggers using Javascript Query API in Azure Cosmos DB
- Working with Azure Cosmos DB stored procedures, triggers, and user-defined functions in Azure Cosmos

DB

- Working with JavaScript language integrated query API in Azure Cosmos DB

How to write stored procedures and triggers in Azure Cosmos DB by using the JavaScript query API

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: SQL API

Azure Cosmos DB allows you to perform optimized queries by using a fluent JavaScript interface without any knowledge of SQL language that can be used to write stored procedures or triggers. To learn more about JavaScript Query API support in Azure Cosmos DB, see Working with JavaScript language integrated query API in Azure Cosmos DB article.

Stored procedure using the JavaScript query API

The following code sample is an example of how the JavaScript query API is used in the context of a stored procedure. The stored procedure inserts an Azure Cosmos item that is specified by an input parameter, and updates a metadata document by using the `__.filter()` method, with minSize, maxSize, and totalSize based upon the input item's size property.

> **NOTE**
>
> `__` (double-underscore) is an alias to `getContext().getCollection()` when using the JavaScript query API.

```
/**
 * Insert an item and update metadata doc: minSize, maxSize, totalSize based on item.size.
 */
function insertDocumentAndUpdateMetadata(item) {
  // HTTP error codes sent to our callback function by CosmosDB server.
  var ErrorCode = {
    RETRY_WITH: 449,
  }

  var isAccepted = __.createDocument(__.getSelfLink(), item, {}, function(err, item, options) {
    if (err) throw err;

    // Check the item (ignore items with invalid/zero size and metadata itself) and call updateMetadata.
    if (!item.isMetadata && item.size > 0) {
      // Get the metadata. We keep it in the same container. it's the only item that has .isMetadata = true.
      var result = __.filter(function(x) {
        return x.isMetadata === true
      }, function(err, feed, options) {
        if (err) throw err;

        // We assume that metadata item was pre-created and must exist when this script is called.
        if (!feed || !feed.length) throw new Error("Failed to find the metadata item.");

        // The metadata item.
        var metaItem = feed[0];

        // Update metaDoc.minSize:
        // for 1st document use doc.Size, for all the rest see if it's less than last min.
        if (metaItem.minSize == 0) metaItem.minSize = item.size;
        else metaItem.minSize = Math.min(metaItem.minSize, item.size);

        // Update metaItem.maxSize.
        metaItem.maxSize = Math.max(metaItem.maxSize, item.size);

        // Update metaItem.totalSize.
        metaItem.totalSize += item.size;

        // Update/replace the metadata item in the store.
        var isAccepted = __.replaceDocument(metaItem._self, metaItem, function(err) {
          if (err) throw err;
          // Note: in case concurrent updates causes conflict with ErrorCode.RETRY_WITH, we can't read the
meta again
          //       and update again because due to Snapshot isolation we will read same exact version (we are
in same transaction).
          //       We have to take care of that on the client side.
        });
        if (!isAccepted) throw new Error("replaceDocument(metaItem) returned false.");
      });
      if (!result.isAccepted) throw new Error("filter for metaItem returned false.");
    }
  });
  if (!isAccepted) throw new Error("createDocument(actual item) returned false.");
}
```

Next steps

See the following articles to learn about stored procedures, triggers, and user-defined functions in Azure Cosmos DB:

- How to work with stored procedures, triggers, user-defined functions in Azure Cosmos DB
- How to register and use stored procedures in Azure Cosmos DB
- How to register and use pre-triggers and post-triggers in Azure Cosmos DB

- How to register and use user-defined functions in Azure Cosmos DB
- Synthetic partition keys in Azure Cosmos DB

How to register and use stored procedures, triggers, and user-defined functions in Azure Cosmos DB

12/4/2020 • 12 minutes to read • Edit Online

APPLIES TO: SQL API

The SQL API in Azure Cosmos DB supports registering and invoking stored procedures, triggers, and user-defined functions (UDFs) written in JavaScript. You can use the SQL API .NET, .NET Core, Java, JavaScript, Node.js, or Python SDKs to register and invoke the stored procedures. Once you have defined one or more stored procedures, triggers, and user-defined functions, you can load and view them in the Azure portal by using Data Explorer.

How to run stored procedures

Stored procedures are written using JavaScript. They can create, update, read, query, and delete items within an Azure Cosmos container. For more information on how to write stored procedures in Azure Cosmos DB, see How to write stored procedures in Azure Cosmos DB article.

The following examples show how to register and call a stored procedure by using the Azure Cosmos DB SDKs. Refer to Create a Document as the source for this stored procedure is saved as `spCreateToDoItem.js`.

> **NOTE**
> For partitioned containers, when executing a stored procedure, a partition key value must be provided in the request options. Stored procedures are always scoped to a partition key. Items that have a different partition key value will not be visible to the stored procedure. This also applied to triggers as well.

Stored procedures - .NET SDK V2

The following example shows how to register a stored procedure by using the .NET SDK V2:

```
string storedProcedureId = "spCreateToDoItems";
StoredProcedure newStoredProcedure = new StoredProcedure
   {
       Id = storedProcedureId,
       Body = File.ReadAllText($@"..\js\{storedProcedureId}.js")
   };
Uri containerUri = UriFactory.CreateDocumentCollectionUri("myDatabase", "myContainer");
var response = await client.CreateStoredProcedureAsync(containerUri, newStoredProcedure);
StoredProcedure createdStoredProcedure = response.Resource;
```

The following code shows how to call a stored procedure by using the .NET SDK V2:

```
dynamic[] newItems = new dynamic[]
{
    new {
        category = "Personal",
        name = "Groceries",
        description = "Pick up strawberries",
        isComplete = false
    },
    new {
        category = "Personal",
        name = "Doctor",
        description = "Make appointment for check up",
        isComplete = false
    }
};

Uri uri = UriFactory.CreateStoredProcedureUri("myDatabase", "myContainer", "spCreateToDoItem");
RequestOptions options = new RequestOptions { PartitionKey = new PartitionKey("Personal") };
var result = await client.ExecuteStoredProcedureAsync<string>(uri, options, new[] { newItems });
```

Stored procedures - .NET SDK V3

The following example shows how to register a stored procedure by using the .NET SDK V3:

```
string storedProcedureId = "spCreateToDoItems";
StoredProcedureResponse storedProcedureResponse = await client.GetContainer("myDatabase",
"myContainer").Scripts.CreateStoredProcedureAsync(new StoredProcedureProperties
{
    Id = storedProcedureId,
    Body = File.ReadAllText($@"..\js\{storedProcedureId}.js")
});
```

The following code shows how to call a stored procedure by using the .NET SDK V3:

```
dynamic[] newItems = new dynamic[]
{
    new {
        category = "Personal",
        name = "Groceries",
        description = "Pick up strawberries",
        isComplete = false
    },
    new {
        category = "Personal",
        name = "Doctor",
        description = "Make appointment for check up",
        isComplete = false
    }
};

var result = await client.GetContainer("database", "container").Scripts.ExecuteStoredProcedureAsync<string>
("spCreateToDoItem", new PartitionKey("Personal"), new[] { newItems });
```

Stored procedures - Java SDK

The following example shows how to register a stored procedure by using the Java SDK:

```
String containerLink = String.format("/dbs/%s/colls/%s", "myDatabase", "myContainer");
StoredProcedure newStoredProcedure = new StoredProcedure(
    "{" +
        "  'id':'spCreateToDoItems'," +
        "  'body':" + new String(Files.readAllBytes(Paths.get("..\\js\\spCreateToDoItems.js"))) +
    "}");
//toBlocking() blocks the thread until the operation is complete and is used only for demo.
StoredProcedure createdStoredProcedure = asyncClient.createStoredProcedure(containerLink, newStoredProcedure,
null)
    .toBlocking().single().getResource();
```

The following code shows how to call a stored procedure by using the Java SDK:

```
String containerLink = String.format("/dbs/%s/colls/%s", "myDatabase", "myContainer");
String sprocLink = String.format("%s/sprocs/%s", containerLink, "spCreateToDoItems");
final CountDownLatch successfulCompletionLatch = new CountDownLatch(1);

List<ToDoItem> ToDoItems = new ArrayList<ToDoItem>();

class ToDoItem {
    public String category;
    public String name;
    public String description;
    public boolean isComplete;
}

ToDoItem newItem = new ToDoItem();
newItem.category = "Personal";
newItem.name = "Groceries";
newItem.description = "Pick up strawberries";
newItem.isComplete = false;

ToDoItems.add(newItem)

newItem.category = "Personal";
newItem.name = "Doctor";
newItem.description = "Make appointment for check up";
newItem.isComplete = false;

ToDoItems.add(newItem)

RequestOptions requestOptions = new RequestOptions();
requestOptions.setPartitionKey(new PartitionKey("Personal"));

Object[] storedProcedureArgs = new Object[] { ToDoItems };
asyncClient.executeStoredProcedure(sprocLink, requestOptions, storedProcedureArgs)
    .subscribe(storedProcedureResponse -> {
        String storedProcResultAsString = storedProcedureResponse.getResponseAsString();
        successfulCompletionLatch.countDown();
        System.out.println(storedProcedureResponse.getActivityId());
    }, error -> {
        successfulCompletionLatch.countDown();
        System.err.println("an error occurred while executing the stored procedure: actual cause: "
                + error.getMessage());
    });

successfulCompletionLatch.await();
```

Stored procedures - JavaScript SDK

The following example shows how to register a stored procedure by using the JavaScript SDK

```
const container = client.database("myDatabase").container("myContainer");
const sprocId = "spCreateToDoItems";
await container.scripts.storedProcedures.create({
    id: sprocId,
    body: require(`../js/${sprocId}`)
});
```

The following code shows how to call a stored procedure by using the JavaScript SDK:

```
const newItem = [{
    category: "Personal",
    name: "Groceries",
    description: "Pick up strawberries",
    isComplete: false
}];
const container = client.database("myDatabase").container("myContainer");
const sprocId = "spCreateToDoItems";
const {body: result} = await container.scripts.storedProcedure(sprocId).execute(newItem, {partitionKey:
newItem[0].category});
```

Stored procedures - Python SDK

The following example shows how to register a stored procedure by using the Python SDK:

```
import azure.cosmos.cosmos_client as cosmos_client

url = "your_cosmos_db_account_URI"
key = "your_cosmos_db_account_key"
database_name = 'your_cosmos_db_database_name'
container_name = 'your_cosmos_db_container_name'

with open('../js/spCreateToDoItems.js') as file:
    file_contents = file.read()

sproc = {
    'id': 'spCreateToDoItem',
    'serverScript': file_contents,
}
client = cosmos_client.CosmosClient(url, key)
database = client.get_database_client(database_name)
container = database.get_container_client(container_name)
created_sproc = container.scripts.create_stored_procedure(body=sproc)
```

The following code shows how to call a stored procedure by using the Python SDK:

```
import uuid

new_id= str(uuid.uuid4())

# Creating a document for a container with "id" as a partition key.

new_item =   {
      "id": new_id,
      "category":"Personal",
      "name":"Groceries",
      "description":"Pick up strawberries",
      "isComplete":False
   }
result = container.scripts.execute_stored_procedure(sproc=created_sproc,params=[[new_item]],
partition_key=new_id)
```

How to run pre-triggers

The following examples show how to register and call a pre-trigger by using the Azure Cosmos DB SDKs. Refer to the Pre-trigger example as the source for this pre-trigger is saved as `trgPreValidateToDoItemTimestamp.js`.

When executing, pre-triggers are passed in the RequestOptions object by specifying `PreTriggerInclude` and then passing the name of the trigger in a List object.

> **NOTE**
> Even though the name of the trigger is passed as a List, you can still execute only one trigger per operation.

Pre-triggers - .NET SDK V2

The following code shows how to register a pre-trigger using the .NET SDK V2:

```
string triggerId = "trgPreValidateToDoItemTimestamp";
Trigger trigger = new Trigger
{
    Id =  triggerId,
    Body = File.ReadAllText($@"..\js\{triggerId}.js"),
    TriggerOperation = TriggerOperation.Create,
    TriggerType = TriggerType.Pre
};
Uri containerUri = UriFactory.CreateDocumentCollectionUri("myDatabase", "myContainer");
await client.CreateTriggerAsync(containerUri, trigger);
```

The following code shows how to call a pre-trigger using the .NET SDK V2:

```
dynamic newItem = new
{
    category = "Personal",
    name = "Groceries",
    description = "Pick up strawberries",
    isComplete = false
};

Uri containerUri = UriFactory.CreateDocumentCollectionUri("myDatabase", "myContainer");
RequestOptions requestOptions = new RequestOptions { PreTriggerInclude = new List<string> {
"trgPreValidateToDoItemTimestamp" } };
await client.CreateDocumentAsync(containerUri, newItem, requestOptions);
```

Pre-triggers - .NET SDK V3

The following code shows how to register a pre-trigger using the .NET SDK V3:

```
await client.GetContainer("database", "container").Scripts.CreateTriggerAsync(new TriggerProperties
{
    Id = "trgPreValidateToDoItemTimestamp",
    Body = File.ReadAllText("@..\js\trgPreValidateToDoItemTimestamp.js"),
    TriggerOperation = TriggerOperation.Create,
    TriggerType = TriggerType.Pre
});
```

The following code shows how to call a pre-trigger using the .NET SDK V3:

```
dynamic newItem = new
{
    category = "Personal",
    name = "Groceries",
    description = "Pick up strawberries",
    isComplete = false
};

await client.GetContainer("database", "container").CreateItemAsync(newItem, null, new ItemRequestOptions {
PreTriggers = new List<string> { "trgPreValidateToDoItemTimestamp" } });
```

Pre-triggers - Java SDK

The following code shows how to register a pre-trigger using the Java SDK:

```
String containerLink = String.format("/dbs/%s/colls/%s", "myDatabase", "myContainer");
String triggerId = "trgPreValidateToDoItemTimestamp";
Trigger trigger = new Trigger();
trigger.setId(triggerId);
trigger.setBody(new String(Files.readAllBytes(Paths.get(String.format("..\\js\\%s.js", triggerId)));
trigger.setTriggerOperation(TriggerOperation.Create);
trigger.setTriggerType(TriggerType.Pre);
//toBlocking() blocks the thread until the operation is complete and is used only for demo.
Trigger createdTrigger = asyncClient.createTrigger(containerLink, trigger, new
RequestOptions()).toBlocking().single().getResource();
```

The following code shows how to call a pre-trigger using the Java SDK:

```
String containerLink = String.format("/dbs/%s/colls/%s", "myDatabase", "myContainer");
    Document item = new Document("{ "
            + "\"category\": \"Personal\", "
            + "\"name\": \"Groceries\", "
            + "\"description\": \"Pick up strawberries\", "
            + "\"isComplete\": false, "
            + "}"
            );
RequestOptions requestOptions = new RequestOptions();
requestOptions.setPreTriggerInclude(Arrays.asList("trgPreValidateToDoItemTimestamp"));
//toBlocking() blocks the thread until the operation is complete and is used only for demo.
asyncClient.createDocument(containerLink, item, requestOptions, false).toBlocking();
```

Pre-triggers - JavaScript SDK

The following code shows how to register a pre-trigger using the JavaScript SDK:

```
const container = client.database("myDatabase").container("myContainer");
const triggerId = "trgPreValidateToDoItemTimestamp";
await container.triggers.create({
    id: triggerId,
    body: require(`../js/${triggerId}`),
    triggerOperation: "create",
    triggerType: "pre"
});
```

The following code shows how to call a pre-trigger using the JavaScript SDK:

```
const container = client.database("myDatabase").container("myContainer");
const triggerId = "trgPreValidateToDoItemTimestamp";
await container.items.create({
    category: "Personal",
    name = "Groceries",
    description = "Pick up strawberries",
    isComplete = false
}, {preTriggerInclude: [triggerId]});
```

Pre-triggers - Python SDK

The following code shows how to register a pre-trigger using the Python SDK:

```
import azure.cosmos.cosmos_client as cosmos_client

url = "your_cosmos_db_account_URI"
key = "your_cosmos_db_account_key"
database_name = 'your_cosmos_db_database_name'
container_name = 'your_cosmos_db_container_name'

with open('../js/trgPreValidateToDoItemTimestamp.js') as file:
    file_contents = file.read()

trigger_definition = {
    'id': 'trgPreValidateToDoItemTimestamp',
    'serverScript': file_contents,
    'triggerType': documents.TriggerType.Pre,
    'triggerOperation': documents.TriggerOperation.All
}
client = cosmos_client.CosmosClient(url, key)
database = client.get_database_client(database_name)
container = database.get_container_client(container_name)
trigger = container.scripts.create_trigger(trigger_definition)
```

The following code shows how to call a pre-trigger using the Python SDK:

```
item = {'category': 'Personal', 'name': 'Groceries',
        'description': 'Pick up strawberries', 'isComplete': False}
container.create_item(item, {'pre_trigger_include': 'trgPreValidateToDoItemTimestamp'})
```

How to run post-triggers

The following examples show how to register a post-trigger by using the Azure Cosmos DB SDKs. Refer to the Post-trigger example as the source for this post-trigger is saved as `trgPostUpdateMetadata.js`.

Post-triggers - .NET SDK V2

The following code shows how to register a post-trigger using the .NET SDK V2:

```
string triggerId = "trgPostUpdateMetadata";
Trigger trigger = new Trigger
{
    Id = triggerId,
    Body = File.ReadAllText($@"..\js\{triggerId}.js"),
    TriggerOperation = TriggerOperation.Create,
    TriggerType = TriggerType.Post
};
Uri containerUri = UriFactory.CreateDocumentCollectionUri("myDatabase", "myContainer");
await client.CreateTriggerAsync(containerUri, trigger);
```

The following code shows how to call a post-trigger using the .NET SDK V2:

```
var newItem = {
    name: "artist_profile_1023",
    artist: "The Band",
    albums: ["Hellujah", "Rotators", "Spinning Top"]
};

RequestOptions options = new RequestOptions { PostTriggerInclude = new List<string> { "trgPostUpdateMetadata"
} };
Uri containerUri = UriFactory.CreateDocumentCollectionUri("myDatabase", "myContainer");
await client.createDocumentAsync(containerUri, newItem, options);
```

Post-triggers - .NET SDK V3

The following code shows how to register a post-trigger using the .NET SDK V3:

```
await client.GetContainer("database", "container").Scripts.CreateTriggerAsync(new TriggerProperties
{
    Id = "trgPostUpdateMetadata",
    Body = File.ReadAllText(@"..\js\trgPostUpdateMetadata.js"),
    TriggerOperation = TriggerOperation.Create,
    TriggerType = TriggerType.Post
});
```

The following code shows how to call a post-trigger using the .NET SDK V3:

```
var newItem = {
    name: "artist_profile_1023",
    artist: "The Band",
    albums: ["Hellujah", "Rotators", "Spinning Top"]
};

await client.GetContainer("database", "container").CreateItemAsync(newItem, null, new ItemRequestOptions {
PostTriggers = new List<string> { "trgPostUpdateMetadata" } });
```

Post-triggers - Java SDK

The following code shows how to register a post-trigger using the Java SDK:

```
String containerLink = String.format("/dbs/%s/colls/%s", "myDatabase", "myContainer");
String triggerId = "trgPostUpdateMetadata";
Trigger trigger = new Trigger();
trigger.setId(triggerId);
trigger.setBody(new String(Files.readAllBytes(Paths.get(String.format("..\\js\\%s.js", triggerId)))));
trigger.setTriggerOperation(TriggerOperation.Create);
trigger.setTriggerType(TriggerType.Post);
Trigger createdTrigger = asyncClient.createTrigger(containerLink, trigger, new
RequestOptions()).toBlocking().single().getResource();
```

The following code shows how to call a post-trigger using the Java SDK:

```
String containerLink = String.format("/dbs/%s/colls/%s", "myDatabase", "myContainer");
Document item = new Document(String.format("{ "
    + "\"name\": \"artist_profile_1023\", "
    + "\"artist\": \"The Band\", "
    + "\"albums\": [\"Hellujah\", \"Rotators\", \"Spinning Top\"]"
    + "}"
));
RequestOptions requestOptions = new RequestOptions();
requestOptions.setPostTriggerInclude(Arrays.asList("trgPostUpdateMetadata"));
//toBlocking() blocks the thread until the operation is complete, and is used only for demo.
asyncClient.createDocument(containerLink, item, requestOptions, false).toBlocking();
```

Post-triggers - JavaScript SDK

The following code shows how to register a post-trigger using the JavaScript SDK:

```
const container = client.database("myDatabase").container("myContainer");
const triggerId = "trgPostUpdateMetadata";
await container.triggers.create({
    id: triggerId,
    body: require(`../js/${triggerId}`),
    triggerOperation: "create",
    triggerType: "post"
});
```

The following code shows how to call a post-trigger using the JavaScript SDK:

```
const item = {
    name: "artist_profile_1023",
    artist: "The Band",
    albums: ["Hellujah", "Rotators", "Spinning Top"]
};
const container = client.database("myDatabase").container("myContainer");
const triggerId = "trgPostUpdateMetadata";
await container.items.create(item, {postTriggerInclude: [triggerId]});
```

Post-triggers - Python SDK

The following code shows how to register a post-trigger using the Python SDK:

```
import azure.cosmos.cosmos_client as cosmos_client

url = "your_cosmos_db_account_URI"
key = "your_cosmos_db_account_key"
database_name = 'your_cosmos_db_database_name'
container_name = 'your_cosmos_db_container_name'

with open('../js/trgPostValidateToDoItemTimestamp.js') as file:
    file_contents = file.read()

trigger_definition = {
    'id': 'trgPostValidateToDoItemTimestamp',
    'serverScript': file_contents,
    'triggerType': documents.TriggerType.Post,
    'triggerOperation': documents.TriggerOperation.All
}
client = cosmos_client.CosmosClient(url, key)
database = client.get_database_client(database_name)
container = database.get_container_client(container_name)
trigger = container.scripts.create_trigger(trigger_definition)
```

The following code shows how to call a post-trigger using the Python SDK:

```
item = {'category': 'Personal', 'name': 'Groceries',
        'description': 'Pick up strawberries', 'isComplete': False}
container.create_item(item, {'post_trigger_include': 'trgPreValidateToDoItemTimestamp'})
```

How to work with user-defined functions

The following examples show how to register a user-defined function by using the Azure Cosmos DB SDKs. Refer to this User-defined function example as the source for this post-trigger is saved as `udfTax.js`.

User-defined functions - .NET SDK V2

The following code shows how to register a user-defined function using the .NET SDK V2:

```
string udfId = "Tax";
var udfTax = new UserDefinedFunction
{
    Id = udfId,
    Body = File.ReadAllText($@"..\js\{udfId}.js")
};

Uri containerUri = UriFactory.CreateDocumentCollectionUri("myDatabase", "myContainer");
await client.CreateUserDefinedFunctionAsync(containerUri, udfTax);
```

The following code shows how to call a user-defined function using the .NET SDK V2:

```
Uri containerUri = UriFactory.CreateDocumentCollectionUri("myDatabase", "myContainer");
var results = client.CreateDocumentQuery<dynamic>(containerUri, "SELECT * FROM Incomes t WHERE
udf.Tax(t.income) > 20000"));

foreach (var result in results)
{
    //iterate over results
}
```

User-defined functions - .NET SDK V3

The following code shows how to register a user-defined function using the .NET SDK V3:

```
await client.GetContainer("database", "container").Scripts.CreateUserDefinedFunctionAsync(new
UserDefinedFunctionProperties
{
    Id = "Tax",
    Body = File.ReadAllText(@"..\js\Tax.js")
});
```

The following code shows how to call a user-defined function using the .NET SDK V3:

```
var iterator = client.GetContainer("database", "container").GetItemQueryIterator<dynamic>("SELECT * FROM
Incomes t WHERE udf.Tax(t.income) > 20000");
while (iterator.HasMoreResults)
{
    var results = await iterator.ReadNextAsync();
    foreach (var result in results)
    {
        //iterate over results
    }
}
```

User-defined functions - Java SDK

The following code shows how to register a user-defined function using the Java SDK:

```
String containerLink = String.format("/dbs/%s/colls/%s", "myDatabase", "myContainer");
String udfId = "Tax";
UserDefinedFunction udf = new UserDefinedFunction();
udf.setId(udfId);
udf.setBody(new String(Files.readAllBytes(Paths.get(String.format("..\\js\\%s.js", udfId)))));
//toBlocking() blocks the thread until the operation is complete and is used only for demo.
UserDefinedFunction createdUDF = client.createUserDefinedFunction(containerLink, udf, new
RequestOptions()).toBlocking().single().getResource();
```

The following code shows how to call a user-defined function using the Java SDK:

```
String containerLink = String.format("/dbs/%s/colls/%s", "myDatabase", "myContainer");
Observable<FeedResponse<Document>> queryObservable = client.queryDocuments(containerLink, "SELECT * FROM
Incomes t WHERE udf.Tax(t.income) > 20000", new FeedOptions());
final CountDownLatch completionLatch = new CountDownLatch(1);
queryObservable.subscribe(
        queryResultPage -> {
            System.out.println("Got a page of query result with " +
                    queryResultPage.getResults().size());
        },
        // terminal error signal
        e -> {
            e.printStackTrace();
            completionLatch.countDown();
        },

        // terminal completion signal
        () -> {
            completionLatch.countDown();
        });
completionLatch.await();
```

User-defined functions - JavaScript SDK

The following code shows how to register a user-defined function using the JavaScript SDK:

```
const container = client.database("myDatabase").container("myContainer");
const udfId = "Tax";
await container.userDefinedFunctions.create({
    id: udfId,
    body: require(`../js/${udfId}`)
```

The following code shows how to call a user-defined function using the JavaScript SDK:

```
const container = client.database("myDatabase").container("myContainer");
const sql = "SELECT * FROM Incomes t WHERE udf.Tax(t.income) > 20000";
const {result} = await container.items.query(sql).toArray();
```

User-defined functions - Python SDK

The following code shows how to register a user-defined function using the Python SDK:

```
import azure.cosmos.cosmos_client as cosmos_client

url = "your_cosmos_db_account_URI"
key = "your_cosmos_db_account_key"
database_name = 'your_cosmos_db_database_name'
container_name = 'your_cosmos_db_container_name'

with open('../js/udfTax.js') as file:
    file_contents = file.read()
udf_definition = {
    'id': 'Tax',
    'serverScript': file_contents,
}
client = cosmos_client.CosmosClient(url, key)
database = client.get_database_client(database_name)
container = database.get_container_client(container_name)
udf = container.scripts.create_user_defined_function(udf_definition)
```

The following code shows how to call a user-defined function using the Python SDK:

```
results = list(container.query_items(
    'query': 'SELECT * FROM Incomes t WHERE udf.Tax(t.income) > 20000'))
```

Next steps

Learn more concepts and how-to write or use stored procedures, triggers, and user-defined functions in Azure Cosmos DB:

- Working with Azure Cosmos DB stored procedures, triggers, and user-defined functions in Azure Cosmos DB
- Working with JavaScript language integrated query API in Azure Cosmos DB
- How to write stored procedures, triggers, and user-defined functions in Azure Cosmos DB
- How to write stored procedures and triggers using Javascript Query API in Azure Cosmos DB

Enable notebooks for Azure Cosmos DB accounts (preview)

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: SQL API

> **IMPORTANT**
>
> Built-in notebooks for Azure Cosmos DB are currently available in the following Azure regions: Australia East, East US, East US 2, North Europe, South Central US, Southeast Asia, UK South, West Europe and West US 2. To use notebooks, create a new account with notebooks or enable notebooks on an existing account in one of these regions.

Built-in Jupyter notebooks in Azure Cosmos DB enable you to analyze and visualize your data from the Azure portal. This article describes how to enable this feature for your Azure Cosmos DB account.

Enable notebooks in a new Cosmos account

1. Sign into the Azure portal.
2. Select **Create a resource** > **Databases** > **Azure Cosmos DB**.
3. On the **Create Azure Cosmos DB Account** page, select **Notebooks**.

4. Select **Review + create**. You can skip the **Network** and **Tags** option.
5. Review the account settings, and then select **Create**. It takes a few minutes to create the account. Wait for the portal page to display **Your deployment is complete**.

6. Select **Go to resource** to go to the Azure Cosmos DB account page.

7. Navigate to the **Data Explorer** pane. You should now see your notebooks workspace.

Enable notebooks in an existing Cosmos account

You can also enable notebooks on existing accounts. This step needs to be done only once per account.

1. Navigate to the **Data Explorer** pane in your Cosmos account.

2. Select **Enable Notebooks**.

3. This will prompt you to create a new notebooks workspace. Select **Complete setup.**

4. Your account is now enabled to use notebooks!

Create and run your first notebook

To verify that you can use notebooks, select one of notebooks under Sample Notebooks. This will save a copy of the notebook to your workspace and open it.

In this example, we'll use **GettingStarted.ipynb**.

To run the notebook:

1. Select the first code cell that contains Python code.

2. Select **Run** to run the cell. You can also use **Shift + Enter** to run the cell.

3. Refresh the resource pane to see the database and container that has been created.

You can also select **New Notebook** to create a new notebook or upload an existing notebook (.ipynb) file by selecting **Upload File** from the **My Notebooks** menu.

Next steps

- Learn about the benefits of Azure Cosmos DB Jupyter Notebooks

Import notebooks from a GitHub repo into Azure Cosmos DB

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: SQL API

After you enable notebook support for your Azure Cosmos accounts, you can create new notebooks, upload new notebooks from your local computer, or import the existing notebooks from your GitHub accounts. This article shows how to connect your notebooks workspace to GitHub and import the notebooks from a GitHub repo to your Azure Cosmos account. After importing, you can run them, make changes, and save the changes back to GitHub.

Get notebooks from GitHub

You can connect to your own GitHub repositories or other public GitHub repositories to read, author, and share notebooks in Azure Cosmos DB. Use the following steps to connect to a GitHub account:

1. Sign into Azure portal and navigate to your Azure Cosmos account.
2. Open the **Data Explorer** tab. This tab will show all your existing databases, containers, and notebooks.
3. Select the **Connect to GitHub** menu item.
4. A tab opens where you can choose to connect to **Public repos** only or **Public and private repos**. After choosing the required option, select **Authorize access**. Authorization is required for Azure Cosmos DB to access the repositories in your GitHub account.

5. You are redirected to "github.com" web page where you can confirm the authorization. Select the **Authorize AzureCosmosDBNotebooks** button and enter your GitHub account password in the prompt.
6. After the authorization is successful, it takes you back to your Azure Cosmos account. You can then see all the public/private repos from your GitHub account. You can select a repo from the list available or add a repo directly by using its URL.
7. Once you select the required repo, the repo entry moves from the **Unpinned repos** section to **Pinned repos** section. If needed, you can also choose a specific branch of that repo to import the notebooks from.

8. Select **OK** to complete the import operation. All the notebooks available in the selected branch of your repo are imported into your Azure Cosmos account.

After you integrate with a GitHub account, only you can see the list of repositories and notebooks in your Azure Cosmos account. This statement is true even if multiple users log into the Azure Cosmos DB account and add their own accounts. In other words, multiple users can use the same Azure Cosmos account to connect the notebook workspace to GitHub. However, each user only sees the list of repositories and notebooks they have imported. The notebooks imported by others are not visible to you.

To disconnect your GitHub account from the notebooks workspace, open the **Data Explorer** tab, select ... next to **GitHub repos** and select **Disconnect from GitHub**.

Edit a notebook and push changes to GitHub

You can edit an existing notebook or add a new notebook to the repo and save the changes back to GitHub.

After you edit an existing notebook, select **Save**. A dialog box opens where you can enter the commit message for the changes you have made. Select **Commit** and the notebook in GitHub is updated. You can validate the updates by signing into your GitHub account, and verifying the commit history.

In the regular GitHub flow after committing the changes, you will typically push changes to a remote. However, in this case the commit option serves the purpose of "staging, committing, and pushing" your updates to GitHub.

Next steps

- Learn about the benefits of Azure Cosmos DB Jupyter notebooks.

Use built-in notebook commands and features in Azure Cosmos DB Python notebooks (preview)

12/4/2020 • 5 minutes to read • Edit Online

APPLIES TO: SQL API

Built-in Jupyter notebooks in Azure Cosmos DB enable you to analyze and visualize your data from the Azure portal. This article describes how to use built-in notebook commands and features to do common operations in Python notebooks.

Install a new package

After you enable notebook support for your Azure Cosmos accounts, you can open a new notebook and install a package.

In a new code cell, insert and run the following code, replacing `PackageToBeInstalled` with the desired Python package.

```
import sys
!{sys.executable} -m pip install PackageToBeInstalled --user
```

This package will be available to use from any notebook in the Azure Cosmos account workspace.

> **TIP**
> If your notebook requires a custom package, we recommend that you add a cell in your notebook to install the package, as packages are removed if you reset the workspace.

Run a SQL query

You can use the `%%sql` magic command to run a SQL query against any container in your account. Use the syntax:

```
%%sql --database {database_id} --container {container_id}
{Query text}
```

- Replace `{database_id}` and `{container_id}` with the name of the database and container in your Cosmos account. If the `--database` and `--container` arguments are not provided, the query will be executed on the default database and container.
- You can run any SQL query that is valid in Azure Cosmos DB. The query text must be on a new line.

For example:

```
%%sql --database RetailDemo --container WebsiteData
SELECT c.Action, c.Price as ItemRevenue, c.Country, c.Item FROM c
```

Run `%%sql?` in a cell to see the help documentation for the sql magic command in the notebook.

Run a SQL query and output to a Pandas DataFrame

You can output the results of a `%%sql` query into a Pandas DataFrame. Use the syntax:

```
%%sql --database {database_id} --container {container_id} --output {outputDataFrameVar}
{Query text}
```

- Replace `{database_id}` and `{container_id}` with the name of the database and container in your Cosmos account. If the `--database` and `--container` arguments are not provided, the query will be executed on the default database and container.
- Replace `{outputDataFrameVar}` with the name of the DataFrame variable that will contain the results.
- You can run any SQL query that is valid in Azure Cosmos DB. The query text must be on a new line.

For example:

```
%%sql --database RetailDemo --container WebsiteData --output df_cosmos
SELECT c.Action, c.Price as ItemRevenue, c.Country, c.Item FROM c
```

```
df_cosmos.head(10)

    Action    ItemRevenue    Country/Region    Item
0    Viewed    9.00    Tunisia    Black Tee
1    Viewed    19.99    Antigua and Barbuda    Flannel Shirt
2    Added    3.75    Guinea-Bissau    Socks
3    Viewed    3.75    Guinea-Bissau    Socks
4    Viewed    55.00    Czech Republic    Rainjacket
5    Viewed    350.00    Iceland    Cosmos T-shirt
6    Added    19.99    Syrian Arab Republic    Button-Up Shirt
7    Viewed    19.99    Syrian Arab Republic    Button-Up Shirt
8    Viewed    33.00    Tuvalu    Red Top
9    Viewed    14.00    Cabo Verde    Flip Flop Shoes
```

Set default database for queries

You can set the default database `%%sql` commands will use for the notebook. Replace `{database_id}` with the name of your database.

```
%database {database_id}
```

Run `%database?` in a cell to see documentation in the notebook.

Set default container for queries

You can set the default container `%%sql` commands will use for the notebook. Replace `{container_id}` with the name of your container.

```
%container {container_id}
```

Run `%container?` in a cell to see documentation in the notebook.

Upload JSON items to a container

You can use the `%%upload` magic command to upload data from a JSON file to a specified Azure Cosmos container. Use the following command to upload the items:

```
%%upload --databaseName {database_id} --containerName {container_id} --url {url_location_of_file}
```

- Replace `{database_id}` and `{container_id}` with the name of the database and container in your Azure Cosmos account. If the `--database` and `--container` arguments are not provided, the query will be executed on the default database and container.
- Replace `{url_location_of_file}` with the location of your JSON file. The file must be an array of valid JSON objects and it should be accessible over the public Internet.

For example:

```
%%upload --database databaseName --container containerName --url
https://contoso.com/path/to/data.json
```

```
Documents successfully uploaded to ContainerName
Total number of documents imported : 2654
Total time taken : 00:00:38.1228087 hours
Total RUs consumed : 25022.58
```

With the output statistics, you can calculate the effective RU/s used to upload the items. For example, if 25,000 RUs were consumed over 38 seconds, the effective RU/s is 25,000 RUs / 38 seconds = 658 RU/s.

Run another notebook in current notebook

You can use the `%%run` magic command to run another notebook in your workspace from your current notebook. Use the syntax:

```
%%run ./path/to/{notebookName}.ipynb
```

Replace `{notebookName}` with the name of the notebook you want to run. The notebook must be in your current 'My Notebooks' workspace.

Use built-in nteract data explorer

You can use the built-in nteract data explorer to filter and visualize a DataFrame. To enable this feature, set the option `pd.options.display.html.table_schema` to `True` and `pd.options.display.max_rows` to the desired value (you can set `pd.options.display.max_rows` to `None` to show all results).

```
import pandas as pd
pd.options.display.html.table_schema = True
pd.options.display.max_rows = None

df_cosmos.groupby("Item").size()
```

Use the built-in Python SDK

Version 4 of the Azure Cosmos DB Python SDK for SQL API is installed and included in the notebook environment for the Azure Cosmos account.

Use the built-in `cosmos_client` instance to run any SDK operation.

For example:

```
## Import modules as needed
from azure.cosmos.partition_key import PartitionKey

## Create a new database if it doesn't exist
database = cosmos_client.create_database_if_not_exists('RetailDemo')

## Create a new container if it doesn't exist
container = database.create_container_if_not_exists(id='WebsiteData',
partition_key=PartitionKey(path='/CartID'))
```

See Python SDK samples.

> **IMPORTANT**
>
> The built-in Python SDK is only supported for SQL (Core) API accounts. For other APIs, you will need to install the relevant Python driver that corresponds to the API.

Create a custom instance of `cosmos_client`

For more flexibility, you can create a custom instance of `cosmos_client` in order to:

- Customize the connection policy
- Run operations against a different Azure Cosmos account than the one you are in

You can access the connection string and primary key of the current account via the environment variables.

```
import azure.cosmos.cosmos_client as cosmos
import azure.cosmos.documents as documents

# These should be set to a region you've added for Azure Cosmos DB
region_1 = "Central US"
region_2 = "East US 2"

custom_connection_policy = documents.ConnectionPolicy()
custom_connection_policy.PreferredLocations = [region_1, region_2] # Set the order of regions the SDK will
route requests to. The regions should be regions you've added for Cosmos, otherwise this will error.

# Create a new instance of CosmosClient, getting the endpoint and key from the environment variables
custom_client = cosmos.CosmosClient(url=COSMOS.ENDPOINT, credential=COSMOS.KEY,
connection_policy=custom_connection_policy)
```

Access the account endpoint and primary key env variables

You can access the built-in endpoint and key of the account your notebook is in.

```
endpoint = COSMOS.ENDPOINT
primary_key = COSMOS.KEY
```

> **IMPORTANT**
>
> The `COSMOS.ENDPOINT` and `COSMOS.KEY` variables are only applicable for SQL API. For other APIs, find the endpoint and key in the **Connection Strings** or **Keys** blade in your Azure Cosmos account.

Reset notebooks workspace

To reset the notebooks workspace to the default settings, select **Reset Workspace** on the command bar. This will remove any custom installed packages and restart the Jupyter server. Your notebooks, files, and Azure Cosmos resources will not be affected.

Next steps

- Learn about the benefits of Azure Cosmos DB Jupyter notebooks
- Learn about the Azure Cosmos DB Python SDK for SQL API

Use built-in notebook commands and features in Azure Cosmos DB C# notebooks (preview)

12/4/2020 • 5 minutes to read • Edit Online

APPLIES TO: SQL API

Built-in Jupyter notebooks in Azure Cosmos DB enable you to analyze and visualize your data from the Azure portal. This article describes how to use built-in notebook commands and features to do common operations in C# notebooks.

Install a new NuGet package

After you enable notebook support for your Azure Cosmos accounts, you can open a new notebook and install a package.

In a new code cell, insert and run the following code, replacing `PackageToBeInstalled` with the desired NuGet package, and `optionalVersion` with a specific version of the package if desired.

```
#r "nuget: PackageToBeInstalled, optionalVersion"
```

You can install multiple NuGet packages in the same cell. Packages will be available to use from any notebook in the Azure Cosmos account workspace.

Currently, the C# notebooks workspace does not support recursive resolution of NuGet packages. If a NuGet package has dependencies on other NuGet packages that are not currently installed, you have to explicitly reference them along with the parent package.

> **TIP**
> If your notebook requires a custom package, we recommend that you add a cell to your notebook to install the package and make it the first cell. This reduces the chance of conflicts with other packages that Azure Cosmos DB loads by default. It is also easy to re-install the packages if you reset the workspace, which removes all packages.

Use the built-in Azure Cosmos DB .NET SDK

Version 3 of the Azure Cosmos DB .NET SDK for SQL API is installed and included in the notebook environment for the Azure Cosmos account.

Create an instance of `CosmosClient` to run any SDK operation.

For example:

```
// Include usings
using System;
using Microsoft.Azure.Cosmos; //namespace for Azure Cosmos DB .NET V3 SDK
using System.Collections;

// Initialize a new instance of CosmosClient using the built-in account endpoint and key parameters
CosmosClient cosmosClient = new CosmosClient(Cosmos.Endpoint, Cosmos.Key);

// Create a new database and container with 400 RU/s
Database database = await cosmosClient.CreateDatabaseIfNotExistsAsync("DatabaseName");
Container container = await database.CreateContainerIfNotExistsAsync("ContainerName", "/partitionKey", 400);
```

To learn more, see the .NET V3 SDK samples.

> **IMPORTANT**
>
> The built-in Azure Cosmos DB .NET SDK is only supported for SQL (Core) API accounts. For other APIs, you will need to install the relevant .NET driver that corresponds to the API.

Set custom options using `CosmosClientOptions`

For more flexibility, you can set the custom `CosmosClientOptions` property and pass it in your `CosmosClient` instance. You can use this property to:

- Set an application name in the user-agent suffix to include it in every request.
- Set the preferred region to be used when running operations against the service.
- Set a custom retry policy to handle rate-limited requests.

See the CosmosClientOptions API reference article for all the supported settings. The following is an example that shows how to set `cosmosClientOptions` property:

```
using Microsoft.Azure.Cosmos;

// Configure CosmosClientOptions
var cosmosClientOptions = new CosmosClientOptions
{
    ApplicationName = "ContosoNotebookAppName",
    ApplicationRegion = Regions.RegionName, // By default, this is the region you first created your account
in
    MaxRetryAttemptsOnRateLimitedRequests = 9, // By default, this value is 9
};

var client = new CosmosClient(Cosmos.Endpoint, Cosmos.Key, cosmosClientOptions);
```

Access the account endpoint and primary key variables

You can access the built-in endpoint and key of the Azure Cosmos account where your notebook exists.

```
var key = Cosmos.Key;
var endpoint = Cosmos.Endpoint;
```

> **IMPORTANT**
>
> The `Cosmos.Key` and `Cosmos.Endpoint` variables are only applicable for SQL API. For other APIs, find the endpoint and key in the **Connection Strings** or **Keys** blade in your Azure Cosmos account.

Print console output in C# code

In your C# code, you can use the Display.AsMarkdown() syntax with string interpolation to print console output that will appear when you run the cell.

For example:

```
// Print text in the output
Display.AsMarkdown($"Hello world!");

// Print a variable in the output
for (int i = 0; i < 5; i++) {
    Display.AsMarkdown($"Printing out variable: {i}");
}
```

> **IMPORTANT**
>
> Console.WriteLine() syntax is not currently supported in C# notebooks. Use Display.AsMarkdown to print console output from your program.

Use built-in nteract data explorer

You can use the built-in nteract data explorer to filter and visualize a collection of items. In a cell, put the variable you want to visualize in the last line, which is automatically displayed in nteract when you run the cell.

For example, in the *GetingStarted_Csharp.ipynb* example, we can print out the variable with our result, the `telemetryEvents`. See the GettingStarted_Csharp.ipynb notebook for the entire sample.

Use built-in dictionary viewer

You can use the built-in dictionary viewer to view a variable. In a cell, put the variable you want to visualize in the last line, which will be automatically displayed when the cell is run.

Upload JSON items to a container

You can use the `%%upload` magic command to upload data from a JSON file to a specified Azure Cosmos container. Use the following command to upload the items:

```
%%upload --databaseName {database_id} --containerName {container_id} --url {url_location_of_file}
```

- Replace `{database_id}` and `{container_id}` with the name of the database and container in your Azure Cosmos account.
- Replace `{url_location_of_file}` with the location of your JSON file. The file must be an array of valid JSON objects and it should be accessible over the public Internet.

For example:

```
%%upload --database databaseName --container containerName --url
https://contoso.com/path/to/data.json
```

```
Documents successfully uploaded to ContainerName
Total number of documents imported : 2654
Total time taken : 00:00:38.1228087 hours
Total RUs consumed : 25022.58
```

With the output statistics, you can calculate the effective RU/s used to upload the items. For example, if 25,000 RUs

were consumed over 38 seconds, the effective RU/s is 25,000 RUs / 38 seconds = 658 RU/s.

Run another notebook in current notebook

You can use the `%%run` magic command to run another notebook in your workspace from your current notebook. Use the syntax:

```
%%run ./path/to/{notebookName}.ipynb
```

Replace `{notebookName}` with the name of the notebook you want to run. The notebook must be in your current 'My Notebooks' workspace.

Reset notebooks workspace

To reset the notebooks workspace to the default settings, select **Reset Workspace** on the command bar. This will remove any custom installed packages and restart the Jupyter server. Your notebooks, files, and Azure Cosmos resources will not be affected.

Next steps

- Learn about the benefits of Azure Cosmos DB Jupyter notebooks
- Learn about the Azure Cosmos DB .NET SDK for SQL API

Azure Cosmos DB bulk executor library overview

11/2/2020 • 2 minutes to read • Edit Online

APPLIES TO: SQL API

Azure Cosmos DB is a fast, flexible, and globally distributed database service that is designed to elastically scale out to support:

- Large read and write throughput (millions of operations per second).
- Storing high volumes of (hundreds of terabytes, or even more) transactional and operational data with predictable millisecond latency.

The bulk executor library helps you leverage this massive throughput and storage. The bulk executor library allows you to perform bulk operations in Azure Cosmos DB through bulk import and bulk update APIs. You can read more about the features of bulk executor library in the following sections.

> **NOTE**
> Currently, bulk executor library supports import and update operations and this library is supported by Azure Cosmos DB SQL API and Gremlin API accounts only.

> **IMPORTANT**
> The bulk executor library is not currently supported on serverless accounts. On .NET, it is recommended to use the bulk support available in the V3 version of the SDK.

Key features of the bulk executor library

- It significantly reduces the client-side compute resources needed to saturate the throughput allocated to a container. A single threaded application that writes data using the bulk import API achieves 10 times greater write throughput when compared to a multi-threaded application that writes data in parallel while saturating the client machine's CPU.
- It abstracts away the tedious tasks of writing application logic to handle rate limiting of request, request timeouts, and other transient exceptions by efficiently handling them within the library.
- It provides a simplified mechanism for applications performing bulk operations to scale out. A single bulk executor instance running on an Azure VM can consume greater than 500K RU/s and you can achieve a higher throughput rate by adding additional instances on individual client VMs.
- It can bulk import more than a terabyte of data within an hour by using a scale-out architecture.
- It can bulk update existing data in Azure Cosmos containers as patches.

How does the bulk executor operate?

When a bulk operation to import or update documents is triggered with a batch of entities, they are initially shuffled into buckets corresponding to their Azure Cosmos DB partition key range. Within each bucket that corresponds to a partition key range, they are broken down into mini-batches and each mini-batch act as a payload that is committed on the server-side. The bulk executor library has built in optimizations for concurrent execution of these mini-batches both within and across partition key ranges. Following image illustrates how bulk

executor batches data into different partition keys:

Azure Cosmos DB Bulk Executor Library

The bulk executor library makes sure to maximally utilize the throughput allocated to a collection. It uses anAIMD-style congestion control mechanism for each Azure Cosmos DB partition key rangeto efficiently handle rate limiting and timeouts.

Next Steps

- Learn more by trying out the sample applications consuming the bulk executor library in .NET and Java.
- Check out the bulk executor SDK information and release notes in .NET and Java.
- The bulk executor library is integrated into the Cosmos DB Spark connector, to learn more, see Azure Cosmos DB Spark connector article.
- The bulk executor library is also integrated into a new version of Azure Cosmos DB connector for Azure Data Factory to copy data.

Use the bulk executor .NET library to perform bulk operations in Azure Cosmos DB

12/4/2020 • 8 minutes to read • Edit Online

APPLIES TO: SQL API

> **NOTE**
>
> This bulk executor library described in this article is maintained for applications using the .NET SDK 2.x version. For new applications, you can use the **bulk support** that is directly available with the .NET SDK version 3.x and it does not require any external library.

> If you are currently using the bulk executor library and planning to migrate to bulk support on the newer SDK, use the steps in the Migration guide to migrate your application.

This tutorial provides instructions on using the bulk executor .NET library to import and update documents to an Azure Cosmos container. To learn about the bulk executor library and how it helps you leverage massive throughput and storage, see the bulk executor library overview article. In this tutorial, you will see a sample .NET application that bulk imports randomly generated documents into an Azure Cosmos container. After importing, it shows you how you can bulk update the imported data by specifying patches as operations to perform on specific document fields.

Currently, bulk executor library is supported by the Azure Cosmos DB SQL API and Gremlin API accounts only. This article describes how to use the bulk executor .NET library with SQL API accounts. To learn about using the bulk executor .NET library with Gremlin API accounts, see perform bulk operations in the Azure Cosmos DB Gremlin API.

Prerequisites

- If you don't already have Visual Studio 2019 installed, you can download and use the Visual Studio 2019 Community Edition. Make sure that you enable "Azure development" during the Visual Studio setup.
- If you don't have an Azure subscription, create a free account before you begin.
- You can Try Azure Cosmos DB for free without an Azure subscription, free of charge and commitments. Or, you can use the Azure Cosmos DB Emulator with the `https://localhost:8081` endpoint. The Primary Key is provided in Authenticating requests.
- Create an Azure Cosmos DB SQL API account by using the steps described in create database account section of the .NET quickstart article.

Clone the sample application

Now let's switch to working with code by downloading a sample .NET application from GitHub. This application performs bulk operations on the data stored in your Azure Cosmos account. To clone the application, open a command prompt, navigate to the directory where you want to copy it and run the following command:

```
git clone https://github.com/Azure/azure-cosmosdb-bulkexecutor-dotnet-getting-started.git
```

The cloned repository contains two samples "BulkImportSample" and "BulkUpdateSample". You can open either of the sample applications, update the connection strings in App.config file with your Azure Cosmos DB account's connection strings, build the solution, and run it.

The "BulkImportSample" application generates random documents and bulk imports them to your Azure Cosmos account. The "BulkUpdateSample" application bulk updates the imported documents by specifying patches as operations to perform on specific document fields. In the next sections, you will review the code in each of these sample apps.

Bulk import data to an Azure Cosmos account

1. Navigate to the "BulkImportSample" folder and open the "BulkImportSample.sln" file.

2. The Azure Cosmos DB's connection strings are retrieved from the App.config file as shown in the following code:

   ```
   private static readonly string EndpointUrl = ConfigurationManager.AppSettings["EndPointUrl"];
   private static readonly string AuthorizationKey = ConfigurationManager.AppSettings["AuthorizationKey"];
   private static readonly string DatabaseName = ConfigurationManager.AppSettings["DatabaseName"];
   private static readonly string CollectionName = ConfigurationManager.AppSettings["CollectionName"];
   private static readonly int CollectionThroughput =
   int.Parse(ConfigurationManager.AppSettings["CollectionThroughput"]);
   ```

   The bulk importer creates a new database and a container with the database name, container name, and the throughput values specified in the App.config file.

3. Next the DocumentClient object is initialized with Direct TCP connection mode:

   ```
   ConnectionPolicy connectionPolicy = new ConnectionPolicy
   {
      ConnectionMode = ConnectionMode.Direct,
      ConnectionProtocol = Protocol.Tcp
   };
   DocumentClient client = new DocumentClient(new Uri(endpointUrl),authorizationKey,
   connectionPolicy)
   ```

4. The BulkExecutor object is initialized with a high retry value for wait time and throttled requests. And then they are set to 0 to pass congestion control to BulkExecutor for its lifetime.

   ```
   // Set retry options high during initialization (default values).
   client.ConnectionPolicy.RetryOptions.MaxRetryWaitTimeInSeconds = 30;
   client.ConnectionPolicy.RetryOptions.MaxRetryAttemptsOnThrottledRequests = 9;

   IBulkExecutor bulkExecutor = new BulkExecutor(client, dataCollection);
   await bulkExecutor.InitializeAsync();

   // Set retries to 0 to pass complete control to bulk executor.
   client.ConnectionPolicy.RetryOptions.MaxRetryWaitTimeInSeconds = 0;
   client.ConnectionPolicy.RetryOptions.MaxRetryAttemptsOnThrottledRequests = 0;
   ```

5. The application invokes the BulkImportAsync API. The .NET library provides two overloads of the bulk import API - one that accepts a list of serialized JSON documents and the other that accepts a list of deserialized POCO documents. To learn more about the definitions of each of these overloaded methods, refer to the API documentation.

```
BulkImportResponse bulkImportResponse = await bulkExecutor.BulkImportAsync(
  documents: documentsToImportInBatch,
  enableUpsert: true,
  disableAutomaticIdGeneration: true,
  maxConcurrencyPerPartitionKeyRange: null,
  maxInMemorySortingBatchSize: null,
  cancellationToken: token);
```

BulkImportAsync method accepts the following parameters:

PARAMETER	DESCRIPTION
enableUpsert	A flag to enable upsert operations on the documents. If a document with the given ID already exists, it's updated. By default, it is set to false.
disableAutomaticIdGeneration	A flag to disable automatic generation of ID. By default, it is set to true.
maxConcurrencyPerPartitionKeyRange	The maximum degree of concurrency per partition key range, setting to null will cause library to use a default value of 20.
maxInMemorySortingBatchSize	The maximum number of documents that are pulled from the document enumerator, which is passed to the API call in each stage. For in-memory sorting phase that happens before bulk importing, setting this parameter to null will cause library to use default minimum value (documents.count, 1000000).
cancellationToken	The cancellation token to gracefully exit the bulk import operation.

Bulk import response object definition The result of the bulk import API call contains the following attributes:

PARAMETER	DESCRIPTION
NumberOfDocumentsImported (long)	The total number of documents that were successfully imported out of the total documents supplied to the bulk import API call.
TotalRequestUnitsConsumed (double)	The total request units (RU) consumed by the bulk import API call.
TotalTimeTaken (TimeSpan)	The total time taken by the bulk import API call to complete the execution.
BadInputDocuments (List<object>)	The list of bad-format documents that were not successfully imported in the bulk import API call. Fix the documents returned and retry import. Bad-formatted documents include documents whose ID value is not a string (null or any other datatype is considered invalid).

Bulk update data in your Azure Cosmos account

You can update existing documents by using the BulkUpdateAsync API. In this example, you will set the `Name` field

to a new value and remove the `Description` field from the existing documents. For the full set of supported update operations, refer to the API documentation.

1. Navigate to the "BulkUpdateSample" folder and open the "BulkUpdateSample.sln" file.

2. Define the update items along with the corresponding field update operations. In this example, you will use `SetUpdateOperation` to update the `Name` field and `UnsetUpdateOperation` to remove the `Description` field from all the documents. You can also perform other operations like increment a document field by a specific value, push specific values into an array field, or remove a specific value from an array field. To learn about different methods provided by the bulk update API, refer to the API documentation.

```
SetUpdateOperation<string> nameUpdate = new SetUpdateOperation<string>("Name", "UpdatedDoc");
UnsetUpdateOperation descriptionUpdate = new UnsetUpdateOperation("description");

List<UpdateOperation> updateOperations = new List<UpdateOperation>();
updateOperations.Add(nameUpdate);
updateOperations.Add(descriptionUpdate);

List<UpdateItem> updateItems = new List<UpdateItem>();
for (int i = 0; i < 10; i++)
{
 updateItems.Add(new UpdateItem(i.ToString(), i.ToString(), updateOperations));
}
```

3. The application invokes the BulkUpdateAsync API. To learn about the definition of the BulkUpdateAsync method, refer to the API documentation.

```
BulkUpdateResponse bulkUpdateResponse = await bulkExecutor.BulkUpdateAsync(
  updateItems: updateItems,
  maxConcurrencyPerPartitionKeyRange: null,
  maxInMemorySortingBatchSize: null,
  cancellationToken: token);
```

BulkUpdateAsync method accepts the following parameters:

PARAMETER	DESCRIPTION
maxConcurrencyPerPartitionKeyRange	The maximum degree of concurrency per partition key range, setting this parameter to null will make the library to use the default value(20).
maxInMemorySortingBatchSize	The maximum number of update items pulled from the update items enumerator passed to the API call in each stage. For the in-memory sorting phase that happens before bulk updating, setting this parameter to null will cause the library to use the default minimum value(updateItems.count, 1000000).
cancellationToken	The cancellation token to gracefully exit the bulk update operation.

Bulk update response object definition The result of the bulk update API call contains the following attributes:

PARAMETER	DESCRIPTION

PARAMETER	DESCRIPTION
NumberOfDocumentsUpdated (long)	The number of documents that were successfully updated out of the total documents supplied to the bulk update API call.
TotalRequestUnitsConsumed (double)	The total request units (RUs) consumed by the bulk update API call.
TotalTimeTaken (TimeSpan)	The total time taken by the bulk update API call to complete the execution.

Performance tips

Consider the following points for better performance when using the bulk executor library:

- For best performance, run your application from an Azure virtual machine that is in the same region as your Azure Cosmos account's write region.
- It is recommended that you instantiate a single `BulkExecutor` object for the whole application within a single virtual machine that corresponds to a specific Azure Cosmos container.
- Since a single bulk operation API execution consumes a large chunk of the client machine's CPU and network IO (This happens by spawning multiple tasks internally). Avoid spawning multiple concurrent tasks within your application process that execute bulk operation API calls. If a single bulk operation API call that is running on a single virtual machine is unable to consume the entire container's throughput (if your container's throughput > 1 million RU/s), it's preferred to create separate virtual machines to concurrently execute the bulk operation API calls.
- Ensure the `InitializeAsync()` method is invoked after instantiating a BulkExecutor object to fetch the target Cosmos container's partition map.
- In your application's App.Config, ensure **`gcServer`** is enabled for better performance

```
<runtime>
  <gcServer enabled="true" />
</runtime>
```

- The library emits traces that can be collected either into a log file or on the console. To enable both, add the following code to your application's App.Config file.

```
<system.diagnostics>
  <trace autoflush="false" indentsize="4">
    <listeners>
      <add name="logListener" type="System.Diagnostics.TextWriterTraceListener"
initializeData="application.log" />
      <add name="consoleListener" type="System.Diagnostics.ConsoleTraceListener" />
    </listeners>
  </trace>
</system.diagnostics>
```

Next steps

- To learn about the NuGet package details and the release notes, see the bulk executor SDK details.

Migrate from the bulk executor library to the bulk support in Azure Cosmos DB .NET V3 SDK

12/4/2020 • 4 minutes to read • Edit Online

APPLIES TO: SQL API

This article describes the required steps to migrate an existing application's code that uses the .NET bulk executor library to the bulk support feature in the latest version of the .NET SDK.

Enable bulk support

Enable bulk support on the `CosmosClient` instance through the AllowBulkExecution configuration:

```
new CosmosClient(endpoint, authKey, new CosmosClientOptions() { AllowBulkExecution = true });
```

Create Tasks for each operation

Bulk support in the .NET SDK works by leveraging the Task Parallel Library and grouping operations that occur concurrently.

There is no single method in the SDK that will take your list of documents or operations as an input parameter, but rather, you need to create a Task for each operation you want to execute in bulk, and then simply wait for them to complete.

For example, if your initial input is a list of items where each item has the following schema:

```
public class MyItem
{
    public string id { get; set; }

    public string pk { get; set; }

    public int operationCounter { get; set; } = 0;
}
```

If you want to do bulk import (similar to using BulkExecutor.BulkImportAsync), you need to have concurrent calls to `CreateItemAsync`. For example:

```
BulkOperations<MyItem> bulkOperations = new BulkOperations<MyItem>(documentsToWorkWith.Count);
foreach (MyItem document in documentsToWorkWith)
{
    bulkOperations.Tasks.Add(container.CreateItemAsync(document, new
PartitionKey(document.pk)).CaptureOperationResponse(document));
}
```

If you want to do bulk *update* (similar to using BulkExecutor.BulkUpdateAsync), you need to have concurrent calls to `ReplaceItemAsync` method after updating the item value. For example:

```
BulkOperations<MyItem> bulkOperations = new BulkOperations<MyItem>(documentsToWorkWith.Count);
foreach (MyItem document in documentsToWorkWith)
{
    document.operationCounter++;
    bulkOperations.Tasks.Add(container.ReplaceItemAsync(document, document.id, new
PartitionKey(document.pk)).CaptureOperationResponse(document));
}
```

And if you want to do bulk *delete* (similar to using BulkExecutor.BulkDeleteAsync), you need to have concurrent calls to `DeleteItemAsync`, with the `id` and partition key of each item. For example:

```
BulkOperations<MyItem> bulkOperations = new BulkOperations<MyItem>(documentsToWorkWith.Count);
foreach (MyItem document in documentsToWorkWith)
{
    document.operationCounter++;
    bulkOperations.Tasks.Add(container.DeleteItemAsync<MyItem>(document.id, new
PartitionKey(document.pk)).CaptureOperationResponse(document));
}
```

Capture task result state

In the previous code examples, we have created a concurrent list of tasks, and called the `CaptureOperationResponse` method on each of those tasks. This method is an extension that lets us maintain a *similar response schema* as BulkExecutor, by capturing any errors and tracking the request units usage.

```
public static Task<OperationResponse<T>> CaptureOperationResponse<T>(this Task<ItemResponse<T>> task, T item)
{
    return task.ContinueWith(itemResponse =>
    {
        if (itemResponse.IsCompletedSuccessfully)
        {
            return new OperationResponse<T>()
            {
                Item = item,
                IsSuccessful = true,
                RequestUnitsConsumed = task.Result.RequestCharge
            };
        }

        AggregateException innerExceptions = itemResponse.Exception.Flatten();
        if (innerExceptions.InnerExceptions.FirstOrDefault(innerEx => innerEx is CosmosException) is
CosmosException cosmosException)
        {
            return new OperationResponse<T>()
            {
                Item = item,
                RequestUnitsConsumed = cosmosException.RequestCharge,
                IsSuccessful = false,
                CosmosException = cosmosException
            };
        }

        return new OperationResponse<T>()
        {
            Item = item,
            IsSuccessful = false,
            CosmosException = innerExceptions.InnerExceptions.FirstOrDefault()
        };
    });
}
```

Where the `OperationResponse` is declared as:

```
public class OperationResponse<T>
{
    public T Item { get; set; }
    public double RequestUnitsConsumed { get; set; } = 0;
    public bool IsSuccessful { get; set; }
    public Exception CosmosException { get; set; }
}
```

Execute operations concurrently

To track the scope of the entire list of Tasks, we use this helper class:

```
public class BulkOperations<T>
{
    public readonly List<Task<OperationResponse<T>>> Tasks;

    private readonly Stopwatch stopwatch = Stopwatch.StartNew();

    public BulkOperations(int operationCount)
    {
        this.Tasks = new List<Task<OperationResponse<T>>>(operationCount);
    }

    public async Task<BulkOperationResponse<T>> ExecuteAsync()
    {
        await Task.WhenAll(this.Tasks);
        this.stopwatch.Stop();
        return new BulkOperationResponse<T>()
        {
            TotalTimeTaken = this.stopwatch.Elapsed,
            TotalRequestUnitsConsumed = this.Tasks.Sum(task => task.Result.RequestUnitsConsumed),
            SuccessfulDocuments = this.Tasks.Count(task => task.Result.IsSuccessful),
            Failures = this.Tasks.Where(task => !task.Result.IsSuccessful).Select(task => (task.Result.Item,
task.Result.CosmosException)).ToList()
        };
    }
}
```

The `ExecuteAsync` method will wait until all operations are completed and you can use it like so:

```
BulkOperationResponse<MyItem> bulkOperationResponse = await bulkOperations.ExecuteAsync();
```

Capture statistics

The previous code waits until all operations are completed and calculates the required statistics. These statistics are similar to that of the bulk executor library's BulkImportResponse.

```
public class BulkOperationResponse<T>
{
    public TimeSpan TotalTimeTaken { get; set; }
    public int SuccessfulDocuments { get; set; } = 0;
    public double TotalRequestUnitsConsumed { get; set; } = 0;

    public IReadOnlyList<(T, Exception)> Failures { get; set; }
}
```

The `BulkOperationResponse` contains:

1. The total time taken to process the list of operations through bulk support.
2. The number of successful operations.
3. The total of request units consumed.
4. If there are failures, it displays a list of tuples that contain the exception and the associated item for logging and identification purpose.

Retry configuration

Bulk executor library had guidance that mentioned to set the `MaxRetryWaitTimeInSeconds` and `MaxRetryAttemptsOnThrottledRequests` of RetryOptions to `0` to delegate control to the library.

For bulk support in the .NET SDK, there is no hidden behavior. You can configure the retry options directly through the CosmosClientOptions.MaxRetryAttemptsOnRateLimitedRequests and CosmosClientOptions.MaxRetryWaitTimeOnRateLimitedRequests.

> **NOTE**
> In cases where the provisioned request units is much lower than the expected based on the amount of data, you might want to consider setting these to high values. The bulk operation will take longer but it has a higher chance of completely succeeding due to the higher retries.

Performance improvements

As with other operations with the .NET SDK, using the stream APIs results in better performance and avoids any unnecessary serialization.

Using stream APIs is only possible if the nature of the data you use matches that of a stream of bytes (for example, file streams). In such cases, using the `CreateItemStreamAsync`, `ReplaceItemStreamAsync`, or `DeleteItemStreamAsync` methods and working with `ResponseMessage` (instead of `ItemResponse`) increases the throughput that can be achieved.

Next steps

- To learn more about the .NET SDK releases, see the Azure Cosmos DB SDK article.
- Get the complete migration source code from GitHub.
- Additional bulk samples on GitHub

Use bulk executor Java library to perform bulk operations on Azure Cosmos DB data

11/2/2020 • 8 minutes to read • Edit Online

APPLIES TO: SQL API

This tutorial provides instructions on using the Azure Cosmos DB's bulk executor Java library to import, and update Azure Cosmos DB documents. To learn about bulk executor library and how it helps you leverage massive throughput and storage, see bulk executor Library overview article. In this tutorial, you build a Java application that generates random documents and they are bulk imported into an Azure Cosmos container. After importing, you will bulk update some properties of a document.

Currently, the bulk executor library is supported only by Azure Cosmos DB SQL API and Gremlin API accounts. This article describes how to use bulk executor Java library with SQL API accounts. To learn about using bulk executor .NET library with Gremlin API, see perform bulk operations in Azure Cosmos DB Gremlin API. The bulk executor library described is available is only available for the Azure Cosmos DB Java sync SDK v2 and it is the current recommended solution for Java bulk support. It is currently not available for the 3.x, 4.x or other higher SDK versions.

Prerequisites

- If you don't have an Azure subscription, create a free account before you begin.
- You can try Azure Cosmos DB for free without an Azure subscription, free of charge and commitments. Or, you can use the Azure Cosmos DB Emulator with the `https://localhost:8081` endpoint. The Primary Key is provided in Authenticating requests.
- Java Development Kit (JDK) 1.7+
  - On Ubuntu, run `apt-get install default-jdk` to install the JDK.
  - Be sure to set the JAVA_HOME environment variable to point to the folder where the JDK is installed.
- Download and install a Maven binary archive
  - On Ubuntu, you can run `apt-get install maven` to install Maven.
- Create an Azure Cosmos DB SQL API account by using the steps described in the create database account section of the Java quickstart article.

Clone the sample application

Now let's switch to working with code by downloading a sample Java application from GitHub. This application performs bulk operations on Azure Cosmos DB data. To clone the application, open a command prompt, navigate to the directory where you want to copy the application and run the following command:

```
git clone https://github.com/Azure/azure-cosmosdb-bulkexecutor-java-getting-started.git
```

The cloned repository contains two samples "bulkimport" and "bulkupdate" relative to the "\azure-cosmosdb-bulkexecutor-java-getting-started\samples\bulkexecutor-sample\src\main\java\com\microsoft\azure\cosmosdb\bulkexecutor" folder. The "bulkimport" application generates random documents and imports them to Azure Cosmos DB. The "bulkupdate" application updates some

documents in Azure Cosmos DB. In the next sections, we will review the code in each of these sample apps.

Bulk import data to Azure Cosmos DB

1. The Azure Cosmos DB's connection strings are read as arguments and assigned to variables defined in CmdLineConfiguration.java file.

2. Next the DocumentClient object is initialized by using the following statements:

```
ConnectionPolicy connectionPolicy = new ConnectionPolicy();
connectionPolicy.setMaxPoolSize(1000);
DocumentClient client = new DocumentClient(
   HOST,
   MASTER_KEY,
   connectionPolicy,
   ConsistencyLevel.Session)
```

3. The DocumentBulkExecutor object is initialized with a high retry values for wait time and throttled requests. And then they are set to 0 to pass congestion control to DocumentBulkExecutor for its lifetime.

```
// Set client's retry options high for initialization
client.getConnectionPolicy().getRetryOptions().setMaxRetryWaitTimeInSeconds(30);
client.getConnectionPolicy().getRetryOptions().setMaxRetryAttemptsOnThrottledRequests(9);

// Builder pattern
Builder bulkExecutorBuilder = DocumentBulkExecutor.builder().from(
  client,
  DATABASE_NAME,
  COLLECTION_NAME,
  collection.getPartitionKey(),
  offerThroughput) // throughput you want to allocate for bulk import out of the container's total
throughput

// Instantiate DocumentBulkExecutor
DocumentBulkExecutor bulkExecutor = bulkExecutorBuilder.build()

// Set retries to 0 to pass complete control to bulk executor
client.getConnectionPolicy().getRetryOptions().setMaxRetryWaitTimeInSeconds(0);
client.getConnectionPolicy().getRetryOptions().setMaxRetryAttemptsOnThrottledRequests(0);
```

4. Call the importAll API that generates random documents to bulk import into an Azure Cosmos container. You can configure the command line configurations within the CmdLineConfiguration.java file.

```
BulkImportResponse bulkImportResponse = bulkExecutor.importAll(documents, false, true, null);
```

The bulk import API accepts a collection of JSON-serialized documents and it has the following syntax, for more details, see the API documentation:

```
public BulkImportResponse importAll(
     Collection<String> documents,
     boolean isUpsert,
     boolean disableAutomaticIdGeneration,
     Integer maxConcurrencyPerPartitionRange) throws DocumentClientException;
```

The importAll method accepts the following parameters:

PARAMETER	DESCRIPTION
isUpsert	A flag to enable upsert of the documents. If a document with given ID already exists, it's updated.
disableAutomaticIdGeneration	A flag to disable automatic generation of ID. By default, it is set to true.
maxConcurrencyPerPartitionRange	The maximum degree of concurrency per partition key range. The default value is 20.

Bulk import response object definition The result of the bulk import API call contains the following get methods:

PARAMETER	DESCRIPTION
int getNumberOfDocumentsImported()	The total number of documents that were successfully imported out of the documents supplied to the bulk import API call.
double getTotalRequestUnitsConsumed()	The total request units (RU) consumed by the bulk import API call.
Duration getTotalTimeTaken()	The total time taken by the bulk import API call to complete execution.
List<Exception> getErrors()	Gets the list of errors if some documents out of the batch supplied to the bulk import API call failed to get inserted.
List<Object> getBadInputDocuments()	The list of bad-format documents that were not successfully imported in the bulk import API call. User should fix the documents returned and retry import. Bad-formatted documents include documents whose ID value is not a string (null or any other datatype is considered invalid).

5. After you have the bulk import application ready, build the command-line tool from source by using the 'mvn clean package' command. This command generates a jar file in the target folder:

```
mvn clean package
```

6. After the target dependencies are generated, you can invoke the bulk importer application by using the following command:

```
java -Xmx12G -jar bulkexecutor-sample-1.0-SNAPSHOT-jar-with-dependencies.jar -serviceEndpoint *<Fill in
your Azure Cosmos DB's endpoint>*  -masterKey *<Fill in your Azure Cosmos DB's primary key>* -
databaseId bulkImportDb -collectionId bulkImportColl -operation import -shouldCreateCollection -
collectionThroughput 1000000 -partitionKey /profileid -maxConnectionPoolSize 6000 -
numberOfDocumentsForEachCheckpoint 1000000 -numberOfCheckpoints 10
```

The bulk importer creates a new database and a collection with the database name, collection name, and throughput values specified in the App.config file.

Bulk update data in Azure Cosmos DB

You can update existing documents by using the BulkUpdateAsync API. In this example, you will set the Name field to a new value and remove the Description field from the existing documents. For the full set of supported field update operations, see API documentation.

1. Defines the update items along with corresponding field update operations. In this example, you will use SetUpdateOperation to update the Name field and UnsetUpdateOperation to remove the Description field from all the documents. You can also perform other operations like increment a document field by a specific value, push specific values into an array field, or remove a specific value from an array field. To learn about different methods provided by the bulk update API, see the API documentation.

```
SetUpdateOperation<String> nameUpdate = new SetUpdateOperation<>("Name","UpdatedDocValue");
UnsetUpdateOperation descriptionUpdate = new UnsetUpdateOperation("description");

ArrayList<UpdateOperationBase> updateOperations = new ArrayList<>();
updateOperations.add(nameUpdate);
updateOperations.add(descriptionUpdate);

List<UpdateItem> updateItems = new ArrayList<>(cfg.getNumberOfDocumentsForEachCheckpoint());
IntStream.range(0, cfg.getNumberOfDocumentsForEachCheckpoint()).mapToObj(j -> {
 return new UpdateItem(Long.toString(prefix + j), Long.toString(prefix + j), updateOperations);
 }).collect(Collectors.toCollection(() -> updateItems));
```

2. Call the updateAll API that generates random documents to be then bulk imported into an Azure Cosmos container. You can configure the command-line configurations to be passed in CmdLineConfiguration.java file.

```
BulkUpdateResponse bulkUpdateResponse = bulkExecutor.updateAll(updateItems, null)
```

The bulk update API accepts a collection of items to be updated. Each update item specifies the list of field update operations to be performed on a document identified by an ID and a partition key value. for more details, see the API documentation:

```
public BulkUpdateResponse updateAll(
     Collection<UpdateItem> updateItems,
     Integer maxConcurrencyPerPartitionRange) throws DocumentClientException;
```

The updateAll method accepts the following parameters:

PARAMETER	DESCRIPTION
maxConcurrencyPerPartitionRange	The maximum degree of concurrency per partition key range. The default value is 20.

Bulk import response object definition The result of the bulk import API call contains the following get methods:

PARAMETER	DESCRIPTION
int getNumberOfDocumentsUpdated()	The total number of documents that were successfully updated out of the documents supplied to the bulk update API call.
double getTotalRequestUnitsConsumed()	The total request units (RU) consumed by the bulk update API call.

PARAMETER	DESCRIPTION
Duration getTotalTimeTaken()	The total time taken by the bulk update API call to complete execution.
List<Exception> getErrors()	Gets the list of operational or networking issues related to the update operation.
List<BulkUpdateFailure> getFailedUpdates()	Gets the list of updates which could not be completed along with the specific exceptions leading to the failures.

3. After you have the bulk update application ready, build the command-line tool from source by using the 'mvn clean package' command. This command generates a jar file in the target folder:

```
mvn clean package
```

4. After the target dependencies are generated, you can invoke the bulk update application by using the following command:

```
java -Xmx12G -jar bulkexecutor-sample-1.0-SNAPSHOT-jar-with-dependencies.jar -serviceEndpoint **<Fill
in your Azure Cosmos DB's endpoint>* -masterKey **<Fill in your Azure Cosmos DB's primary key>* -
databaseId bulkUpdateDb -collectionId bulkUpdateColl -operation update -collectionThroughput 1000000 -
partitionKey /profileid -maxConnectionPoolSize 6000 -numberOfDocumentsForEachCheckpoint 1000000 -
numberOfCheckpoints 10
```

Performance tips

Consider the following points for better performance when using bulk executor library:

- For best performance, run your application from an Azure VM in the same region as your Cosmos DB account write region.
- For achieving higher throughput:
  - Set the JVM's heap size to a large enough number to avoid any memory issue in handling large number of documents. Suggested heap size: max(3GB, 3 * sizeof(all documents passed to bulk import API in one batch)).
  - There is a preprocessing time, due to which you will get higher throughput when performing bulk operations with a large number of documents. So, if you want to import 10,000,000 documents, running bulk import 10 times on 10 bulk of documents each of size 1,000,000 is preferable than running bulk import 100 times on 100 bulk of documents each of size 100,000 documents.
- It is recommended to instantiate a single DocumentBulkExecutor object for the entire application within a single virtual machine that corresponds to a specific Azure Cosmos container.
- Since a single bulk operation API execution consumes a large chunk of the client machine's CPU and network IO. This happens by spawning multiple tasks internally, avoid spawning multiple concurrent tasks within your application process each executing bulk operation API calls. If a single bulk operation API call running on a single virtual machine is unable to consume your entire container's throughput (if your container's throughput > 1 million RU/s), it's preferable to create separate virtual machines to concurrently execute bulk operation API calls.

Next steps

- To learn about maven package details and release notes of bulk executor Java library, seebulk executor SDK

details.

Visualize Azure Cosmos DB data by using the Power BI connector

12/4/2020 • 8 minutes to read • Edit Online

APPLIES TO: SQL API

Power BI is an online service where you can create and share dashboards and reports. Power BI Desktop is a report authoring tool that enables you to retrieve data from various data sources. Azure Cosmos DB is one of the data source that you can use with Power BI Desktop. You can connect Power BI Desktop to Azure Cosmos DB account with the Azure Cosmos DB connector for Power BI. After you import Azure Cosmos DB data to Power BI, you can transform it, create reports, and publish the reports to Power BI.

This article describes the steps required to connect Azure Cosmos DB account to Power BI Desktop. After connecting, you navigate to a collection, extract the data, transform the JSON data into tabular format, and publish a report to Power BI.

> **NOTE**
> The Power BI connector for Azure Cosmos DB connects to Power BI Desktop. Reports created in Power BI Desktop can be published to PowerBI.com. Direct extraction of Azure Cosmos DB data cannot be performed from PowerBI.com.

> **NOTE**
> Connecting to Azure Cosmos DB with the Power BI connector is currently supported for Azure Cosmos DB SQL API and Gremlin API accounts only.

Prerequisites

Before following the instructions in this Power BI tutorial, ensure that you have access to the following resources:

- Download the latest version of Power BI Desktop.
- Download the sample volcano data from GitHub.
- Create an Azure Cosmos database account and import the volcano data by using the Azure Cosmos DB data migration tool. When importing data, consider the following settings for the source and destinations in the data migration tool:
  - **Source parameters**
    - **Import from:** JSON file(s)
  - **Target parameters**
    - **Connection string:**
      ```
      AccountEndpoint=<Your_account_endpoint>;AccountKey=
      <Your_primary_or_secondary_key>;Database= <Your_database_name>
      ```
    - **Partition key:** /Country
    - **Collection Throughput:** 1000

To share your reports in PowerBI.com, you must have an account in PowerBI.com. To learn more about Power BI

and Power BI Pro, see https://powerbi.microsoft.com/pricing.

Let's get started

In this tutorial, let's imagine that you are a geologist studying volcanoes around the world. The volcano data is stored in an Azure Cosmos DB account and the JSON document format is as follows:

```
{
    "Volcano Name": "Rainier",
        "Country": "United States",
        "Region": "US-Washington",
        "Location": {
          "type": "Point",
          "coordinates": [
            -121.758,
            46.87
          ]
        },
        "Elevation": 4392,
        "Type": "Stratovolcano",
        "Status": "Dendrochronology",
        "Last Known Eruption": "Last known eruption from 1800-1899, inclusive"
}
```

You will retrieve the volcano data from the Azure Cosmos DB account and visualize data in an interactive Power BI report.

1. Run Power BI Desktop.

2. You can **Get Data**, see **Recent Sources**, or **Open Other Reports** directly from the welcome screen. Select the "X" at the top right corner to close the screen. The **Report** view of Power BI Desktop is displayed.

3. Select the **Home** ribbon, then click on **Get Data**. The **Get Data** window should appear.

4. Click on **Azure**, select **Azure Cosmos DB (Beta)**, and then click **Connect**.

5. On the **Preview Connector** page, click **Continue**. The **Azure Cosmos DB** window appears.

6. Specify the Azure Cosmos DB account endpoint URL you would like to retrieve the data from as shown below, and then click **OK**. To use your own account, you can retrieve the URL from the URI box in the **Keys** blade of the Azure portal. Optionally you can provide the database name, collection name or use the navigator to select the database and collection to identify where the data comes from.

7. If you are connecting to this endpoint for the first time, you are prompted for the account key. For your own account, retrieve the key from the **Primary Key** box in the **Read-only Keys** blade of the Azure portal. Enter the appropriate key and then click **Connect**.

   We recommend that you use the read-only key when building reports. This prevents unnecessary exposure of the primary key to potential security risks. The read-only key is available from the **Keys** blade of the Azure portal.

8. When the account is successfully connected, the **Navigator** pane appears. The **Navigator** shows a list of databases under the account.

9. Click and expand on the database where the data for the report comes from, select **volcanodb** (your database name can be different).

10. Now, select a collection that contains the data to retrieve, select **volcano1** (your collection name can be different).

    The Preview pane shows a list of **Record** items. A Document is represented as a **Record** type in Power BI. Similarly, a nested JSON block inside a document is also a **Record**.

11. Click **Edit** to launch the Query Editor in a new window to transform the data.

Flattening and transforming JSON documents

1. Switch to the Power BI Query Editor window, where the **Document** column in the center pane.

2. Click on the expander at the right side of the **Document** column header. The context menu with a list of fields will appear. Select the fields you need for your report, for instance, Volcano Name, Country, Region, Location, Elevation, Type, Status and Last Know Eruption. Uncheck the **Use original column name as prefix** box, and then click **OK**.

3. The center pane displays a preview of the result with the fields selected.

4. In our example, the Location property is a GeoJSON block in a document. As you can see, Location is represented as a **Record** type in Power BI Desktop.

5. Click on the expander at the right side of the Document.Location column header. The context menu with type and coordinates fields appear. Let's select the coordinates field, ensure **Use original column name as prefix** is not selected, and click **OK**.

6. The center pane now shows a coordinates column of **List** type. As shown at the beginning of the tutorial, the GeoJSON data in this tutorial is of Point type with Latitude and Longitude values recorded in the coordinates array.

   The coordinates[0] element represents Longitude while coordinates[1] represents Latitude.

7. To flatten the coordinates array, create a **Custom Column** called LatLong. Select the **Add Column** ribbon and click on **Custom Column**. The **Custom Column** window appears.

8. Provide a name for the new column, e.g. LatLong.

9. Next, specify the custom formula for the new column. For our example, we will concatenate the Latitude and Longitude values separated by a comma as shown below using the following formula: `Text.From([coordinates]{1})&","&Text.From([coordinates]{0})`. Click **OK**.

   For more information on Data Analysis Expressions (DAX) including DAX functions, please visit DAX Basics in Power BI Desktop.

10. Now, the center pane shows the new LatLong columns populated with the values.

If you receive an Error in the new column, make sure that the applied steps under Query Settings match the following figure:

If your steps are different, delete the extra steps and try adding the custom column again.

11. Click **Close and Apply** to save the data model.

Build the reports

Power BI Desktop Report view is where you can start creating reports to visualize data. You can create reports by dragging and dropping fields into the **Report** canvas.

In the Report view, you should find:

1. The **Fields** pane, this is where you can see a list of data models with fields you can use for your reports.
2. The **Visualizations** pane. A report can contain a single or multiple visualizations. Pick the visual types fitting your needs from the **Visualizations** pane.
3. The **Report** canvas, this is where you build the visuals for your report.
4. The **Report** page. You can add multiple report pages in Power BI Desktop.

The following shows the basic steps of creating a simple interactive Map view report.

1. For our example, we will create a map view showing the location of each volcano. In the **Visualizations** pane, click on the Map visual type as highlighted in the screenshot above. You should see the Map visual type painted on the **Report** canvas. The **Visualization** pane should also display a set of properties related to the Map visual type.

2. Now, drag and drop the LatLong field from the **Fields** pane to the **Location** property in **Visualizations** pane.

3. Next, drag and drop the Volcano Name field to the **Legend** property.

4. Then, drag and drop the Elevation field to the **Size** property.

5. You should now see the Map visual showing a set of bubbles indicating the location of each volcano with the size of the bubble correlating to the elevation of the volcano.

6. You now have created a basic report. You can further customize the report by adding more visualizations. In our case, we added a Volcano Type slicer to make the report interactive.

7. On the File menu, click **Save** and save the file as PowerBITutorial.pbix.

Publish and share your report

To share your report, you must have an account in PowerBI.com.

1. In the Power BI Desktop, click on the **Home** ribbon.

2. Click **Publish**. You are be prompted to enter the user name and password for your PowerBI.com account.

3. Once the credential has been authenticated, the report is published to your destination you selected.

4. Click **Open 'PowerBITutorial.pbix' in Power BI** to see and share your report on PowerBI.com.

Create a dashboard in PowerBI.com

Now that you have a report, lets share it on PowerBI.com

When you publish your report from Power BI Desktop to PowerBI.com, it generates a **Report** and a **Dataset** in your PowerBI.com tenant. For example, after you published a report called **PowerBITutorial** to PowerBI.com, you will see PowerBITutorial in both the **Reports** and **Datasets** sections on PowerBI.com.

To create a sharable dashboard, click the **Pin Live Page** button on your PowerBI.com report.

Then follow the instructions in Pin a tile from a report to create a new dashboard.

You can also do ad hoc modifications to report before creating a dashboard. However, it's recommended that you use Power BI Desktop to perform the modifications and republish the report to PowerBI.com.

Next steps

- To learn more about Power BI, see Get started with Power BI.
- To learn more about Azure Cosmos DB, see the Azure Cosmos DB documentation landing page.

Create a real-time dashboard using Azure Cosmos DB and Power BI

12/4/2020 • 7 minutes to read • Edit Online

APPLIES TO: SQL API

This article describes the steps required to create a live weather dashboard in Power BI using Azure Cosmos DB and Azure Analysis Services. The Power BI dashboard will display charts to show real-time information about temperature and rainfall in a region.

Reporting scenarios

There are multiple ways to set up reporting dashboards on data stored in Azure Cosmos DB. Depending on the staleness requirements and the size of the data, the following table describes the reporting setup for each scenario:

SCENARIO	SETUP
1. Generating ad-hoc reports (no refresh)	Power BI Azure Cosmos DB connector with import mode
2. Generating ad-hoc reports with periodic refresh	Power BI Azure Cosmos DB connector with import mode (Scheduled periodic refresh)
3. Reporting on large data sets (< 10 GB)	Power BI Azure Cosmos DB connector with incremental refresh
4. Reporting real time on large data sets	Power BI Azure Analysis Services connector with direct query + Azure Analysis Services (Azure Cosmos DB connector)
5. Reporting on live data with aggregates	Power BI Spark connector with direct query + Azure Databricks + Cosmos DB Spark connector.
6. Reporting on live data with aggregates on large data sets	Power BI Azure Analysis Services connector with direct query + Azure Analysis Services + Azure Databricks + Cosmos DB Spark connector.

Scenarios 1 and 2 can be easily set up using the Azure Cosmos DB Power BI connector. This article describes the setups for scenarios 3 and 4.

Power BI with incremental refresh

Power BI has a mode where incremental refresh can be configured. This mode eliminates the need to create and manage Azure Analysis Services partitions. Incremental refresh can be set up to filter only the latest updates in large datasets. However, this mode works only with Power BI Premium service that has a dataset limitation of 10 GB.

Power BI Azure Analysis connector + Azure Analysis Services

Azure Analysis Services provides a fully managed platform as a service that hosts enterprise-grade data models in the cloud. Massive data sets can be loaded from Azure Cosmos DB into Azure Analysis Services. To avoid querying the entire dataset all the time, the datasets can be subdivided into Azure Analysis Services partitions, which can be refreshed independently at different frequencies.

Power BI incremental refresh

Ingest weather data into Azure Cosmos DB

Set up an ingestion pipeline to load weather data to Azure Cosmos DB. You can set up an Azure Data Factory (ADF) job to periodically load the latest weather data into Azure Cosmos DB using the HTTP Source and Cosmos DB sink.

Connect Power BI to Azure Cosmos DB

1. **Connect Azure Cosmos account to Power BI** - Open the Power BI Desktop and use the Azure Cosmos DB connector to select the right database and container.

2. **Configure incremental refresh** - Follow the steps in incremental refresh with Power BI article to configure incremental refresh for the dataset. Add the **RangeStart** and **RangeEnd** parameters as shown in the following screenshot:

Since the dataset has a Date column that is in text form, the **RangeStart** and **RangeEnd** parameters should be transformed to use the following filter. In the **Advanced Editor** pane, modify your query add the following text to filter the rows based on the RangeStart and RangeEnd parameters:

```
#"Filtered Rows" = Table.SelectRows(#"Expanded Document", each [Document.date] >
DateTime.ToText(RangeStart,"yyyy-MM-dd") and [Document.date] < DateTime.ToText(RangeEnd,"yyyy-MM-dd"))
```

Depending on which column and data type is present in the source dataset, you can change the RangeStart and RangeEnd fields accordingly

PROPERTY	DATA TYPE	FILTER
_ts	Numeric	[_ts] > Duration.TotalSeconds(RangeStart - #datetime(1970, 1, 1, 0, 0, 0)) and [_ts] < Duration.TotalSeconds(RangeEnd - #datetime(1970, 1, 1, 0, 0, 0)))

PROPERTY	DATA TYPE	FILTER
Date (for example:- 2019-08-19)	String	[Document.date]> DateTime.ToText(RangeStart,"yyyy-MM-dd") and [Document.date] < DateTime.ToText(RangeEnd,"yyyy-MM-dd")
Date (for example:- 2019-08-11 12:00:00)	String	[Document.date]> DateTime.ToText(RangeStart," yyyy-mm-dd HH:mm:ss") and [Document.date] < DateTime.ToText(RangeEnd,"yyyy-mm-dd HH:mm:ss")

3. **Define the refresh policy** - Define the refresh policy by navigating to the **Incremental refresh** tab on the **context** menu for the table. Set the refresh policy to refresh **every day** and store the last month data.

Ignore the warning that says *the M query cannot be confirmed to be folded*. The Azure Cosmos DB connector folds filter queries.

4. **Load the data and generate the reports** - By using the data you have loaded earlier, create the charts to report on temperature and rainfall.

5. **Publish the report to Power BI premium** - Since incremental refresh is a Premium only feature, the publish dialog only allows selection of a workspace on Premium capacity. The first refresh may take longer to import the historical data. Subsequent data refreshes are much quicker because they use incremental refresh.

Power BI Azure Analysis connector + Azure Analysis Services

Ingest weather data into Azure Cosmos DB

Set up an ingestion pipeline to load weather data to Azure Cosmos DB. You can set up an Azure Data Factory(ADF) job to periodically load the latest weather data into Azure Cosmos DB using the HTTP Source and Cosmos DB Sink.

Connect Azure Analysis Services to Azure Cosmos account

1. **Create a new Azure Analysis Services cluster** - Create an instance of Azure Analysis services in the same region as the Azure Cosmos account and the Databricks cluster.

2. **Create a new Analysis Services Tabular Project in Visual Studio** - Install the SQL Server Data Tools (SSDT) and create an Analysis Services Tabular project in Visual Studio.

Choose the **Integrated Workspace** instance and the set the Compatibility Level to **SQL Server 2017 / Azure Analysis Services (1400)**

3. **Add the Azure Cosmos DB data source** - Navigate to **Models** > **Data Sources** > **New Data Source** and add the Azure Cosmos DB data source as shown in the following screenshot:

Connect to Azure Cosmos DB by providing the **account URI**, **database name**, and the **container name**. You can now see the data from Azure Cosmos container is imported into Power BI.

= Table.ExpandRecordColumn(Source, "Document",

	Document.freezingfog	Document.temperaturemax	Document.blowingsnow	Document.freezingrain	Document.rain
1	No	59	No	No	No
2	No	89.1	No	No	No
3	No	80.1	No	No	No
4	No	55	No	No	No
5	No	68	No	No	No
6	No	95	No	No	No
7	No	53.1	No	No	No
8	No	60.1	No	No	No
9	No	84	No	No	No
10	No	86	No	No	No
11	No	61	No	No	No
12	No	48	No	No	No
13	No	88	No	No	No
14	No	87.1	No	No	No
15	No	89.1	No	No	No
16	No	75.9	No	No	No
17	No	57.9	No	No	Yes

4. **Construct the Analysis Services model** - Open the query editor, perform the required operations to optimize the loaded data set:

   - Extract only the weather-related columns (temperature and rainfall)

- Extract the month information from the table. This data is useful in creating partitions as described in the next section.
- Convert the temperature columns to number

The resulting M expression is as follows:

```
 let
     Source=#"DocumentDB/https://[ACCOUNTNAME].documents.azure.com:443/",
     #"Expanded Document" = Table.ExpandRecordColumn(Source, "Document", {"id", "_rid", "_self",
"_etag", "fogground", "snowfall", "dust", "snowdepth", "mist", "drizzle", "hail",
"fastest2minwindspeed", "thunder", "glaze", "snow", "ice", "fog", "temperaturemin",
"fastest5secwindspeed", "freezingfog", "temperaturemax", "blowingsnow", "freezingrain", "rain",
"highwind", "date", "precipitation", "fogheavy", "smokehaze", "avgwindspeed", "fastest2minwinddir",
"fastest5secwinddir", "_attachments", "_ts"}, {"Document.id", "Document._rid", "Document._self",
"Document._etag", "Document.fogground", "Document.snowfall", "Document.dust", "Document.snowdepth",
"Document.mist", "Document.drizzle", "Document.hail", "Document.fastest2minwindspeed",
"Document.thunder", "Document.glaze", "Document.snow", "Document.ice", "Document.fog",
"Document.temperaturemin", "Document.fastest5secwindspeed", "Document.freezingfog",
"Document.temperaturemax", "Document.blowingsnow", "Document.freezingrain", "Document.rain",
"Document.highwind", "Document.date", "Document.precipitation", "Document.fogheavy",
"Document.smokehaze", "Document.avgwindspeed", "Document.fastest2minwinddir",
"Document.fastest5secwinddir", "Document._attachments", "Document._ts"}),
     #"Select Columns" = Table.SelectColumns(#"Expanded Document",{"Document.temperaturemin",
"Document.temperaturemax", "Document.rain", "Document.date"}),
     #"Duplicated Column" = Table.DuplicateColumn(#"Select Columns", "Document.date", "Document.month"),
     #"Extracted First Characters" = Table.TransformColumns(#"Duplicated Column", {{"Document.month",
each Text.Start(_, 7), type text}}),
     #"Sorted Rows" = Table.Sort(#"Extracted First Characters",{{"Document.date", Order.Ascending}}),
     #"Changed Type" = Table.TransformColumnTypes(#"Sorted Rows",{{"Document.temperaturemin", type
number}, {"Document.temperaturemax", type number}}),
     #"Filtered Rows" = Table.SelectRows(#"Changed Type", each [Document.month] = "2019-07")
 in
     #"Filtered Rows"
```

Additionally, change the data type of the temperature columns to Decimal to make sure that these values can be plotted in Power BI.

5. **Create Azure Analysis partitions** - Create partitions in Azure Analysis Services to divide the dataset into logical partitions that can be refreshed independently and at different frequencies. In this example, you create two partitions that would divide the dataset into the most recent month's data and everything else.

Create the following two partitions in Azure Analysis Services:

- **Latest Month** - `#"Filtered Rows" = Table.SelectRows(#"Sorted Rows", each [Document.month] = "2019-07")`
- **Historical** - `#"Filtered Rows" = Table.SelectRows(#"Sorted Rows", each [Document.month] <> "2019-07")`

6. **Deploy the Model to the Azure Analysis Server** - Right click on the Azure Analysis Services project and choose **Deploy**. Add the server name in the **Deployment Server properties** pane.

7. **Configure partition refreshes and merges** - Azure Analysis Services allows independent processing of partitions. Since we want the **Latest Month** partition to be constantly updated with the most recent data, set the refresh interval to 5 minutes. You can refresh the data by using the REST API, Azure automation, or

with a Logic App. It's not required to refresh the data in historical partition. Additionally, you need to write some code to consolidate the latest month partition to the historical partition and create a new latest month partition.

Connect Power BI to Analysis Services

1. **Connect to the Azure Analysis Server using the Azure Analysis Services database Connector** - Choose the **Live mode** and connect to the Azure Analysis Services instance as shown in the following screenshot:

2. **Load the data and generate reports** - By using the data you have loaded earlier, create charts to report on temperature and rainfall. Since you are creating a live connection, the queries should be executed on the data in the Azure Analysis Services model that you have deployed in the previous step. The temperature charts will be updated within five minutes after the new data is loaded into Azure Cosmos DB.

Next steps

- To learn more about Power BI, see Get started with Power BI.
- Connect Qlik Sense to Azure Cosmos DB and visualize your data

Connect Qlik Sense to Azure Cosmos DB and visualize your data

12/4/2020 • 3 minutes to read • Edit Online

APPLIES TO: SQL API Azure Cosmos DB API for MongoDB

Qlik Sense is a data visualization tool that combines data from different sources into a single view. Qlik Sense indexes every possible relationship in your data so that you can gain immediate insights to the data. You can visualize Azure Cosmos DB data by using Qlik Sense. This article describes the steps required to connect Azure Cosmos DB to Qlik Sense and visualize your data.

> **NOTE**
> Connecting Qlik Sense to Azure Cosmos DB is currently supported for SQL API and Azure Cosmos DB's API for MongoDB accounts only.

You can Connect Qlik Sense to Azure Cosmos DB with:

- Cosmos DB SQL API by using the ODBC connector.
- Azure Cosmos DB's API for MongoDB by using the Qlik Sense MongoDB connector (currently in preview).
- Azure Cosmos DB's API for MongoDB and SQL API by using REST API connector in Qlik Sense.
- Cosmos DB Mongo DB API by using the gRPC connector for Qlik Core. This article describes the details of connecting to the Cosmos DB SQL API by using the ODBC connector.

This article describes the details of connecting to the Cosmos DB SQL API by using the ODBC connector.

Prerequisites

Before following the instructions in this article, ensure that you have the following resources ready:

- Download the Qlik Sense Desktop or set up Qlik Sense in Azure by Installing the Qlik Sense marketplace item.
- Download the video game data, this sample data is in CSV format. You will store this data in a Cosmos DB account and visualize it in Qlik Sense.
- Create an Azure Cosmos DB SQL API account by using the steps described in create an account section of the quickstart article.
- Create a database and a collection – You can use set the collection throughput value to 1000 RU/s.
- Load the sample video game sales data to your Cosmos DB account. You can import the data by using Azure Cosmos DB data migration tool, you can do a sequential or a bulk import of data. It takes around 3-5 minutes for the data to import to the Cosmos DB account.
- Download, install, and configure the ODBC driver by using the steps in the connect to Cosmos DB with ODBC driver article. The video game data is a simple data set and you don't have to edit the schema, just use the default collection-mapping schema.

Connect Qlik Sense to Cosmos DB

1. Open Qlik Sense and select **Create new app**. Provide a name for your app and select **Create**.

2. After the new app is created successfully, select **Open app** and choose **Add data from files and other sources**.

3. From the data sources, select **ODBC** to open the new connection setup window.

4. Switch to **User DSN** and choose the ODBC connection you created earlier. Provide a name for the connection and select **Create**.

5. After you create the connection, you can choose the database, collection where the video game data is located and then preview it.

6. Next select **Add data** to load the data to Qlik Sense. After you load data to Qlik Sense, you can generate insights and perform analysis on the data. You can either use the insights or build your own app exploring the video games sales. The following image shows

Limitations when connecting with ODBC

Cosmos DB is a schema-less distributed database with drivers modeled around developer needs. The ODBC driver requires a database with schema to infer columns, their data types, and other properties. The regular SQL query or the DML syntax with relational capability is not applicable to Cosmos DB SQL API because SQL API is not ANSI SQL. Due to this reason, the SQL statements issued through the ODBC driver are translated into Cosmos DB-specific SQL syntax that doesn't have equivalents for all constructs. To prevent these translation issues, you must apply a schema when setting up the ODBC connection. The connect with ODBC driver article gives you suggestions and methods to help you configure the schema. Make sure to create this mapping for every database/collection within the Cosmos DB account.

Next Steps

If you are using a different visualization tool such as Power BI, you can connect to it by using the instructions in the following doc:

- Visualize Cosmos DB data by using the Power BI connector

Accelerate big data analytics by using the Apache Spark to Azure Cosmos DB connector

12/4/2020 • 5 minutes to read • Edit Online

APPLIES TO: SQL API

You can run Spark jobs with data stored in Azure Cosmos DB using the Cosmos DB Spark connector. Cosmos can be used for batch and stream processing, and as a serving layer for low latency access.

You can use the connector with Azure Databricks or Azure HDInsight, which provide managed Spark clusters on Azure. The following table shows supported Spark versions.

COMPONENT	VERSION
Apache Spark	2.4.x, 2.3.x, 2.2.x, and 2.1.x
Scala	2.11
Azure Databricks runtime version	> 3.4

WARNING

This connector supports the core (SQL) API of Azure Cosmos DB. For Cosmos DB for MongoDB API, use the MongoDB Spark connector. For Cosmos DB Cassandra API, use the Cassandra Spark connector.

Quickstart

- Follow the steps at Get started with the Java SDK to set up a Cosmos DB account, and populate some data.
- Follow the steps at Azure Databricks getting started to set up an Azure Databricks workspace and cluster.
- You can now create new Notebooks, and import the Cosmos DB connector library. Jump to Working with the Cosmos DB connector for details on how to set up your workspace.
- The following section has snippets on how to read and write using the connector.

Batch reads from Cosmos DB

The following snippet shows how to create a Spark DataFrame to read from Cosmos DB in PySpark.

```
# Read Configuration
readConfig = {
    "Endpoint": "https://doctorwho.documents.azure.com:443/",
    "Masterkey": "YOUR-KEY-HERE",
    "Database": "DepartureDelays",
    "Collection": "flights_pcoll",
    "query_custom": "SELECT c.date, c.delay, c.distance, c.origin, c.destination FROM c WHERE c.origin =
'SEA'" // Optional
}

# Connect via azure-cosmosdb-spark to create Spark DataFrame
flights = spark.read.format(
    "com.microsoft.azure.cosmosdb.spark").options(**readConfig).load()
flights.count()
```

And the same code snippet in Scala:

```
// Import Necessary Libraries
import com.microsoft.azure.cosmosdb.spark.schema._
import com.microsoft.azure.cosmosdb.spark._
import com.microsoft.azure.cosmosdb.spark.config.Config

// Read Configuration
val readConfig = Config(Map(
  "Endpoint" -> "https://doctorwho.documents.azure.com:443/",
  "Masterkey" -> "YOUR-KEY-HERE",
  "Database" -> "DepartureDelays",
  "Collection" -> "flights_pcoll",
  "query_custom" -> "SELECT c.date, c.delay, c.distance, c.origin, c.destination FROM c WHERE c.origin =
'SEA'" // Optional
))

// Connect via azure-cosmosdb-spark to create Spark DataFrame
val flights = spark.read.cosmosDB(readConfig)
flights.count()
```

Batch writes to Cosmos DB

The following snippet shows how to write a data frame to Cosmos DB in PySpark.

```
# Write configuration
writeConfig = {
    "Endpoint": "https://doctorwho.documents.azure.com:443/",
    "Masterkey": "YOUR-KEY-HERE",
    "Database": "DepartureDelays",
    "Collection": "flights_fromsea",
    "Upsert": "true"
}

# Write to Cosmos DB from the flights DataFrame
flights.write.mode("overwrite").format("com.microsoft.azure.cosmosdb.spark").options(
    **writeConfig).save()
```

And the same code snippet in Scala:

```
// Write configuration

val writeConfig = Config(Map(
  "Endpoint" -> "https://doctorwho.documents.azure.com:443/",
  "Masterkey" -> "YOUR-KEY-HERE",
  "Database" -> "DepartureDelays",
  "Collection" -> "flights_fromsea",
  "Upsert" -> "true"
))

// Write to Cosmos DB from the flights DataFrame
import org.apache.spark.sql.SaveMode
flights.write.mode(SaveMode.Overwrite).cosmosDB(writeConfig)
```

Streaming reads from Cosmos DB

The following snippet shows how to connect to and read from Azure Cosmos DB Change Feed.

```
# Read Configuration
readConfig = {
    "Endpoint": "https://doctorwho.documents.azure.com:443/",
    "Masterkey": "YOUR-KEY-HERE",
    "Database": "DepartureDelays",
    "Collection": "flights_pcoll",
    "ReadChangeFeed": "true",
    "ChangeFeedQueryName": "Departure-Delays",
    "ChangeFeedStartFromTheBeginning": "false",
    "InferStreamSchema": "true",
    "ChangeFeedCheckpointLocation": "dbfs:/Departure-Delays"
}


# Open a read stream to the Cosmos DB Change Feed via azure-cosmosdb-spark to create Spark DataFrame
changes = (spark
           .readStream
           .format("com.microsoft.azure.cosmosdb.spark.streaming.CosmosDBSourceProvider")
           .options(**readConfig)
           .load())
```

And the same code snippet in Scala:

```
// Import Necessary Libraries
import com.microsoft.azure.cosmosdb.spark.schema._
import com.microsoft.azure.cosmosdb.spark._
import com.microsoft.azure.cosmosdb.spark.config.Config

// Read Configuration
val readConfig = Config(Map(
  "Endpoint" -> "https://doctorwho.documents.azure.com:443/",
  "Masterkey" -> "YOUR-KEY-HERE",
  "Database" -> "DepartureDelays",
  "Collection" -> "flights_pcoll",
  "ReadChangeFeed" -> "true",
  "ChangeFeedQueryName" -> "Departure-Delays",
  "ChangeFeedStartFromTheBeginning" -> "false",
  "InferStreamSchema" -> "true",
  "ChangeFeedCheckpointLocation" -> "dbfs:/Departure-Delays"
))

// Open a read stream to the Cosmos DB Change Feed via azure-cosmosdb-spark to create Spark DataFrame
val df = spark.readStream.format(classOf[CosmosDBSourceProvider].getName).options(readConfig).load()
```

Streaming writes to Cosmos DB

The following snippet shows how to write a data frame to Cosmos DB in PySpark.

```
# Write configuration
writeConfig = {
    "Endpoint": "https://doctorwho.documents.azure.com:443/",
    "Masterkey": "YOUR-KEY-HERE",
    "Database": "DepartureDelays",
    "Collection": "flights_fromsea",
    "Upsert": "true",
    "WritingBatchSize": "500",
    "CheckpointLocation": "/checkpointlocation_write1"
}

# Write to Cosmos DB from the flights DataFrame
changeFeed = (changes
              .writeStream
              .format("com.microsoft.azure.cosmosdb.spark.streaming.CosmosDBSinkProvider")
              .outputMode("append")
              .options(**writeconfig)
              .start())
```

And the same code snippet in Scala:

```
// Write configuration

val writeConfig = Config(Map(
  "Endpoint" -> "https://doctorwho.documents.azure.com:443/",
  "Masterkey" -> "YOUR-KEY-HERE",
  "Database" -> "DepartureDelays",
  "Collection" -> "flights_fromsea",
  "Upsert" -> "true",
  "WritingBatchSize" -> "500",
  "CheckpointLocation" -> "/checkpointlocation_write1"
))

// Write to Cosmos DB from the flights DataFrame
df
.writeStream
.format(classOf[CosmosDBSinkProvider].getName)
.options(writeConfig)
.start()
```

More more snippets and end to end samples, see Jupyter.

Working with the connector

You can build the connector from source in GitHub, or download the uber jars from Maven in the links below.

SPARK	SCALA	LATEST VERSION
2.4.0	2.11	azure-cosmosdb-spark_lkg_version
2.3.0	2.11	azure-cosmosdb-spark_2.3.0_2.11_1.3.3
2.2.0	2.11	azure-cosmosdb-spark_2.2.0_2.11_1.1.1
2.1.0	2.11	azure-cosmosdb-spark_2.1.0_2.11_1.2.2

Using Databricks notebooks

Create a library using your Databricks workspace by following the guidance in the Azure Databricks Guide > Use the Azure Cosmos DB Spark connector

> **NOTE**
> The **Use the Azure Cosmos DB Spark Connector** page currently is not up-to-date. Instead of downloading the six separate jars into six different libraries, you can download the uber jar from maven at azure-cosmosdb-spark_lkg_version and install this one jar/library.

Using spark-cli

To work with the connector using the spark-cli (that is, `spark-shell`, `pyspark`, `spark-submit`), you can use the `--packages` parameter with the connector's maven coordinates.

```
spark-shell --master yarn --packages "com.microsoft.azure:azure-cosmosdb-spark_2.4.0_2.11:1.4.0"
```

Using Jupyter Notebooks

If you're using Jupyter Notebook within HDInsight, you can use spark-magic `%%configure` cell to specify the connector's maven coordinates.

```
{ "name":"Spark-to-Cosmos_DB_Connector",
  "conf": {
    "spark.jars.packages": "com.microsoft.azure:azure-cosmosdb-spark_2.4.0_2.11:1.4.0",
    "spark.jars.excludes": "org.scala-lang:scala-reflect"
   }
   ...
}
```

> Note, the inclusion of the `spark.jars.excludes` is specific to remove potential conflicts between the connector, Apache Spark, and Livy.

Build the connector

Currently, this connector project uses `maven` so to build without dependencies, you can run:

```
mvn clean package
```

Working with our samples

The Cosmos DB Spark GitHub repository has the following sample notebooks and scripts that you can try.

- **On-Time Flight Performance with Spark and Cosmos DB (Seattle)** ipynb | html: Connect Spark to Cosmos DB using HDInsight Jupyter Notebook service to showcase Spark SQL, GraphFrames, and predicting flight delays using ML pipelines.
- **Twitter Source with Apache Spark and Azure Cosmos DB Change Feed**: ipynb | html
- **Using Apache Spark to query Cosmos DB Graphs**: ipynb | html
- **Connecting Azure Databricks to Azure Cosmos DB** using `azure-cosmosdb-spark`. Linked here is also an Azure Databricks version of the On-Time Flight Performance notebook.
- **Lambda Architecture with Azure Cosmos DB and HDInsight (Apache Spark)**: You can reduce the operational overhead of maintaining big data pipelines using Cosmos DB and Spark.

More Information

We have more information in the `azure-cosmosdb-spark` wiki including:

- Azure Cosmos DB Spark Connector User Guide
- Aggregations Examples

Configuration and Setup

- Spark Connector Configuration
- Spark to Cosmos DB Connector Setup (In progress)
- Configuring Power BI Direct Query to Azure Cosmos DB via Apache Spark (HDI)

Troubleshooting

- Using Cosmos DB Aggregates
- Known Issues

Performance

- Performance Tips
- Query Test Runs
- Writing Test Runs

Change Feed

- Stream Processing Changes using Azure Cosmos DB Change Feed and Apache Spark
- Change Feed Demos
- Structured Stream Demos

Monitoring

- Monitoring Spark jobs with application insights

Next steps

If you haven't already, download the Spark to Azure Cosmos DB connector from the azure-cosmosdb-spark GitHub repository. Explore the following additional resources in the repo:

- Aggregations examples
- Sample scripts and notebooks

You might also want to review the Apache Spark SQL, DataFrames, and Datasets Guide, and the Apache Spark on Azure HDInsight article.

Use Apache Spark Structured Streaming with Apache Kafka and Azure Cosmos DB

12/4/2020 • 5 minutes to read • Edit Online

Learn how to use Apache Spark Structured Streaming to read data from Apache Kafka on Azure HDInsight, and then store the data into Azure Cosmos DB.

Azure Cosmos DB is a globally distributed, multi-model database. This example uses a SQL API database model. For more information, see the Welcome to Azure Cosmos DB document.

Spark structured streaming is a stream processing engine built on Spark SQL. It allows you to express streaming computations the same as batch computation on static data. For more information on Structured Streaming, see the Structured Streaming Programming Guide at Apache.org.

> **IMPORTANT**
>
> This example used Spark 2.2 on HDInsight 3.6.
>
> The steps in this document create an Azure resource group that contains both a Spark on HDInsight and a Kafka on HDInsight cluster. These clusters are both located within an Azure Virtual Network, which allows the Spark cluster to directly communicate with the Kafka cluster.
>
> When you are done with the steps in this document, remember to delete the clusters to avoid excess charges.

Create the clusters

Apache Kafka on HDInsight doesn't provide access to the Kafka brokers over the public internet. Anything that talks to Kafka must be in the same Azure virtual network as the nodes in the Kafka cluster. For this example, both the Kafka and Spark clusters are located in an Azure virtual network. The following diagram shows how communication flows between the clusters:

> **NOTE**
>
> The Kafka service is limited to communication within the virtual network. Other services on the cluster, such as SSH and Ambari, can be accessed over the internet. For more information on the public ports available with HDInsight, see Ports and URIs used by HDInsight.

While you can create an Azure virtual network, Kafka, and Spark clusters manually, it's easier to use an Azure Resource Manager template. Use the following steps to deploy an Azure virtual network, Kafka, and Spark clusters to your Azure subscription.

1. Use the following button to sign in to Azure and open the template in the Azure portal.

   Deploy to Azure

   The Azure Resource Manager template is located in the GitHub repository for this project (https://github.com/Azure-Samples/hdinsight-spark-scala-kafka-cosmosdb).

   This template creates the following resources:

   - A Kafka on HDInsight 3.6 cluster.
   - A Spark on HDInsight 3.6 cluster.
   - An Azure Virtual Network, which contains the HDInsight clusters. The virtual network created by the template uses the 10.0.0.0/16 address space.
   - An Azure Cosmos DB SQL API database.

   > **IMPORTANT**
   > The structured streaming notebook used in this example requires Spark on HDInsight 3.6. If you use an earlier version of Spark on HDInsight, you receive errors when using the notebook.

2. Use the following information to populate the entries on the **Custom deployment** section:

PROPERTY	VALUE
Subscription	Select your Azure subscription.
Resource group	Create a group or select an existing one. This group contains the HDInsight cluster.
Cosmos DB Account Name	This value is used as the name for the Cosmos DB account. The name can only contain lowercase letters, numbers, and the hyphen (-) character. It must be between 3-31 characters in length.
Base Cluster Name	This value is used as the base name for the Spark and Kafka clusters. For example, entering **myhdi** creates a Spark cluster named **spark-myhdi** and a Kafka cluster named **kafka-myhdi**.
Cluster Version	The HDInsight cluster version. This example is tested with HDInsight 3.6, and may not work with other cluster types.
Cluster Login User Name	The admin user name for the Spark and Kafka clusters.
Cluster Login Password	The admin user password for the Spark and Kafka clusters.
Ssh User Name	The SSH user to create for the Spark and Kafka clusters.
Ssh Password	The password for the SSH user for the Spark and Kafka clusters.

3. Read the **Terms and Conditions**, and then select **I agree to the terms and conditions stated above**.

4. Finally, select **Purchase**. It may take up to 45 minutes to create the clusters, virtual network, and Cosmos DB account.

Create the Cosmos DB database and collection

The project used in this document stores data in Cosmos DB. Before running the code, you must first create a *database* and *collection* in your Cosmos DB instance. You must also retrieve the document endpoint and the *key* used to authenticate requests to Cosmos DB.

One way to do this is to use the Azure CLI. The following script will create a database named `kafkadata` and a collection named `kafkacollection`. It then returns the primary key.

```
#!/bin/bash

# Replace 'myresourcegroup' with the name of your resource group
resourceGroupName='myresourcegroup'
# Replace 'mycosmosaccount' with the name of your Cosmos DB account name
name='mycosmosaccount'

# WARNING: If you change the databaseName or collectionName
#          then you must update the values in the Jupyter notebook
databaseName='kafkadata'
collectionName='kafkacollection'

# Create the database
az cosmosdb sql database create --account-name $name --name $databaseName --resource-group $resourceGroupName

# Create the collection
az cosmosdb sql container create --account-name $name --database-name $databaseName --name $collectionName --
partition-key-path "/my/path" --resource-group $resourceGroupName

# Get the endpoint
az cosmosdb show --name $name --resource-group $resourceGroupName --query documentEndpoint

# Get the primary key
az cosmosdb keys list --name $name --resource-group $resourceGroupName --type keys
```

The document endpoint and primary key information is similar to the following text:

```
# endpoint
"https://mycosmosaccount.documents.azure.com:443/"
# key
"YqPXw3RP7TsJoBF5imkYR0QNA02IrreNAlkrUMkL8EW94YHs41bktBhIgWq4pqj6HCGYijQKMRkCTsSaKUO2pw=="
```

> **IMPORTANT**
>
> Save the endpoint and key values, as they are needed in the Jupyter Notebooks.

Get the notebooks

The code for the example described in this document is available at https://github.com/Azure-Samples/hdinsight-spark-scala-kafka-cosmosdb.

Upload the notebooks

Use the following steps to upload the notebooks from the project to your Spark on HDInsight cluster:

1. In your web browser, connect to the Jupyter notebook on your Spark cluster. In the following URL, replace `CLUSTERNAME` with the name of your **Spark** cluster:

   ```
   https://CLUSTERNAME.azurehdinsight.net/jupyter
   ```

   When prompted, enter the cluster login (admin) and password used when you created the cluster.

2. From the upper right side of the page, use the **Upload** button to upload the **Stream-taxi-data-to-kafka.ipynb** file to the cluster. Select **Open** to start the upload.

3. Find the **Stream-taxi-data-to-kafka.ipynb** entry in the list of notebooks, and select **Upload** button beside it.

4. Repeat steps 1-3 to load the **Stream-data-from-Kafka-to-Cosmos-DB.ipynb** notebook.

Load taxi data into Kafka

Once the files have been uploaded, select the **Stream-taxi-data-to-kafka.ipynb** entry to open the notebook. Follow the steps in the notebook to load data into Kafka.

Process taxi data using Spark Structured Streaming

From the Jupyter Notebook home page, select the **Stream-data-from-Kafka-to-Cosmos-DB.ipynb** entry. Follow the steps in the notebook to stream data from Kafka and into Azure Cosmos DB using Spark Structured Streaming.

Next steps

Now that you've learned how to use Apache Spark Structured Streaming, see the following documents to learn more about working with Apache Spark, Apache Kafka, and Azure Cosmos DB:

- How to use Apache Spark streaming (DStream) with Apache Kafka.
- Start with Jupyter Notebook and Apache Spark on HDInsight
- Welcome to Azure Cosmos DB

Migrate data to Azure Cosmos DB SQL API account using Striim

12/4/2020 • 8 minutes to read • Edit Online

APPLIES TO: SQL API

The Striim image in the Azure marketplace offers continuous real-time data movement from data warehouses and databases to Azure. While moving the data, you can perform in-line denormalization, data transformation, enable real-time analytics, and data reporting scenarios. It's easy to get started with Striim to continuously move enterprise data to Azure Cosmos DB SQL API. Azure provides a marketplace offering that makes it easy to deploy Striim and migrate data to Azure Cosmos DB.

This article shows how to use Striim to migrate data from an **Oracle database** to an **Azure Cosmos DB SQL API account**.

Prerequisites

- If you don't have an Azure subscription, create a free account before you begin.
- An Oracle database running on-premises with some data in it.

Deploy the Striim marketplace solution

1. Sign into the Azure portal.
2. Select **Create a resource** and search for **Striim** in the Azure marketplace. Select the first option and **Create**.

3. Next, enter the configuration properties of the Striim instance. The Striim environment is deployed in a virtual machine. From the **Basics** pane, enter the **VM user name**, **VM password** (this password is used to SSH into the VM). Select your **Subscription**, **Resource Group**, and **Location details** where you'd like to

deploy Striim. Once complete, select **OK**.

4. In the **Striim Cluster settings** pane, choose the type of Striim deployment and the virtual machine size.

SETTING	VALUE	DESCRIPTION
Striim deployment type	Standalone	Striim can run in a **Standalone** or **Cluster** deployment types. Standalone mode will deploy the Striim server on a single virtual machine and you can select the size of the VMs depending on your data volume. Cluster mode will deploy the Striim server on two or more VMs with the selected size. Cluster environments with more than 2 nodes offer automatic high availability and failover. In this tutorial, you can select Standalone option. Use the default "Standard_F4s" size VM.
Name of the Striim cluster	<Striim_cluster_Name>	Name of the Striim cluster.
Striim cluster password	<Striim_cluster_password>	Password for the cluster.

After you fill the form, select **OK** to continue.

5. In the **Striim access settings** pane, configure the **Public IP address** (choose the default values), **Domain name for Striim**, **Admin password** that you'd like to use to login to the Striim UI. Configure a VNET and Subnet (choose the default values). After filling in the details, select **OK** to continue.

6. Azure will validate the deployment and make sure everything looks good; validation takes few minutes to complete. After the validation is completed, select **OK**.

7. Finally, review the terms of use and select **Create** to create your Striim instance.

Configure the source database

In this section, you configure the Oracle database as the source for data movement. You'll need the Oracle JDBC driver to connect to Oracle. To read changes from your source Oracle database, you can either use the LogMiner or the XStream APIs. The Oracle JDBC driver must be present in Striim's Java classpath to read, write, or persist data from Oracle database.

Download the ojdbc8.jar driver onto your local machine. You will install it in the Striim cluster later.

Configure the target database

In this section, you will configure the Azure Cosmos DB SQL API account as the target for data movement.

1. Create an Azure Cosmos DB SQL API account using the Azure portal.

2. Navigate to the **Data Explorer** pane in your Azure Cosmos account. Select **New Container** to create a new container. Assume that you are migrating *products* and *orders* data from Oracle database to Azure Cosmos DB. Create a new database named **StriimDemo** with a container named **Orders**. Provision the container with **1000 RUs** (this example uses 1000 RUs, but you should use the throughput estimated for your workload), and **/ORDER_ID** as the partition key. These values will differ depending on your source data.

Configure Oracle to Azure Cosmos DB data flow

1. Now, let's get back to Striim. Before interacting with Striim, install the Oracle JDBC driver that you downloaded earlier.

2. Navigate to the Striim instance that you deployed in the Azure portal. Select the **Connect** button in the upper menu bar and from the **SSH** tab, copy the URL in **Login using VM local account** field.

3. Open a new terminal window and run the SSH command you copied from the Azure portal. This article uses terminal in a MacOS, you can follow the similar instructions using PuTTY or a different SSH client on a Windows machine. When prompted, type **yes** to continue and enter the **password** you have set for the virtual machine in the previous step.

```
Striim — striimdemo@striimdemo-masternode:~ — ssh striimdemo@striimdemo.westus.cloud
Striim-Edwards-MacBook-Pro:Striim edwardbell$ ssh striimdemo@striimdemo.westus.cloudapp.azure.com
Password:
Last login: Tue Jul 16 21:04:40 2019 from 208.89.164.26
[striimdemo@striimdemo-masternode ~]$
```

4. Now, open a new terminal tab to copy the **ojdbc8.jar** file you downloaded previously. Use the following SCP command to copy the jar file from your local machine to the tmp folder of the Striim instance running in Azure:

```
cd <Directory_path_where_the_Jar_file_exists>
scp ojdbc8.jar striimdemo@striimdemo.westus.cloudapp.azure.com:/tmp
```

```
Jars — ssh ◂ scp ojdbc8.jar striimdemo@striimdemo.westus.cloudapp.azure.com:/tmp — 134×27
...e:~ — ssh striimdemo@striimdemo.westus.cloudapp.azure.com ...striimdemo@striimdemo.westus.cloudapp.azure.com:/tmp
Last login: Tue Jul 16 14:03:40 on ttys001
Striim-Edwards-MacBook-Pro:Striim edwardbell$ cd /Users/edwardbell/Desktop/Jars
Striim-Edwards-MacBook-Pro:Jars edwardbell$ scp ojdbc8.jar striimdemo@striimdemo.westus.cloudapp.azure.com:/tmp
Password:
ojdbc8.jar                                                    82% 3360KB   2.3MB/s   00:00 ETA
```

5. Next, navigate back to the window where you did SSH to the Striim instance and Login as sudo. Move the **ojdbc8.jar** file from the **/tmp** directory into the **lib** directory of your Striim instance with the following commands:

```
sudo su
cd /tmp
mv ojdbc8.jar /opt/striim/lib
chmod +x ojdbc8.jar
```

```
Striim — root@striimdemo-masternode:/opt/striim/lib — ssh striimdemo@striimdemo.westus.cloudapp.azure.com — 13...
.../lib — ssh striimdemo@striimdemo.westus.cloudapp.azure.com ~/Desktop/Jars — -bash
Striim-Edwards-MacBook-Pro:Striim edwardbell$ ssh striimdemo@striimdemo.westus.cloudapp.azure.com
Password:
Last login: Tue Jul 16 21:05:02 2019 from 208.89.164.26
[striimdemo@striimdemo-masternode ~]$ sudo su
[sudo] password for striimdemo:
[root@striimdemo-masternode striimdemo]# cd /tmp
[root@striimdemo-masternode tmp]# ls
hsperfdata_root  hsperfdata_striim  jna--891986052  ojdbc8.jar
[root@striimdemo-masternode tmp]# mv ojdbc8.jar /opt/striim/lib
[root@striimdemo-masternode tmp]# cd /opt/striim/lib
[root@striimdemo-masternode lib]# chmod +x ojdbc8.jar
```

6. From the same terminal window, restart the Striim server by executing the following commands:

```
Systemctl stop striim-node
Systemctl stop striim-dbms
Systemctl start striim-dbms
Systemctl start striim-node
```

7. Striim will take a minute to start up. If you'd like to see the status, run the following command:

```
tail -f /opt/striim/logs/striim-node.log
```

8. Now, navigate back to Azure and copy the Public IP address of your Striim VM.

9. To navigate to the Striim's Web UI, open a new tab in a browser and copy the public IP followed by: 9080. Sign in by using the **admin** username, along with the admin password you specified in the Azure portal.

10. Now you'll arrive at Striim's home page. There are three different panes – **Dashboards**, **Apps**, and **SourcePreview**. The Dashboards pane allows you to move data in real time and visualize it. The Apps pane contains your streaming data pipelines, or data flows. On the right hand of the page is SourcePreview where you can preview your data before moving it.

11. Select the **Apps** pane, we'll focus on this pane for now. There are a variety of sample apps that you can use to learn about Striim, however in this article you will create our own. Select the **Add App** button in the top

right-hand corner.

12. There are a few different ways to create Striim applications. Select **Start with Template** to start with an existing template.

13. In the **Search templates** field, type "Cosmos" and select **Target: Azure Cosmos DB** and then select **Oracle CDC to Azure Cosmos DB**.

14. In the next page, name your application. You can provide a name such as **oraToCosmosDB** and then select

Save.

15. Next, enter the source configuration of your source Oracle instance. Enter a value for the **Source Name**. The source name is just a naming convention for the Striim application, you can use something like **src_onPremOracle**. Enter values for rest of the source parameters **URL**, **Username**, **Password**, choose **LogMiner** as the reader to read data from Oracle. Select **Next** to continue.

16. Striim will check your environment and make sure that it can connect to your source Oracle instance, have the right privileges, and that CDC was configured properly. Once all the values are validated, select **Next**.

17. Select the tables from Oracle database that you'd like to migrate. For example, let's choose the Orders table and select **Next**.

18. After selecting the source table, you can do more complicated operations such as mapping and filtering. In this case, you will just create a replica of your source table in Azure Cosmos DB. So, select **Next** to configure the target

19. Now, let's configure the target:

    - **Target Name** - Provide a friendly name for the target.
    - **Input From** - From the dropdown list, select the input stream from the one you created in the source Oracle configuration.
    - **Collections**- Enter the target Azure Cosmos DB configuration properties. The collections syntax is **SourceSchema.SourceTable, TargetDatabase.TargetContainer**. In this example, the value would be "SYSTEM.ORDERS, StriimDemo.Orders".
    - **AccessKey** - The PrimaryKey of your Azure Cosmos account.
    - **ServiceEndpoint** – The URI of your Azure Cosmos account, they can be found under the **Keys** section of the Azure portal.

    Select **Save** and **Next**.

20. Next, you'll arrive at the flow designer, where you can drag and drop out of the box connectors to create your streaming applications. You will not make any modifications to the flow at this point. so go ahead and deploy the application by selecting the **Deploy App** button.

21. In the deployment window, you can specify if you want to run certain parts of your application on specific parts of your deployment topology. Since we're running in a simple deployment topology through Azure, we'll use the default option.

22. After deploying, you can preview the stream to see data flowing through. Select the **wave** icon and the eyeball next to it. Select the **Deployed** button in the top menu bar, and select **Start App**.

23. By using a **CDC(Change Data Capture)** reader, Striim will pick up only new changes on the database. If you have data flowing through your source tables, you'll see it. However, since this is a demo table, the source isn't connected to any application. If you use a sample data generator, you can insert a chain of events into your Oracle database.

24. You'll see data flowing through the Striim platform. Striim picks up all the metadata associated with your table as well, which is helpful to monitor the data and make sure that the data lands on the right target.

25. Finally, let's sign into Azure and navigate to your Azure Cosmos account. Refresh the Data Explorer, and you can see that data has arrived.

By using the Striim solution in Azure, you can continuously migrate data to Azure Cosmos DB from various sources such as Oracle, Cassandra, MongoDB, and various others to Azure Cosmos DB. To learn more please visit the Striim website, download a free 30-day trial of Striim, and for any issues when setting up the migration path with Striim, file a support request.

Next steps

- If you are migrating data to Azure Cosmos DB SQL API, see how to migrate data to Cassandra API account using Striim
- Monitor and debug your data with Azure Cosmos DB metrics

Migrate one-to-few relational data into Azure Cosmos DB SQL API account

11/2/2020 • 8 minutes to read • Edit Online

APPLIES TO: SQL API

In order to migrate from a relational database to Azure Cosmos DB SQL API, it can be necessary to make changes to the data model for optimization.

One common transformation is denormalizing data by embedding related subitems within one JSON document. Here we look at a few options for this using Azure Data Factory or Azure Databricks. For general guidance on data modeling for Cosmos DB, please review Data modeling in Azure Cosmos DB.

Example Scenario

Assume we have the following two tables in our SQL database, Orders and OrderDetails.

We want to combine this one-to-few relationship into one JSON document during migration. To do this, we can create a T-SQL query using "FOR JSON" as below:

```
SELECT
  o.OrderID,
  o.OrderDate,
  o.FirstName,
  o.LastName,
  o.Address,
  o.City,
  o.State,
  o.PostalCode,
  o.Country,
  o.Phone,
  o.Total,
  (select OrderDetailId, ProductId, UnitPrice, Quantity from OrderDetails od where od.OrderId = o.OrderId for
json auto) as OrderDetails
FROM Orders o;
```

The results of this query would look as below:

Ideally, you want to use a single Azure Data Factory (ADF) copy activity to query SQL data as the source and write the output directly to Azure Cosmos DB sink as proper JSON objects. Currently, it is not possible to perform the needed JSON transformation in one copy activity. If we try to copy the results of the above query into an Azure Cosmos DB SQL API container, we will see the OrderDetails field as a string property of our document, instead of the expected JSON array.

We can work around this current limitation in one of the following ways:

- **Use Azure Data Factory with two copy activities**:
  1. Get JSON-formatted data from SQL to a text file in an intermediary blob storage location, and
  2. Load data from the JSON text file to a container in Azure Cosmos DB.
- **Use Azure Databricks to read from SQL and write to Azure Cosmos DB** - we will present two options here.

Let's look at these approaches in more detail:

Azure Data Factory

Although we cannot embed OrderDetails as a JSON-array in the destination Cosmos DB document, we can work around the issue by using two separate Copy Activities.

Copy Activity #1: SqlJsonToBlobText

For the source data, we use a SQL query to get the result set as a single column with one JSON object (representing the Order) per row using the SQL Server OPENJSON and FOR JSON PATH capabilities:

```
SELECT [value] FROM OPENJSON(
  (SELECT
    id = o.OrderID,
    o.OrderDate,
    o.FirstName,
    o.LastName,
    o.Address,
    o.City,
    o.State,
    o.PostalCode,
    o.Country,
    o.Phone,
    o.Total,
    (select OrderDetailId, ProductId, UnitPrice, Quantity from OrderDetails od where od.OrderId = o.OrderId for
json auto) as OrderDetails
   FROM Orders o FOR JSON PATH)
)
```

For the sink of the SqlJsonToBlobText copy activity, we choose "Delimited Text" and point it to a specific folder in Azure Blob Storage with a dynamically generated unique file name (for example, '@concat(pipeline().RunId,'.json'). Since our text file is not really "delimited" and we do not want it to be parsed into separate columns using commas and want to preserve the double-quotes ("), we set "Column delimiter" to a Tab ("\t") - or another character not occurring in the data - and "Quote character" to "No quote character".

Copy Activity #2: BlobJsonToCosmos

Next, we modify our ADF pipeline by adding the second Copy Activity that looks in Azure Blob Storage for the text file that was created by the first activity. It processes it as "JSON" source to insert to Cosmos DB sink as one document per JSON-row found in the text file.

Optionally, we also add a "Delete" activity to the pipeline so that it deletes all of the previous files remaining in the /Orders/ folder prior to each run. Our ADF pipeline now looks something like this:

After we trigger the pipeline above, we see a file created in our intermediary Azure Blob Storage location containing one JSON-object per row:

We also see Orders documents with properly embedded OrderDetails inserted into our Cosmos DB collection:

Azure Databricks

We can also use Spark in Azure Databricks to copy the data from our SQL Database source to the Azure Cosmos DB destination without creating the intermediary text/JSON files in Azure Blob Storage.

> **NOTE**
>
> For clarity and simplicity, the code snippets below include dummy database passwords explicitly inline, but you should always use Azure Databricks secrets.

First, we create and attach the required SQL connector and Azure Cosmos DB connector libraries to our Azure Databricks cluster. Restart the cluster to make sure libraries are loaded.

Next, we present two samples, for Scala and Python.

Scala

Here, we get the results of the SQL query with "FOR JSON" output into a DataFrame:

```
// Connect to Azure SQL https://docs.databricks.com/data/data-sources/sql-databases-azure.html
import com.microsoft.azure.sqldb.spark.config.Config
import com.microsoft.azure.sqldb.spark.connect._
val configSql = Config(Map(
  "url"          -> "xxxx.database.windows.net",
  "databaseName" -> "xxxx",
  "queryCustom"  -> "SELECT o.OrderID, o.OrderDate, o.FirstName, o.LastName, o.Address, o.City, o.State,
o.PostalCode, o.Country, o.Phone, o.Total, (SELECT OrderDetailId, ProductId, UnitPrice, Quantity FROM
OrderDetails od WHERE od.OrderId = o.OrderId FOR JSON AUTO) as OrderDetails FROM Orders o",
  "user"         -> "xxxx",
  "password"     -> "xxxx" // NOTE: For clarity and simplicity, this example includes secrets explicitely as a
string, but you should always use Databricks secrets
))
// Create DataFrame from Azure SQL query
val orders = sqlContext.read.sqlDB(configSql)
display(orders)
```

cluster1

Cmd 1

```
// Connect to Azure SQL https://docs.databricks.com/data/data-sources/sql-databases-azure.html
import com.microsoft.azure.sqldb.spark.config.Config
import com.microsoft.azure.sqldb.spark.connect._

val configSql = Config(Map(
  "url"          -> "[redacted].database.windows.net",
  "databaseName" -> "[redacted]",
  "queryCustom"  -> "SELECT o.OrderID, o.OrderDate, o.FirstName, o.LastName, o.Address, o.City, o.State, o.PostalCode, o.Country, o.Phone, o.Total, (SELECT OrderDetailId, ProductId, UnitPrice, Quantity FROM OrderDetails od WHERE od.OrderId = o.OrderId FOR JSON AUTO) as OrderDetails FROM Orders o",
  "user"         -> "[redacted]",
  "password"     -> "[redacted]" // NOTE: For clarity and simplicity, this example includes secrets explicitely as a string, but you should always use Databricks secrets
))

// Create DataFrame from Azure SQL query
val orders = sqlContext.read.sqlDB(configSql)
display(orders)
```

String containing JSON array

(1) Spark Jobs

orders: org.apache.spark.sql.DataFrame = [OrderID: integer, OrderDate: timestamp ... 10 more fields]

ss	City	State	PostalCode	Country	Phone	Total	OrderDetails
in	Cityville	IL	60001	United States	555-555-5555	10	[{"OrderDetailId":10,"ProductId":200,"UnitPrice":3.5000,"Quantity":2},{"OrderDetailId":11,"ProductId":201,"UnitPrice":3.0000,"Quantity":1}]
in	Cityville	IL	60001	United States	555-555-5551	100	[{"OrderDetailId":12,"ProductId":200,"UnitPrice":3.5000,"Quantity":2},{"OrderDetailId":13,"ProductId":202,"UnitPrice":5.0000,"Quantity":15},{"OrderDetailId":14,"ProductId":203,"UnitPrice":9.0000,"Quantity":2}]
in	Cityville	IL	60002	United States	555-555-5552	50	[{"OrderDetailId":15,"ProductId":201,"UnitPrice":3.0000,"Quantity":10},{"OrderDetailId":16,"ProductId":202,"UnitPrice":5.0000,"Quantity":1},{"OrderDetailId":17,"ProductId":203,"UnitPrice":9.0000,"Quantity":1},{"OrderDetailId":18,"ProductId":204,"UnitPrice":2.0000,"Quantity":1},{"OrderDetailId":19,"ProductId":205,"UnitPrice":2.0000,"Quantity":1}

Command took 5.32 seconds -- by [redacted] at 12/10/2019, 1:47:59 PM on cluster1

Next, we connect to our Cosmos DB database and collection:

```
// Connect to Cosmos DB https://docs.databricks.com/data/data-sources/azure/cosmosdb-connector.html
import org.joda.time._
import org.joda.time.format._
import com.microsoft.azure.cosmosdb.spark.schema._
import com.microsoft.azure.cosmosdb.spark.CosmosDBSpark
import com.microsoft.azure.cosmosdb.spark.config.Config
import org.apache.spark.sql.functions._
import org.joda.time._
import org.joda.time.format._
import com.microsoft.azure.cosmosdb.spark.schema._
import com.microsoft.azure.cosmosdb.spark.CosmosDBSpark
import com.microsoft.azure.cosmosdb.spark.config.Config
import org.apache.spark.sql.functions._
val configMap = Map(
  "Endpoint" -> "https://xxxx.documents.azure.com:443/",
  // NOTE: For clarity and simplicity, this example includes secrets explicitely as a string, but you should always use Databricks secrets
  "Masterkey" -> "xxxx",
  "Database" -> "StoreDatabase",
  "Collection" -> "Orders")
val configCosmos = Config(configMap)
```

Finally, we define our schema and use from_json to apply the DataFrame prior to saving it to the CosmosDB collection.

```
// Convert DataFrame to proper nested schema
import org.apache.spark.sql.types._
val orderDetailsSchema = ArrayType(StructType(Array(
    StructField("OrderDetailId",IntegerType,true),
    StructField("ProductId",IntegerType,true),
    StructField("UnitPrice",DoubleType,true),
    StructField("Quantity",IntegerType,true)
  )))
val ordersWithSchema = orders.select(
  col("OrderId").cast("string").as("id"),
  col("OrderDate").cast("string"),
  col("FirstName").cast("string"),
  col("LastName").cast("string"),
  col("Address").cast("string"),
  col("City").cast("string"),
  col("State").cast("string"),
  col("PostalCode").cast("string"),
  col("Country").cast("string"),
  col("Phone").cast("string"),
  col("Total").cast("double"),
  from_json(col("OrderDetails"), orderDetailsSchema).as("OrderDetails")
)
display(ordersWithSchema)
// Save nested data to Cosmos DB
CosmosDBSpark.save(ordersWithSchema, configCosmos)
```

Python

As an alternative approach, you may need to execute JSON transformations in Spark (if the source database does not support "FOR JSON" or a similar operation), or you may wish to use parallel operations for a very large data

set. Here we present a PySpark sample. Start by configuring the source and target database connections in the first cell:

```
import uuid
import pyspark.sql.functions as F
from pyspark.sql.functions import col
from pyspark.sql.types import StringType,DateType,LongType,IntegerType,TimestampType

#JDBC connect details for SQL Server database
jdbcHostname = "jdbcHostname"
jdbcDatabase = "OrdersDB"
jdbcUsername = "jdbcUsername"
jdbcPassword = "jdbcPassword"
jdbcPort = "1433"

connectionProperties = {
  "user" : jdbcUsername,
  "password" : jdbcPassword,
  "driver" : "com.microsoft.sqlserver.jdbc.SQLServerDriver"
}
jdbcUrl = "jdbc:sqlserver://{0}:{1};database={2};user={3};password={4}".format(jdbcHostname, jdbcPort,
jdbcDatabase, jdbcUsername, jdbcPassword)

#Connect details for Target Azure Cosmos DB SQL API account
writeConfig = {
    "Endpoint": "Endpoint",
    "Masterkey": "Masterkey",
    "Database": "OrdersDB",
    "Collection": "Orders",
    "Upsert": "true"
}
```

Then, we will query the source Database (in this case SQL Server) for both the order and order detail records, putting the results into Spark Dataframes. We will also create a list containing all the order IDs, and a Thread pool for parallel operations:

```
import json
import ast
import pyspark.sql.functions as F
import uuid
import numpy as np
import pandas as pd
from functools import reduce
from pyspark.sql import SQLContext
from pyspark.sql.types import *
from pyspark.sql import *
from pyspark.sql.functions import exp
from pyspark.sql.functions import col
from pyspark.sql.functions import lit
from pyspark.sql.functions import array
from pyspark.sql.types import *
from multiprocessing.pool import ThreadPool

#get all orders
orders = sqlContext.read.jdbc(url=jdbcUrl, table="orders", properties=connectionProperties)

#get all order details
orderdetails = sqlContext.read.jdbc(url=jdbcUrl, table="orderdetails", properties=connectionProperties)

#get all OrderId values to pass to map function
orderids = orders.select('OrderId').collect()

#create thread pool big enough to process merge of details to orders in parallel
pool = ThreadPool(10)
```

Then, create a function for writing Orders into the target SQL API collection. This function will filter all order details for the given order ID, convert them into a JSON array, and insert the array into a JSON document that we will write into the target SQL API Collection for that order:

```
def writeOrder(orderid):
  #filter the order on current value passed from map function
  order = orders.filter(orders['OrderId'] == orderid[0])

  #set id to be a uuid
  order = order.withColumn("id", lit(str(uuid.uuid1())))

  #add details field to order dataframe
  order = order.withColumn("details", lit(''))

  #filter order details dataframe to get details we want to merge into the order document
  orderdetailsgroup = orderdetails.filter(orderdetails['OrderId'] == orderid[0])

  #convert dataframe to pandas
  orderpandas = order.toPandas()

  #convert the order dataframe to json and remove enclosing brackets
  orderjson = orderpandas.to_json(orient='records', force_ascii=False)
  orderjson = orderjson[1:-1]

  #convert orderjson to a dictionaory so we can set the details element with order details later
  orderjsondata = json.loads(orderjson)


  #convert orderdetailsgroup dataframe to json, but only if details were returned from the earlier filter
  if (orderdetailsgroup.count() !=0):
    #convert orderdetailsgroup to pandas dataframe to work better with json
    orderdetailsgroup = orderdetailsgroup.toPandas()

    #convert orderdetailsgroup to json string
    jsonstring = orderdetailsgroup.to_json(orient='records', force_ascii=False)

    #convert jsonstring to dictionary to ensure correct encoding and no corrupt records
    jsonstring = json.loads(jsonstring)

    #set details json element in orderjsondata to jsonstring which contains orderdetailsgroup - this merges
order details into the order
    orderjsondata['details'] = jsonstring

  #convert dictionary to json
  orderjsondata = json.dumps(orderjsondata)

  #read the json into spark dataframe
  df = spark.read.json(sc.parallelize([orderjsondata]))

  #write the dataframe (this will be a single order record with merged many-to-one order details) to cosmos db
using spark the connector
  #https://docs.microsoft.com/azure/cosmos-db/spark-connector
  df.write.format("com.microsoft.azure.cosmosdb.spark").mode("append").options(**writeConfig).save()
```

Finally, we will call the above using a map function on the thread pool, to execute in parallel, passing in the list of order IDs we created earlier:

```
#map order details to orders in parallel using the above function
pool.map(writeOrder, orderids)
```

In either approach, at the end, we should get properly saved embedded OrderDetails within each Order document in Cosmos DB collection:

Next steps

- Learn about data modeling in Azure Cosmos DB
- Learn how to model and partition data on Azure Cosmos DB

Migrate from CouchBase to Azure Cosmos DB SQL API

12/4/2020 • 8 minutes to read • Edit Online

APPLIES TO: SQL API

Azure Cosmos DB is a scalable, globally distributed, fully managed database. It provides guaranteed low latency access to your data. To learn more about Azure Cosmos DB, see the overview article. This article provides instructions to migrate Java applications that are connected to Couchbase to a SQL API account in Azure Cosmos DB.

Differences in nomenclature

The following are the key features that work differently in Azure Cosmos DB when compared to Couchbase:

COUCHBASE	AZURE COSMOS DB
Couchbase server	Account
Bucket	Database
Bucket	Container/Collection
JSON Document	Item / Document

Key differences

- Azure Cosmos DB has an "ID" field within the document whereas Couchbase has the ID as a part of bucket. The "ID" field is unique across the partition.
- Azure Cosmos DB scales by using the partitioning or sharding technique. Which means it splits the data into multiple shards/partitions. These partitions/shards are created based on the partition key property that you provide. You can select the partition key to optimize read as well write operations or read/write optimized too. To learn more, see the partitioning article.
- In Azure Cosmos DB, it is not required for the top-level hierarchy to denote the collection because the collection name already exists. This feature makes the JSON structure much simpler. The following is an example that shows differences in the data model between Couchbase and Azure Cosmos DB:

  Couchbase: Document ID = "99FF4444"

```
{
  "TravelDocument":
  {
   "Country":"India",
  "Validity" : "2022-09-01",
    "Person":
    {
      "Name": "Manish",
      "Address": "AB Road, City-z"
    },
    "Visas":
    [
      {
      "Country":"India",
      "Type":"Multi-Entry",
      "Validity":"2022-09-01"
      },
      {
      "Country":"US",
      "Type":"Single-Entry",
      "Validity":"2022-08-01"
      }
    ]
  }
}
```

Azure Cosmos DB: Refer "ID" within the document as shown below

```
{
  "id" : "99FF4444",

  "Country":"India",
   "Validity" : "2022-09-01",
    "Person":
    {
      "Name": "Manish",
      "Address": "AB Road, City-z"
    },
    "Visas":
    [
      {
      "Country":"India",
      "Type":"Multi-Entry",
      "Validity":"2022-09-01"
      },
      {
      "Country":"US",
      "Type":"Single-Entry",
      "Validity":"2022-08-01"
      }
    ]
  }
```

Java SDK support

Azure Cosmos DB has following SDKs to support different Java frameworks:

- Async SDK
- Spring Boot SDK

The following sections describe when to use each of these SDKs. Consider an example where we have three types of workloads:

Couchbase as document repository & spring data-based custom queries

If the workload that you are migrating is based on Spring Boot Based SDK, then you can use the following steps:

1. Add parent to the POM.xml file:

```
<parent>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-parent</artifactId>
   <version>2.1.5.RELEASE</version>
   <relativePath/>
</parent>
```

2. Add properties to the POM.xml file:

```
<azure.version>2.1.6</azure.version>
```

3. Add dependencies to the POM.xml file:

```
<dependency>
    <groupId>com.microsoft.azure</groupId>
    <artifactId>azure-cosmosdb-spring-boot-starter</artifactId>
    <version>2.1.6</version>
</dependency>
```

4. Add application properties under resources and specify the following. Make sure to replace the URL, key, and database name parameters:

```
   azure.cosmosdb.uri=<your-cosmosDB-URL>
   azure.cosmosdb.key=<your-cosmosDB-key>
   azure.cosmosdb.database=<your-cosmosDB-dbName>
```

5. Define the name of the collection in the model. You can also specify further annotations. For example, ID, partition key to denote them explicitly:

```
@Document(collection = "mycollection")
    public class User {
        @id
        private String id;
        private String firstName;
        @PartitionKey
        private String lastName;
    }
```

The following are the code snippets for CRUD operations:

Insert and update operations

Where *_repo* is the object of repository and *doc* is the POJO class's object. You can use `.save` to insert or upsert (if document with specified ID found). The following code snippet shows how to insert or update a doc object:

```
_repo.save(doc);
```

Delete Operation

Consider the following code snippet, where doc object will have ID and partition key mandatory to locate and delete the object:

```
_repo.delete(doc);
```

Read Operation

You can read the document with or without specifying the partition key. If you don't specify the partition key, then it is treated as a cross-partition query. Consider the following code samples, first one will perform operation using ID and partition key field. Second example uses a regular field & without specifying the partition key field.

- `_repo.findByIdAndName(objDoc.getId(),objDoc.getName());`
- `_repo.findAllByStatus(objDoc.getStatus());`

That's it, you can now use your application with Azure Cosmos DB. Complete code sample for the example described in this doc is available in the CouchbaseToCosmosDB-SpringCosmos GitHub repo.

Couchbase as a document repository & using N1QL queries

N1QL queries is the way to define queries in the Couchbase.

N1QL QUERY	AZURE COSMOSDB QUERY
SELECT META(`TravelDocument`).id AS id, `TravelDocument` .* FROM `TravelDocument` WHERE `_type` = "com.xx.xx.xx.xxx.xxx.xxxx " and country = 'India' and ANY m in Visas SATISFIES m.type == 'Multi-Entry' and m.Country IN ['India', Bhutan'] ORDER BY `Validity` DESC LIMIT 25 OFFSET 0	SELECT c.id,c FROM c JOIN m in c.country='India' WHERE c._type = " com.xx.xx.xx.xxx.xxx.xxxx" and c.country = 'India' and m.type = 'Multi-Entry' and m.Country IN ('India', 'Bhutan') ORDER BY c.Validity DESC OFFSET 0 LIMIT 25

You can notice the following changes in your N1QL queries:

- You don't need to use the META keyword or refer to the first-level document. Instead you can create your own reference to the container. In this example, we have considered it as "c" (it can be anything). This reference is used as a prefix for all the first-level fields. Fr example, c.id, c.country etc.
- Instead of "ANY" now you can do a join on subdocument and refer it with a dedicated alias such as "m". Once you have created alias for a subdocument you need to use alias. For example, m.Country.
- The sequence of OFFSET is different in Azure Cosmos DB query, first you need to specify OFFSET then LIMIT. It is recommended not to use Spring Data SDK if you are using maximum custom defined queries as it can have unnecessary overhead at the client side while passing the query to Azure Cosmos DB. Instead we have a direct Async Java SDK, which can be utilized much efficiently in this case.

Read operation

Use the Async Java SDK with the following steps:

1. Configure the following dependency onto the POM.xml file:

```
<!-- https://mvnrepository.com/artifact/com.microsoft.azure/azure-cosmosdb -->
<dependency>
    <groupId>com.microsoft.azure</groupId>
    <artifactId>azure-cosmos</artifactId>
    <version>3.0.0</version>
</dependency>
```

2. Create a connection object for Azure Cosmos DB by using the `ConnectionBuilder` method as shown in the following example. Make sure you put this declaration into the bean such that the following code should get

executed only once:

```
ConnectionPolicy cp=new ConnectionPolicy();
cp.connectionMode(ConnectionMode.DIRECT);

if(client==null)
 client= CosmosClient.builder()
  .endpoint(Host)//(Host, MasterKey, dbName, collName).Builder()
     .connectionPolicy(cp)
     .key(MasterKey)
     .consistencyLevel(ConsistencyLevel.EVENTUAL)
     .build();

container = client.getDatabase(_dbName).getContainer(_collName);
```

3. To execute the query, you need to run the following code snippet:

```
Flux<FeedResponse<CosmosItemProperties>> objFlux= container.queryItems(query, fo);
```

Now, with the help of above method you can pass multiple queries and execute without any hassle. In case you have the requirement to execute one large query, which can be split into multiple queries then try the following code snippet instead of the previous one:

```
for(SqlQuerySpec query:queries)
{
 objFlux= container.queryItems(query, fo);
 objFlux .publishOn(Schedulers.elastic())
   .subscribe(feedResponse->
    {
     if(feedResponse.results().size()>0)
     {
      _docs.addAll(feedResponse.results());
     }

    },
    Throwable::printStackTrace,latch::countDown);
 lstFlux.add(objFlux);
}

  Flux.merge(lstFlux);
  latch.await();
}
```

With the previous code, you can run queries in parallel and increase the distributed executions to optimize. Further you can run the insert and update operations too:

Insert operation

To insert the document, run the following code:

```
Mono<CosmosItemResponse> objMono= container.createItem(doc,ro);
```

Then subscribe to Mono as:

```
CountDownLatch latch=new CountDownLatch(1);
objMono .subscribeOn(Schedulers.elastic())
        .subscribe(resourceResponse->
        {
         if(resourceResponse.statusCode()!=successStatus)
          {
           throw new RuntimeException(resourceResponse.toString());
          }
         },
        Throwable::printStackTrace,latch::countDown);
latch.await();
```

Upsert operation

Upsert operation requires you to specify the document that needs to be updated. To fetch the complete document, you can use the snippet mentioned under heading read operation then modify the required field(s). The following code snippet upserts the document:

```
Mono<CosmosItemResponse> obs= container.upsertItem(doc, ro);
```

Then subscribe to mono. Refer to the mono subscription snippet in insert operation.

Delete operation

Following snippet will do delete operation:

```
CosmosItem objItem= container.getItem(doc.Id, doc.Tenant);
Mono<CosmosItemResponse> objMono = objItem.delete(ro);
```

Then subscribe to mono, refer mono subscription snippet in insert operation. The complete code sample is available in the CouchbaseToCosmosDB-AsyncInSpring GitHub repo.

Couchbase as a key/value pair

This is a simple type of workload in which you can perform lookups instead of queries. Use the following steps for key/value pairs:

1. Consider having "/ID" as primary key, which will makes sure you can perform lookup operation directly in the specific partition. Create a collection and specify "/ID" as partition key.

2. Switch off the indexing completely. Because you will execute lookup operations, there is no point of carrying indexing overhead. To turn off indexing, sign into Azure portal, goto Azure Cosmos DB Account. Open the **Data Explorer**, select your **Database** and the **Container**. Open the **Scale & Settings** tab and select the **Indexing Policy**. Currently indexing policy looks like the following:

```
{
 "indexingMode": "consistent",
 "automatic": true,
 "includedPaths": [
     {
         "path": "/*"
     }
 ],
 "excludedPaths": [
     {
         "path": "/\"_etag\"/?"
     }
 ]
 }
```

Replace the above indexing policy with the following policy:

```
{
 "indexingMode": "none",
 "automatic": false,
 "includedPaths": [],
 "excludedPaths": []
 }
```

3. Use the following code snippet to create the connection object. Connection Object (to be placed in @Bean or make it static):

```
ConnectionPolicy cp=new ConnectionPolicy();
cp.connectionMode(ConnectionMode.DIRECT);

if(client==null)
 client= CosmosClient.builder()
  .endpoint(Host)//(Host, MasterKey, dbName, collName).Builder()
     .connectionPolicy(cp)
     .key(MasterKey)
     .consistencyLevel(ConsistencyLevel.EVENTUAL)
     .build();

container = client.getDatabase(_dbName).getContainer(_collName);
```

Now you can execute the CRUD operations as follows:

Read operation

To read the item, use the following snippet:

```
CosmosItemRequestOptions ro=new CosmosItemRequestOptions();
ro.partitionKey(new PartitionKey(documentId));
CountDownLatch latch=new CountDownLatch(1);

var objCosmosItem= container.getItem(documentId, documentId);
Mono<CosmosItemResponse> objMono = objCosmosItem.read(ro);
objMono .subscribeOn(Schedulers.elastic())
        .subscribe(resourceResponse->
        {
         if(resourceResponse.item()!=null)
         {
          doc= resourceResponse.properties().toObject(UserModel.class);
         }
        },
        Throwable::printStackTrace,latch::countDown);
latch.await();
```

Insert operation

To insert an item, you can perform the following code:

```
Mono<CosmosItemResponse> objMono= container.createItem(doc,ro);
```

Then subscribe to mono as:

```
CountDownLatch latch=new CountDownLatch(1);
objMono.subscribeOn(Schedulers.elastic())
  .subscribe(resourceResponse->
  {
   if(resourceResponse.statusCode()!=successStatus)
    {
     throw new RuntimeException(resourceResponse.toString());
    }
   },
  Throwable::printStackTrace,latch::countDown);
latch.await();
```

Upsert operation

To update the value of an item, refer the code snippet below:

```
Mono<CosmosItemResponse> obs= container.upsertItem(doc, ro);
```

Then subscribe to mono, refer mono subscription snippet in insert operation.

Delete operation

Use the following snippet to execute the delete operation:

```
CosmosItem objItem= container.getItem(id, id);
Mono<CosmosItemResponse> objMono = objItem.delete(ro);
```

Then subscribe to mono, refer mono subscription snippet in insert operation. The complete code sample is available in the CouchbaseToCosmosDB-AsyncKeyValue GitHub repo.

Data Migration

There are two ways to migrate data.

- **Use Azure Data Factory:** This is the most recommended method to migrate the data. Configure the source as Couchbase and sink as Azure Cosmos DB SQL API, see the Azure Cosmos DB Data Factory connector article for detailed steps.
- **Use the Azure Cosmos DB data import tool:** This option is recommended to migrate using VMs with less amount of data. For detailed steps, see the Data importer article.

Next Steps

- To do performance testing, see Performance and scale testing with Azure Cosmos DB article.
- To optimize the code, see Performance tips for Azure Cosmos DB article.
- Explore Java Async V3 SDK, SDK reference GitHub repo.

Migrate your application from Amazon DynamoDB to Azure Cosmos DB

12/4/2020 • 9 minutes to read • Edit Online

APPLIES TO: SQL API

Azure Cosmos DB is a scalable, globally distributed, fully managed database. It provides guaranteed low latency access to your data. To learn more about Azure Cosmos DB, see the overview article. This article describes how to migrate your .NET application from DynamoDB to Azure Cosmos DB with minimal code changes.

Conceptual differences

The following are the key conceptual differences between Azure Cosmos DB and DynamoDB:

DYNAMODB	AZURE COSMOS DB
Not applicable	Database
Table	Collection
Item	Document
Attribute	Field
Secondary Index	Secondary Index
Primary Key – Partition key	Partition Key
Primary Key – Sort Key	Not Required
Stream	ChangeFeed
Write Compute Unit	Request Unit (Flexible, can be used for reads or writes)
Read Compute Unit	Request Unit (Flexible, can be used for reads or writes)
Global Tables	Not Required. You can directly select the region while provisioning the Azure Cosmos account (you can change the region later)

Structural differences

Azure Cosmos DB has a simpler JSON structure when compared to that of DynamoDB. The following example shows the differences

DynamoDB:

The following JSON object represents the data format in DynamoDB

```
{
TableName: "Music",
KeySchema: [
{
  AttributeName: "Artist",
  KeyType: "HASH", //Partition key
},
{
  AttributeName: "SongTitle",
  KeyType: "RANGE" //Sort key
}
],
AttributeDefinitions: [
{
  AttributeName: "Artist",
  AttributeType: "S"
},
{
  AttributeName: "SongTitle",
  AttributeType: "S"
}
],
ProvisionedThroughput: {
  ReadCapacityUnits: 1,
  WriteCapacityUnits: 1
 }
}
```

Azure Cosmos DB:

The following JSON object represents the data format in Azure Cosmos DB

```
{
"Artist": "",
"SongTitle": "",
"AlbumTitle": "",
"Year": 9999,
"Price": 0.0,
"Genre": "",
"Tags": ""
}
```

Migrate your data

There are various options available to migrate your data to Azure Cosmos DB. To learn more, see the Options to migrate your on-premises or cloud data to Azure Cosmos DB article.

Migrate your code

This article is scoped to migrate an application's code to Azure Cosmos DB, which is the critical aspect of database migration. To help you reduce learning curve, the following sections include a side-by-side code comparison between Amazon DynamoDB and Azure Cosmos DB's equivalent code snippet.

To download the source code, clone the following repo:

```
git clone https://github.com/Azure-Samples/DynamoDB-to-CosmosDB
```

Pre-requisites

- .NET Framework 4.7.2

- Visual Studio 2019
- Access to Azure Cosmos DB SQL API Account
- Local installation of Amazon DynamoDB
- Java 8
- Run the downloadable version of Amazon DynamoDB at port 8000 (you can change and configure the code)

Set up your code

Add the following "NuGet package" to your project:

```
Install-Package Microsoft.Azure.Cosmos
```

Establish connection

DynamoDB:

In Amazon DynamoDB, the following code is used to connect:

```
    AmazonDynamoDBConfig addbConfig = new AmazonDynamoDBConfig() ;
        addbConfig.ServiceURL = "endpoint";
        try { aws_dynamodbclient = new AmazonDynamoDBClient( addbConfig ); }
```

Azure Cosmos DB:

To connect Azure Cosmos DB, update your code to:

```
client_documentDB = new CosmosClient("your connectionstring from the Azure portal");
```

Optimize the connection in Azure Cosmos DB

With Azure Cosmos DB, you can use the following options to optimize your connection:

- **ConnectionMode** - Use direct connection mode to connect to the data nodes in the Azure Cosmos DB service. Use gateway mode only to initialize and cache the logical addresses and refresh on updates. See the connectivity modes article for more details.
- **ApplicationRegion** - This option is used to set the preferred geo-replicated region that is used to interact with Azure Cosmos DB. To learn more see the global distribution article.
- **ConsistencyLevel** - This option is used to override default consistency level. To learn more, see the Consistency levels article.
- **BulkExecutionMode** - This option is used to execute bulk operations by setting the *AllowBulkExecution* property to true. To learn more see the Bulk import article.

  ```
  client_cosmosDB = new CosmosClient(" Your connection string ",new CosmosClientOptions()
  {
    ConnectionMode=ConnectionMode.Direct,
    ApplicationRegion=Regions.EastUS2,
    ConsistencyLevel=ConsistencyLevel.Session,
    AllowBulkExecution=true
  });
  ```

Provision the container

DynamoDB:

To store the data into Amazon DynamoDB you need to create the table first. In this process you define the schema,

key type, and attributes as shown in the following code:

```
// movies_key_schema
public static List<KeySchemaElement> movies_key_schema
  = new List<KeySchemaElement>
{
  new KeySchemaElement
  {
    AttributeName = partition_key_name,
    KeyType = "HASH"
  },
  new KeySchemaElement
  {
    AttributeName = sort_key_name,
    KeyType = "RANGE"
  }
};

// key names for the Movies table
public const string partition_key_name = "year";
public const string sort_key_name      = "title";
  public const int readUnits=1, writeUnits=1;

    // movie_items_attributes
    public static List<AttributeDefinition> movie_items_attributes
  = new List<AttributeDefinition>
{
  new AttributeDefinition
  {
    AttributeName = partition_key_name,
    AttributeType = "N"
  },
  new AttributeDefinition
  {
    AttributeName = sort_key_name,
    AttributeType = "S"
  }

CreateTableRequest  request;
CreateTableResponse response;

// Build the 'CreateTableRequest' structure for the new table
request = new CreateTableRequest
{
  TableName             = table_name,
  AttributeDefinitions  = table_attributes,
  KeySchema             = table_key_schema,
  // Provisioned-throughput settings are always required,
  // although the local test version of DynamoDB ignores them.
  ProvisionedThroughput = new ProvisionedThroughput( readUnits, writeUnits );
};
```

Azure Cosmos DB:

In Amazon DynamoDB, you need to provision the read compute units & write compute units. Whereas in Azure Cosmos DB you specify the throughput as Request Units (RU/s), which can be used for any operations dynamically. The data is organized as database --> container--> item. You can specify the throughput at database level or at collection level or both.

To create a database:

```
await client_cosmosDB.CreateDatabaseIfNotExistsAsync(movies_table_name);
```

To create the container:

```
await cosmosDatabase.CreateContainerIfNotExistsAsync(new ContainerProperties() { PartitionKeyPath = "/" +
partitionKey, Id = new_collection_name }, provisionedThroughput);
```

Load the data

DynamoDB:

The following code shows how to load the data in Amazon DynamoDB. The moviesArray consists of list of JSON document then you need to iterate through and load the JSON document into Amazon DynamoDB:

```
int n = moviesArray.Count;
for( int i = 0, j = 99; i < n; i++ )
    {
  try
  {
    string itemJson = moviesArray[i].ToString();
    Document doc = Document.FromJson(itemJson);
    Task putItem = moviesTable.PutItemAsync(doc);
    if( i >= j )
    {
      j++;
      Console.Write( "{0,5:#,##0}, ", j );
      if( j % 1000 == 0 )
        Console.Write( "\n " );
      j += 99;
    }
    await putItem;
```

Azure Cosmos DB:

In Azure Cosmos DB, you can opt for stream and write with `moviesContainer.CreateItemStreamAsync()`. However, in this sample, the JSON will be deserialized into the *MovieModel* type to demonstrate type-casting feature. The code is multi-threaded, which will use Azure Cosmos DB's distributed architecture and speed-up the loading:

```
List<Task> concurrentTasks = new List<Task>();
for (int i = 0, j = 99; i < n; i++)
{
  try
  {
      MovieModel doc= JsonConvert.DeserializeObject<MovieModel>(moviesArray[i].ToString());
      doc.Id = Guid.NewGuid().ToString();
      concurrentTasks.Add(moviesContainer.CreateItemAsync(doc,new PartitionKey(doc.Year)));
      {
          j++;
          Console.Write("{0,5:#,##0}, ", j);
          if (j % 1000 == 0)
              Console.Write("\n               ");
          j += 99;
      }

  }
  catch (Exception ex)
  {
      Console.WriteLine("\n     ERROR: Could not write the movie record #{0:#,##0}, because:\n       {1}",
                          i, ex.Message);
      operationFailed = true;
      break;
  }
}
await Task.WhenAll(concurrentTasks);
```

Create a document

DynamoDB:

Writing a new document in Amazon DynamoDB isn't type safe, the following example uses newItem as document type:

```
Task<Document> writeNew = moviesTable.PutItemAsync(newItem, token);
await writeNew;
```

Azure Cosmos DB:

Azure Cosmos DB provides you type safety via data model. We use data model named 'MovieModel':

```
public class MovieModel
{
    [JsonProperty("id")]
    public string Id { get; set; }
    [JsonProperty("title")]
    public string Title{ get; set; }
    [JsonProperty("year")]
    public int Year { get; set; }
    public MovieModel(string title, int year)
    {
        this.Title = title;
        this.Year = year;
    }
    public MovieModel()
    {

    }
    [JsonProperty("info")]
    public   MovieInfo MovieInfo { get; set; }

    internal string PrintInfo()
    {
        if(this.MovieInfo!=null)
        return            string.Format("\nMovie with title:{1}\n Year: {2}, Actors: {3}\n Directors:{4}\n
Rating:{5}\n", this.Id, this.Title, this.Year, String.Join(",",this.MovieInfo.Actors), this.MovieInfo,
this.MovieInfo.Rating);
        else
            return string.Format("\nMovie with  title:{0}\n Year: {1}\n",  this.Title, this.Year);
    }
}
```

In Azure Cosmos DB newItem will be MovieModel:

```
 MovieModel movieModel = new MovieModel()
            {
                Id = Guid.NewGuid().ToString(),
                Title = "The Big New Movie",
                Year = 2018,
                MovieInfo = new MovieInfo() { Plot = "Nothing happens at all.", Rating = 0 }
            };
    var writeNew= moviesContainer.CreateItemAsync(movieModel, new
Microsoft.Azure.Cosmos.PartitionKey(movieModel.Year));
    await writeNew;
```

Read a document

DynamoDB:

To read in Amazon DynamoDB, you need to define primitives:

```
// Create Primitives for the HASH and RANGE portions of the primary key
Primitive hash = new Primitive(year.ToString(), true);
Primitive range = new Primitive(title, false);

  Task<Document> readMovie = moviesTable.GetItemAsync(hash, range, token);
  movie_record = await readMovie;
```

Azure Cosmos DB:

However, with Azure Cosmos DB the query is natural (linq):

```
IQueryable<MovieModel> movieQuery = moviesContainer.GetItemLinqQueryable<MovieModel>(true)
                        .Where(f => f.Year == year && f.Title == title);
// The query is executed synchronously here, but can also be executed asynchronously via the IDocumentQuery<T>
interface
    foreach (MovieModel movie in movieQuery)
    {
      movie_record_cosmosdb = movie;
    }
```

The documents collection in the above example will be:

- type safe
- provide a natural query option.

Update an item

DynamoDB: To update the item in Amazon DynamoDB:

```
updateResponse = await client.UpdateItemAsync( updateRequest );
```

Azure Cosmos DB:

In Azure Cosmos DB, update will be treated as Upsert operation meaning insert the document if it doesn't exist:

```
await moviesContainer.UpsertItemAsync<MovieModel>(updatedMovieModel);
```

Delete a document

DynamoDB:

To delete an item in Amazon DynamoDB, you again need to fall on primitives:

```
Primitive hash = new Primitive(year.ToString(), true);
      Primitive range = new Primitive(title, false);
      DeleteItemOperationConfig deleteConfig = new DeleteItemOperationConfig( );
      deleteConfig.ConditionalExpression = condition;
      deleteConfig.ReturnValues = ReturnValues.AllOldAttributes;

  Task<Document> delItem = table.DeleteItemAsync( hash, range, deleteConfig );
        deletedItem = await delItem;
```

Azure Cosmos DB:

In Azure Cosmos DB, we can get the document and delete them asynchronously:

```
var result= ReadingMovieItem_async_List_CosmosDB("select * from c where c.info.rating>7 AND c.year=2018 AND
c.title='The Big New Movie'");
while (result.HasMoreResults)
{
  var resultModel = await result.ReadNextAsync();
  foreach (var movie in resultModel.ToList<MovieModel>())
  {
    await moviesContainer.DeleteItemAsync<MovieModel>(movie.Id, new PartitionKey(movie.Year));
  }
  }
```

Query documents

DynamoDB:

In Amazon DynamoDB, api functions are required to query the data:

```
QueryOperationConfig config = new QueryOperationConfig( );
  config.Filter = new QueryFilter( );
  config.Filter.AddCondition( "year", QueryOperator.Equal, new DynamoDBEntry[ ] { 1992 } );
  config.Filter.AddCondition( "title", QueryOperator.Between, new DynamoDBEntry[ ] { "B", "Hzz" } );
  config.AttributesToGet = new List<string> { "year", "title", "info" };
  config.Select = SelectValues.SpecificAttributes;
  search = moviesTable.Query( config );
```

Azure Cosmos DB:

In Azure Cosmos DB, you can do projection and filter inside a simple sql query:

```
var result = moviesContainer.GetItemQueryIterator<MovieModel>(
  "select c.Year, c.Title, c.info from c where Year=1998 AND (CONTAINS(Title,'B') OR CONTAINS(Title,'Hzz'))");
```

For range operations, for example, 'between', you need to do a scan in Amazon DynamoDB:

```
ScanRequest sRequest = new ScanRequest
{
  TableName = "Movies",
  ExpressionAttributeNames = new Dictionary<string, string>
  {
    { "#yr", "year" }
  },
  ExpressionAttributeValues = new Dictionary<string, AttributeValue>
  {
      { ":y_a", new AttributeValue { N = "1960" } },
      { ":y_z", new AttributeValue { N = "1969" } },
  },
  FilterExpression = "#yr between :y_a and :y_z",
  ProjectionExpression = "#yr, title, info.actors[0], info.directors, info.running_time_secs"
};

ClientScanning_async( sRequest ).Wait( );
```

In Azure Cosmos DB, you can use SQL query and a single-line statement:

```
var result = moviesContainer.GetItemQueryIterator<MovieModel>(
  "select c.title, c.info.actors[0], c.info.directors,c.info.running_time_secs from c where BETWEEN year 1960
AND 1969");
```

Delete a container

DynamoDB:

To delete the table in Amazon DynamoDB, you can specify:

```
client.DeleteTableAsync( tableName );
```

Azure Cosmos DB:

To delete the collection in Azure Cosmos DB, you can specify:

```
await moviesContainer.DeleteContainerAsync();
```

Then delete the database too if you need:

```
await cosmosDatabase.DeleteAsync();
```

As you can see, Azure Cosmos DB supports natural queries (SQL), operations are asynchronous and much easier. You can easily migrate your complex code to Azure Cosmos DB, which becomes simpler after the migration.

Next Steps

- Learn about performance optimization.
- Learn about optimize reads and writes
- Learn about Monitoring in Cosmos DB

Configure Cross-Origin Resource Sharing (CORS)

12/4/2020 • 3 minutes to read • Edit Online

APPLIES TO: SQL API

Cross-Origin Resource Sharing (CORS) is an HTTP feature that enables a web application running under one domain to access resources in another domain. Web browsers implement a security restriction known as same-origin policy that prevents a web page from calling APIs in a different domain. However, CORS provides a secure way to allow the origin domain to call APIs in another domain. The Core (SQL) API in Azure Cosmos DB now supports Cross-Origin Resource Sharing (CORS) by using the "allowedOrigins" header. After you enable the CORS support for your Azure Cosmos account, only authenticated requests are evaluated to determine whether they are allowed according to the rules you have specified.

You can configure the Cross-origin resource sharing (CORS) setting from the Azure portal or from an Azure Resource Manager template. For Cosmos accounts using the Core (SQL) API, Azure Cosmos DB supports a JavaScript library that works in both Node.js and browser-based environments. This library can now take advantage of CORS support when using Gateway mode. There is no client-side configuration needed to use this feature. With CORS support, resources from a browser can directly access Azure Cosmos DB through the JavaScript library or directly from the REST API for simple operations.

> **NOTE**
>
> CORS support is only applicable and supported for the Azure Cosmos DB Core (SQL) API. It is not applicable to the Azure Cosmos DB APIs for Cassandra, Gremlin, or MongoDB, as these protocols do not use HTTP for client-server communication.

Enable CORS support from Azure portal

Use the following steps to enable Cross-Origin Resource Sharing by using Azure portal:

1. Navigate to your Azure cosmos DB account. Open the **CORS** blade.
2. Specify a comma-separated list of origins that can make cross-origin calls to your Azure Cosmos DB account. For example, `https://www.mydomain.com`, `https://mydomain.com`, `https://api.mydomain.com`. You can also use a wildcard "*" to allow all origins and select **Submit**.

   > **NOTE**
   >
   > Currently, you cannot use wildcards as part of the domain name. For example `https://*.mydomain.net` format is not yet supported.

Enable CORS support from Resource Manager template

To enable CORS by using a Resource Manager template, add the “cors” section with “allowedOrigins” property to any existing template. The following JSON is an example of a template that creates a new Azure Cosmos account with CORS enabled.

```
{
  "type": "Microsoft.DocumentDB/databaseAccounts",
  "name": "[variables('accountName')]",
  "apiVersion": "2019-08-01",
  "location": "[parameters('location')]",
  "kind": "GlobalDocumentDB",
  "properties": {
    "consistencyPolicy": "[variables('consistencyPolicy')[parameters('defaultConsistencyLevel')]]",
    "locations": "[variables('locations')]",
    "databaseAccountOfferType": "Standard",
    "cors": [
      {
        "allowedOrigins": "*"
      }
    ]
  }
}
```

Using the Azure Cosmos DB JavaScript library from a browser

Today, the Azure Cosmos DB JavaScript library only has the CommonJS version of the library shipped with its package. To use this library from the browser, you have to use a tool such as Rollup or Webpack to create a browser compatible library. Certain Node.js libraries should have browser mocks for them. The following is an example of a webpack config file that has the necessary mock settings.

```
const path = require("path");

module.exports = {
  entry: "./src/index.ts",
  devtool: "inline-source-map",
  node: {
    net: "mock",
    tls: "mock"
  },
  output: {
    filename: "bundle.js",
    path: path.resolve(__dirname, "dist")
  }
};
```

Here is a code sample that uses TypeScript and Webpack with the Azure Cosmos DB JavaScript SDK library to build a Todo app that sends real time updates when new items are created. As a best practice, do not use the primary key to communicate with Azure Cosmos DB from the browser. Instead, use resource tokens to communicate. For more information about resource tokens, see Securing access to Azure Cosmos DB article.

Next steps

To learn about other ways to secure your Azure Cosmos account, see the following articles:

- Configure a firewall for Azure Cosmos DB article.
- Configure virtual network and subnet-based access for your Azure Cosmos DB account

Use system-assigned managed identities to access Azure Cosmos DB data

12/4/2020 • 5 minutes to read • Edit Online

APPLIES TO: SQL API

In this article, you'll set up a *robust, key rotation agnostic* solution to access Azure Cosmos DB keys by using managed identities. The example in this article uses Azure Functions, but you can use any service that supports managed identities.

You'll learn how to create a function app that can access Azure Cosmos DB data without needing to copy any Azure Cosmos DB keys. The function app will wake up every minute and record the current temperature of an aquarium fish tank. To learn how to set up a timer-triggered function app, see the Create a function in Azure that is triggered by a timer article.

To simplify the scenario, a Time To Live setting is already configured to clean up older temperature documents.

Assign a system-assigned managed identity to a function app

In this step, you'll assign a system-assigned managed identity to your function app.

1. In the Azure portal, open the **Azure Function** pane and go to your function app.
2. Open the **Platform features** > **Identity** tab:

3. On the **Identity** tab, turn **On** the system identity **Status** and select **Save**. The **Identity** pane should look as follows:

Grant access to your Azure Cosmos account

In this step, you'll assign a role to the function app's system-assigned managed identity. Azure Cosmos DB has multiple built-in roles that you can assign to the managed identity. For this solution, you'll use the following two roles:

BUILT-IN ROLE	DESCRIPTION
DocumentDB Account Contributor	Can manage Azure Cosmos DB accounts. Allows retrieval of read/write keys.
Cosmos DB Account Reader Role	Can read Azure Cosmos DB account data. Allows retrieval of read keys.

IMPORTANT

Support for role-based access control in Azure Cosmos DB applies to control plane operations only. Data plane operations are secured through primary keys or resource tokens. To learn more, see the Secure access to data article.

TIP

When you assign roles, assign only the needed access. If your service requires only reading data, then assign the **Cosmos DB Account Reader** role to the managed identity. For more information about the importance of least privilege access, see the Lower exposure of privileged accounts article.

In this scenario, the function app will read the temperature of the aquarium, then write back that data to a container in Azure Cosmos DB. Because the function app must write the data, you'll need to assign the **DocumentDB Account Contributor** role.

Assign the role using Azure portal

1. Sign in to the Azure portal and go to your Azure Cosmos DB account. Open the **Access control (IAM)** pane and then the **Role assignments** tab:

2. Select **+ Add** > **Add role assignment**.

3. The **Add role assignment** panel opens to the right:

- **Role**: Select **DocumentDB Account Contributor**
- **Assign access to**: Under the **Select system-assigned managed identity** subsection, select **Function App**.
- **Select**: The pane will be populated with all the function apps in your subscription that have a **Managed System Identity**. In this case, select the **FishTankTemperatureService** function app:

4. After you have selected your function app, select **Save**.

Assign the role using Azure CLI

To assign the role by using Azure CLI, open the Azure Cloud Shell and run the following commands:

```
scope=$(az cosmosdb show --name '<Your_Azure_Cosmos_account_name>' --resource-group '<CosmosDB_Resource_Group>'
--query id)

principalId=$(az webapp identity show -n '<Your_Azure_Function_name>' -g '<Azure_Function_Resource_Group>' --
query principalId)

az role assignment create --assignee $principalId --role "DocumentDB Account Contributor" --scope $scope
```

Programmatically access the Azure Cosmos DB keys

Now we have a function app that has a system-assigned managed identity with the **DocumentDB Account Contributor** role in the Azure Cosmos DB permissions. The following function app code will get the Azure Cosmos DB keys, create a CosmosClient object, get the temperature of the aquarium, and then save this to Azure Cosmos DB.

This sample uses the List Keys API to access your Azure Cosmos DB account keys.

> **IMPORTANT**
>
> If you want to assign the Cosmos DB Account Reader role, you'll need to use the List Read Only Keys API. This will populate just the read-only keys.

The List Keys API returns the `DatabaseAccountListKeysResult` object. This type isn't defined in the C# libraries. The following code shows the implementation of this class:

```
namespace Monitor
{
  public class DatabaseAccountListKeysResult
  {
      public string primaryMasterKey {get;set;}
      public string primaryReadonlyMasterKey {get; set;}
      public string secondaryMasterKey {get; set;}
      public string secondaryReadonlyMasterKey {get;set;}
  }
}
```

The example also uses a simple document called "TemperatureRecord," which is defined as follows:

```
using System;

namespace Monitor
{
    public class TemperatureRecord
    {
        public string id { get; set; } = Guid.NewGuid().ToString();
        public DateTime RecordTime { get; set; }
        public int Temperature { get; set; }

    }
}
```

You'll use the Microsoft.Azure.Services.AppAuthentication library to get the system-assigned managed identity token. To learn other ways to get the token and find out more information about the `Microsoft.Azure.Service.AppAuthentication` library, see the Service-to-service authentication article.

```
using System;
using System.Net.Http;
using System.Net.Http.Headers;
using System.Threading.Tasks;
using Microsoft.Azure.Cosmos;
using Microsoft.Azure.Services.AppAuthentication;
using Microsoft.Azure.WebJobs;
using Microsoft.Extensions.Logging;

namespace Monitor
{
    public static class FishTankTemperatureService
```

```
    {
        private static string subscriptionId =
        "<azure subscription id>";
        private static string resourceGroupName =
        "<name of your azure resource group>";
        private static string accountName =
        "<Azure Cosmos DB account name>";
        private static string cosmosDbEndpoint =
        "<Azure Cosmos DB endpoint>";
        private static string databaseName =
        "<Azure Cosmos DB name>";
        private static string containerName =
        "<container to store the temperature in>";

        [FunctionName("FishTankTemperatureService")]
        public static async Task Run([TimerTrigger("0 * * * * *")]TimerInfo myTimer, ILogger log)
        {
            log.LogInformation($"Starting temperature monitoring: {DateTime.Now}");

            // AzureServiceTokenProvider will help us to get the Service Managed token.
            var azureServiceTokenProvider = new AzureServiceTokenProvider();

            // Authenticate to the Azure Resource Manager to get the Service Managed token.
            string accessToken = await
azureServiceTokenProvider.GetAccessTokenAsync("https://management.azure.com/");

            // Setup the List Keys API to get the Azure Cosmos DB keys.
            string endpoint =
$"https://management.azure.com/subscriptions/{subscriptionId}/resourceGroups/{resourceGroupName}/providers/Micr
osoft.DocumentDB/databaseAccounts/{accountName}/listKeys?api-version=2019-12-12";

            // Setup an HTTP Client and add the access token.
            HttpClient httpClient = new HttpClient();
            httpClient.DefaultRequestHeaders.Authorization = new AuthenticationHeaderValue("Bearer",
accessToken);

            // Post to the endpoint to get the keys result.
            var result = await httpClient.PostAsync(endpoint, new StringContent(""));

            // Get the result back as a DatabaseAccountListKeysResult.
            DatabaseAccountListKeysResult keys = await
result.Content.ReadAsAsync<DatabaseAccountListKeysResult>();

            log.LogInformation("Starting to create the client");

            CosmosClient client = new CosmosClient(cosmosDbEndpoint, keys.primaryMasterKey);

            log.LogInformation("Client created");

            var database = client.GetDatabase(databaseName);
            var container = database.GetContainer(containerName);

            log.LogInformation("Get the temperature.");

            var tempRecord = new TemperatureRecord() { RecordTime = DateTime.UtcNow, Temperature =
GetTemperature() };

            log.LogInformation("Store temperature");

            await container.CreateItemAsync<TemperatureRecord>(tempRecord);

            log.LogInformation($"Ending temperature monitor: {DateTime.Now}");
        }

        private static int GetTemperature()
        {
            // Fake the temperature sensor for this demo.
            Random r = new Random(DateTime.UtcNow.Second);
            return r.Next(0, 120);
```

```
        }
    }
}
```

You are now ready to deploy your function app.

Next steps

- Certificate-based authentication with Azure Cosmos DB and Azure Active Directory
- Secure Azure Cosmos DB keys using Azure Key Vault
- Security baseline for Azure Cosmos DB

Certificate-based authentication for an Azure AD identity to access keys from an Azure Cosmos DB account

12/4/2020 • 6 minutes to read • Edit Online

APPLIES TO: SQL API

Certificate-based authentication enables your client application to be authenticated by using Azure Active Directory (Azure AD) with a client certificate. You can perform certificate-based authentication on a machine where you need an identity, such as an on-premises machine or virtual machine in Azure. Your application can then read Azure Cosmos DB keys without having the keys directly in the application. This article describes how to create a sample Azure AD application, configure it for certificate-based authentication, sign into Azure using the new application identity, and then it retrieves the keys from your Azure Cosmos account. This article uses Azure PowerShell to set up the identities and provides a C# sample app that authenticates and accesses keys from your Azure Cosmos account.

Prerequisites

- Install the latest version of Azure PowerShell.
- If you don't have an Azure subscription, create a free account before you begin.

Register an app in Azure AD

In this step, you will register a sample web application in your Azure AD account. This application is later used to read the keys from your Azure Cosmos DB account. Use the following steps to register an application:

1. Sign into the Azure portal.
2. Open the Azure **Active Directory** pane, go to **App registrations** pane, and select **New registration**.

3. Fill the **Register an application** form with the following details:

- **Name** – Provide a name for your application, it can be any name such as "sampleApp".
- **Supported account types** – Choose **Accounts in this organizational directory only (Default Directory)** to allow resources in your current directory to access this application.
- **Redirect URL** – Choose application of type **Web** and provide a URL where your application is hosted, it can be any URL. For this example, you can provide a test URL such as `https://sampleApp.com` it's okay even if the app doesn't exist.

4. Select **Register** after you fill the form.

5. After the app is registered, make a note of the **Application(client) ID** and **Object ID**, you will use these details in the next steps.

Install the AzureAD module

In this step, you will install the Azure AD PowerShell module. This module is required to get the ID of the application you registered in the previous step and associate a self-signed certificate to that application.

1. Open Windows PowerShell ISE with administrator rights. If you haven't already done, install the AZ PowerShell module and connect to your subscription. If you have multiple subscriptions, you can set the context of current subscription as shown in the following commands:

```
Install-Module -Name Az -AllowClobber
Connect-AzAccount

Get-AzSubscription
$context = Get-AzSubscription -SubscriptionId <Your_Subscription_ID>
Set-AzContext $context
```

2. Install and import the AzureAD module

```
Install-Module AzureAD
Import-Module AzureAD
```

Sign into your Azure AD

Sign into your Azure AD where you have registered the application. Use the Connect-AzureAD command to sign into your account, enter your Azure account credentials in the pop-up window.

```
Connect-AzureAD
```

Create a self-signed certificate

Open another instance of Windows PowerShell ISE, and run the following commands to create a self-signed certificate and read the key associated with the certificate:

```
$cert = New-SelfSignedCertificate -CertStoreLocation "Cert:\CurrentUser\My" -Subject "CN=sampleAppCert" -
KeySpec KeyExchange
$keyValue = [System.Convert]::ToBase64String($cert.GetRawCertData())
```

Create the certificate-based credential

Next run the following commands to get the object ID of your application and create the certificate-based credential. In this example, we set the certificate to expire after a year, you can set it to any required end date.

```
$application = Get-AzureADApplication -ObjectId <Object_ID_of_Your_Application>

New-AzureADApplicationKeyCredential -ObjectId $application.ObjectId -CustomKeyIdentifier "Key1" -Type
AsymmetricX509Cert -Usage Verify -Value $keyValue -EndDate "2020-01-01"
```

The above command results in the output similar to the screenshot below:

```
PS C:\Users\sngun> New-AzureADApplicationKeyCredential -ObjectId $application.ObjectId -CustomKeyIdentifier "Key1" -Type

CustomKeyIdentifier : {75, 101, 121, 49}
EndDate             : 1/1/2020 12:00:00 AM
KeyId               : [redacted]
StartDate           : 6/12/2019 11:47:23 AM
Type                : AsymmetricX509Cert
Usage               : Verify
Value               : {77, 73, 73, 68...}
```

Configure your Azure Cosmos account to use the new identity

1. Sign into the Azure portal.

2. Navigate to your Azure Cosmos account, open the **Access control (IAM)** blade.

3. Select **Add** and **Add role assignment**. Add the sampleApp you created in the previous step with **Contributor** role as shown in the following screenshot:

4. Select **Save** after you fill out the form

Register your certificate with Azure AD

You can associate the certificate-based credential with the client application in Azure AD from the Azure portal. To associate the credential, you must upload the certificate file with the following steps:

In the Azure app registration for the client application:

1. Sign into the Azure portal.
2. Open the Azure **Active Directory** pane, go to the **App registrations** pane, and open the sample app you created in the previous step.
3. Select **Certificates & secrets** and then **Upload certificate**. Browse the certificate file you created in the previous step to upload.
4. Select **Add**. After the certificate is uploaded, the thumbprint, start date, and expiration values are displayed.

Access the keys from PowerShell

In this step, you will sign into Azure by using the application and the certificate you created and access your Azure Cosmos account's keys.

1. Initially clear the Azure account's credentials you have used to sign into your account. You can clear credentials by using the following command:

```
Disconnect-AzAccount -Username <Your_Azure_account_email_id>
```

2. Next validate that you can sign into Azure portal by using the application's credentials and access the Azure Cosmos DB keys:

```
Login-AzAccount -ApplicationId <Your_Application_ID> -CertificateThumbprint $cert.Thumbprint -
ServicePrincipal -Tenant <Tenant_ID_of_your_application>

Get-AzCosmosDBAccountKey `
   -ResourceGroupName "<Resource_Group_Name_of_your_Azure_Cosmos_account>" `
   -Name "<Your_Azure_Cosmos_Account_Name>" `
   -Type "Keys"
```

The previous command will display the primary and secondary primary keys of your Azure Cosmos account. You can view the Activity log of your Azure Cosmos account to validate that the get keys request succeeded and the event is initiated by the "sampleApp" application.

Access the keys from a C# application

You can also validate this scenario by accessing keys from a C# application. The following C# console application, that can access Azure Cosmos DB keys by using the app registered in Active Directory. Make sure to update the tenantId, clientID, certName, resource group name, subscription ID, Azure Cosmos account name details before you run the code.

```
using System;
using Microsoft.IdentityModel.Clients.ActiveDirectory;
using System.Linq;
using System.Net.Http;
using System.Security.Cryptography.X509Certificates;
using System.Threading;
using System.Threading.Tasks;

namespace TodoListDaemonWithCert
{
    class Program
    {
        private static string aadInstance = "https://login.windows.net/";
        private static string tenantId = "<Your_Tenant_ID>";
        private static string clientId = "<Your_Client_ID>";
        private static string certName = "<Your_Certificate_Name>";

        private static int errorCode = 0;
        static int Main(string[] args)
        {
            MainAync().Wait();
            Console.ReadKey();

            return 0;
        }

        static async Task MainAync()
        {
            string authContextURL = aadInstance + tenantId;
            AuthenticationContext authContext = new AuthenticationContext(authContextURL);
            X509Certificate2 cert = ReadCertificateFromStore(certName);

            ClientAssertionCertificate credential = new ClientAssertionCertificate(clientId, cert);
            AuthenticationResult result = await authContext.AcquireTokenAsync("https://management.azure.com/",
credential);
            if (result == null)
            {
                throw new InvalidOperationException("Failed to obtain the JWT token");
            }

            string token = result.AccessToken;
            string subscriptionId = "<Your_Subscription_ID>";
            string rgName = "<ResourceGroup_of_your_Cosmos_account>";
            string accountName = "<Your_Cosmos_account_name>";
            string cosmosDBRestCall =
$"https://management.azure.com/subscriptions/{subscriptionId}/resourceGroups/{rgName}/providers/Microsoft.Docu
mentDB/databaseAccounts/{accountName}/listKeys?api-version=2015-04-08";
```

```
            Uri restCall = new Uri(cosmosDBRestCall);
            HttpClient httpClient = new HttpClient();
            httpClient.DefaultRequestHeaders.Remove("Authorization");
            httpClient.DefaultRequestHeaders.Add("Authorization", "Bearer " + token);
            HttpResponseMessage response = await httpClient.PostAsync(restCall, null);

            Console.WriteLine("Got result {0} and keys {1}", response.StatusCode.ToString(),
response.Content.ReadAsStringAsync().Result);
        }

        /// <summary>
        /// Reads the certificate
        /// </summary>
        private static X509Certificate2 ReadCertificateFromStore(string certName)
        {
            X509Certificate2 cert = null;
            X509Store store = new X509Store(StoreName.My, StoreLocation.CurrentUser);
            store.Open(OpenFlags.ReadOnly);
            X509Certificate2Collection certCollection = store.Certificates;

            // Find unexpired certificates.
            X509Certificate2Collection currentCerts = certCollection.Find(X509FindType.FindByTimeValid,
DateTime.Now, false);

            // From the collection of unexpired certificates, find the ones with the correct name.
            X509Certificate2Collection signingCert = currentCerts.Find(X509FindType.FindBySubjectName,
certName, false);

            // Return the first certificate in the collection, has the right name and is current.
            cert = signingCert.OfType<X509Certificate2>().OrderByDescending(c =>
c.NotBefore).FirstOrDefault();
            store.Close();
            return cert;
        }
    }
}
```

This script outputs the primary and secondary primary keys as shown in the following screenshot:

Similar to the previous section, you can view the Activity log of your Azure Cosmos account to validate that the get keys request event is initiated by the "sampleApp" application.

Next steps

- Secure Azure Cosmos keys using Azure Key Vault
- Security baseline for Azure Cosmos DB

Use Azure Cosmos DB change feed to visualize real-time data analytics

12/4/2020 • 16 minutes to read • Edit Online

APPLIES TO: SQL API

The Azure Cosmos DB change feed is a mechanism to get a continuous and incremental feed of records from an Azure Cosmos container as those records are being created or modified. Change feed support works by listening to container for any changes. It then outputs the sorted list of documents that were changed in the order in which they were modified. To learn more about change feed, see working with change feed article.

This article describes how change feed can be used by an e-commerce company to understand user patterns, perform real-time data analysis and visualization. You will analyze events such as a user viewing an item, adding an item to their cart, or purchasing an item. When one of these events occurs, a new record is created, and the change feed logs that record. Change feed then triggers a series of steps resulting in visualization of metrics that analyze the company performance and activity. Sample metrics that you can visualize include revenue, unique site visitors, most popular items, and average price of the items that are viewed versus added to a cart versus purchased. These sample metrics can help an e-commerce company evaluate its site popularity, develop its advertising and pricing strategies, and make decisions regarding what inventory to invest in.

Interested in watching a video about the solution before getting started, see the following video:

Solution components

The following diagram represents the data flow and components involved in the solution:

1. **Data Generation:** Data simulator is used to generate retail data that represents events such as a user viewing an item, adding an item to their cart, and purchasing an item. You can generate large set of sample data by using the data generator. The generated sample data contains documents in the following format:

   ```
   {
     "CartID": 2486,
     "Action": "Viewed",
     "Item": "Women's Denim Jacket",
     "Price": 31.99
   }
   ```

2. **Cosmos DB:** The generated data is stored in an Azure Cosmos container.

3. **Change Feed:** The change feed will listen for changes to the Azure Cosmos container. Each time a new document is added into the collection (that is when an event occurs such a user viewing an item, adding an item to their cart, or purchasing an item), the change feed will trigger an Azure Function.

4. **Azure Function:** The Azure Function processes the new data and sends it to an Azure Event Hub.

5. **Event Hub:** The Azure Event Hub stores these events and sends them to Azure Stream Analytics to perform further analysis.

6. **Azure Stream Analytics:** Azure Stream Analytics defines queries to process the events and perform real-time data analysis. This data is then sent to Microsoft Power BI.

7. **Power BI:** Power BI is used to visualize the data sent by Azure Stream Analytics. You can build a dashboard to see how the metrics change in real time.

Prerequisites

- Microsoft .NET Framework 4.7.1 or higher
- Microsoft .NET Core 2.1 (or higher)
- Visual Studio with Universal Windows Platform development, .NET desktop development, and ASP.NET and web development workloads
- Microsoft Azure Subscription
- Microsoft Power BI Account
- Download the Azure Cosmos DB change feed lab from GitHub.

Create Azure resources

Create the Azure resources - Azure Cosmos DB, Storage account, Event Hub, Stream Analytics required by the solution. You will deploy these resources through an Azure Resource Manager template. Use the following steps to deploy these resources:

1. Set the Windows PowerShell execution policy to **Unrestricted**. To do so, open **Windows PowerShell as an Administrator** and run the following commands:

   ```
   Get-ExecutionPolicy
   Set-ExecutionPolicy Unrestricted
   ```

2. From the GitHub repository you downloaded in the previous step, navigate to the **Azure Resource Manager** folder, and open the file called **parameters.json** file.

3. Provide values for cosmosdbaccount_name, eventhubnamespace_name, storageaccount_name, parameters as indicated in **parameters.json** file. You'll need to use the names that you give to each of your resources later.

4. From **Windows PowerShell**, navigate to the **Azure Resource Manager** folder and run the following command:

   ```
   .\deploy.ps1
   ```

5. When prompted, enter your Azure **Subscription ID**, **changefeedlab** for the resource group name, and **run1** for the deployment name. Once the resources begin to deploy, it may take up to 10 minutes for it to complete.

Create a database and the collection

You will now create a collection to hold e-commerce site events. When a user views an item, adds an item to their cart, or purchases an item, the collection will receive a record that includes the action ("viewed", "added", or

"purchased"), the name of the item involved, the price of the item involved, and the ID number of the user cart involved.

1. Go to Azure portal and find the **Azure Cosmos DB Account** that's created by the template deployment.

2. From the **Data Explorer** pane, select **New Collection** and fill the form with the following details:

   - For the **Database id** field, select **Create new**, then enter **changefeedlabdatabase**. Leave the **Provision database throughput** box unchecked.
   - For the **Collection** id field, enter **changefeedlabcollection**.
   - For the **Partition key** field, enter **/Item**. This is case-sensitive, so make sure you enter it correctly.
   - For the **Throughput** field, enter **10000**.
   - Select the **OK** button.

3. Next create another collection named **leases** for change feed processing. The leases collection coordinates processing the change feed across multiple workers. A separate collection is used to store the leases with one lease per partition.

4. Return to the **Data Explorer** pane and select **New Collection** and fill the form with the following details:

   - For the **Database id** field, select **Use existing**, then enter **changefeedlabdatabase**.
   - For the **Collection id** field, enter **leases**.
   - For **Storage capacity**, select **Fixed**.
   - Leave the **Throughput** field set to its default value.
   - Select the **OK** button.

Get the connection string and keys

Get the Azure Cosmos DB connection string

1. Go to Azure portal and find the **Azure Cosmos DB Account** that's created by the template deployment.

2. Navigate to the **Keys** pane, copy the PRIMARY CONNECTION STRING and copy it to a notepad or another document that you will have access to throughout the lab. You should label it **Cosmos DB Connection String**. You'll need to copy the string into your code later, so take a note and remember where you are storing it.

Get the storage account key and connection string

Azure Storage Accounts allow users to store data. In this lab, you will use a storage account to store data that is used by the Azure Function. The Azure Function is triggered when any modification is made to the collection.

1. Return to your resource group and open the storage account that you created earlier

2. Select **Access keys** from the menu on the left-hand side.

3. Copy the values under **key 1** to a notepad or another document that you will have access to throughout the lab. You should label the **Key** as **Storage Key** and the **Connection string** as **Storage Connection String**. You'll need to copy these strings into your code later, so take a note and remember where you are storing them.

Get the event hub namespace connection string

An Azure Event Hub receives the event data, stores, processes, and forwards the data. In this lab, the Azure Event Hub will receive a document every time a new event occurs (i.e. an item is viewed by a user, added to a user's cart, or purchased by a user) and then will forward that document to Azure Stream Analytics.

1. Return to your resource group and open the **Event Hub Namespace** that you created and named in the prelab.

2. Select **Shared access policies** from the menu on the left-hand side.

3. Select **RootManageSharedAccessKey**. Copy the **Connection string-primary key** to a notepad or another document that you will have access to throughout the lab. You should label it **Event Hub Namespace** connection string. You'll need to copy the string into your code later, so take a note and remember where you are storing it.

Set up Azure Function to read the change feed

When a new document is created, or a current document is modified in a Cosmos container, the change feed automatically adds that modified document to its history of collection changes. You will now build and run an Azure Function that processes the change feed. When a document is created or modified in the collection you created, the Azure Function will be triggered by the change feed. Then the Azure Function will send the modified document to the Event Hub.

1. Return to the repository that you cloned on your device.

2. Right-click the file named **ChangeFeedLabSolution.sln** and select **Open With Visual Studio**.

3. Navigate to **local.settings.json** in Visual Studio. Then use the values you recorded earlier to fill in the blanks.

4. Navigate to **ChangeFeedProcessor.cs**. In the parameters for the **Run** function, perform the following actions:

   - Replace the text **YOUR COLLECTION NAME HERE** with the name of your collection. If you followed earlier instructions, the name of your collection is changefeedlabcollection.
   - Replace the text **YOUR LEASES COLLECTION NAME HERE** with the name of your leases collection. If you followed earlier instructions, the name of your leases collection is **leases**.
   - At the top of Visual Studio, make sure that the Startup Project box on the left of the green arrow says **ChangeFeedFunction**.
   - Select **Start** at the top of the page to run the program
   - You can confirm that the function is running when the console app says "Job host started".

Insert data into Azure Cosmos DB

To see how change feed processes new actions on an e-commerce site, have to simulate data that represents users viewing items from the product catalog, adding those items to their carts, and purchasing the items in their carts. This data is arbitrary and for the purpose of replicating what data on an Ecommerce site would look like.

1. Navigate back to the repository in File Explorer, and right-click **ChangeFeedFunction.sln** to open it again in a new Visual Studio window.

2. Navigate to the **App.config** file.Within the `<appSettings>` block, add the endpoint and unique **PRIMARY KEY** that of your Azure Cosmos DB account that you retrieved earlier.

3. Add in the **collection** and **database** names. (These names should be **changefeedlabcollection** and **changefeedlabdatabase** unless you choose to name yours differently.)

```
packages.config   App.config*   Program.cs   Microsoft.NET.Publish.targets
1  <?xml version="1.0" encoding="utf-8"?>
2  <configuration>
3      <startup>
4          <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.6.1" />
5      </startup>
6      <appSettings>
7        <add key="endpoint" value="https://xxxxxxxxxxxx/" />
8        <add key="authKey" value="xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx==" />
9        <add key="database" value="xxxxxxxxxxx" />
10       <add key="collection" value="xxxxxxxx" />
11     </appSettings>
12     <runtime>
13       <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
14         <dependentAssembly>
15           <assemblyIdentity name="Newtonsoft.Json" publicKeyToken="xxxxxxxx" culture="neutral" />
16           <bindingRedirect oldVersion="0.0.0.0-11.0.0.0" newVersion="11.0.0.0" />
17         </dependentAssembly>
18       </assemblyBinding>
19     </runtime>
20 </configuration>
```

4. Save the changes on all the files edited.

5. At the top of Visual Studio, make sure that the **Startup Project** box on the left of the green arrow says **DataGenerator**. Then select **Start** at the top of the page to run the program.

6. Wait for the program to run. The stars mean that data is coming in! Keep the program running - it is important that lots of data is collected.

7. If you navigate to Azure portal , then to the Cosmos DB account within your resource group, then to **Data Explorer**, you will see the randomized data imported in your **changefeedlabcollection** .

Set up a stream analytics job

Azure Stream Analytics is a fully managed cloud service for real-time processing of streaming data. In this lab, you will use stream analytics to process new events from the Event Hub (i.e. when an item is viewed, added to a cart, or purchased), incorporate those events into real-time data analysis, and send them into Power BI for visualization.

1. From the Azure portal, navigate to your resource group, then to **streamjob1** (the stream analytics job that you created in the prelab).

2. Select **Inputs** as demonstrated below.

3. Select **+ Add stream input**. Then select **Event Hub** from the drop-down menu.

4. Fill the new input form with the following details:

   - In the **Input** alias field, enter **input**.
   - Select the option for **Select Event Hub from your subscriptions**.
   - Set the **Subscription** field to your subscription.
   - In the **Event Hub namespace** field, enter the name of your Event Hub Namespace that you created during the prelab.
   - In the **Event Hub name** field, select the option for **Use existing** and choose **event-hub1** from the drop-down menu.
   - Leave **Event Hub policy** name field set to its default value.
   - Leave **Event serialization format** as **JSON**.
   - Leave **Encoding field** set to **UTF-8**.
   - Leave **Event compression type** field set to **None**.
   - Select the **Save** button.

5. Navigate back to the stream analytics job page, and select **Outputs**.

6. Select **+ Add**. Then select **Power BI** from the drop-down menu.

7. To create a new Power BI output to visualize average price, perform the following actions:

   - In the **Output alias** field, enter **averagePriceOutput**.
   - Leave the **Group workspace** field set to **Authorize connection to load workspaces**.
   - In the **Dataset name** field, enter **averagePrice**.
   - In the **Table name** field, enter **averagePrice**.
   - Select the **Authorize** button, then follow the instructions to authorize the connection to Power BI.
   - Select the **Save** button.

8. Then go back to **streamjob1** and select **Edit query**.

9. Paste the following query into the query window. The **AVERAGE PRICE** query calculates the average price of all items that are viewed by users, the average price of all items that are added to users' carts, and the average price of all items that are purchased by users. This metric can help e-commerce companies decide what prices to sell items at and what inventory to invest in. For example, if the average price of items viewed is much higher than the average price of items purchased, then a company might choose to add less expensive items to its inventory.

```
/*AVERAGE PRICE*/
SELECT System.TimeStamp AS Time, Action, AVG(Price)
 INTO averagePriceOutput
 FROM input
 GROUP BY Action, TumblingWindow(second,5)
```

10. Then select **Save** in the upper left-hand corner.

11. Now return to **streamjob1** and select the **Start** button at the top of the page. Azure Stream Analytics can take a few minutes to start up, but eventually you will see it change from "Starting" to "Running".

Connect to Power BI

Power BI is a suite of business analytics tools to analyze data and share insights. It's a great example of how you can strategically visualize the analyzed data.

1. Sign in to Power BI and navigate to **My Workspace** by opening the menu on the left-hand side of the page.

2. Select **+ Create** in the top right-hand corner and then select **Dashboard** to create a dashboard.

3. Select **+ Add tile** in the top right-hand corner.

4. Select **Custom Streaming Data**, then select the **Next** button.

5. Select **averagePrice** from **YOUR DATASETS**, then select **Next**.

6. In the **Visualization Type** field, choose **Clustered bar chart** from the drop-down menu. Under **Axis**, add action. Skip **Legend** without adding anything. Then, under the next section called **Value**, add **avg**. Select **Next**, then title your chart, and select **Apply**. You should see a new chart on your dashboard!

7. Now, if you want to visualize more metrics, you can go back to **streamjob1** and create three more outputs with the following fields.

   a. **Output alias:** incomingRevenueOutput, Dataset name: incomingRevenue, Table name: incomingRevenue
   b. **Output alias:** top5Output, Dataset name: top5, Table name: top5
   c. **Output alias:** uniqueVisitorCountOutput, Dataset name: uniqueVisitorCount, Table name: uniqueVisitorCount

   Then select **Edit query** and paste the following queries **above** the one you already wrote.

```
/*TOP 5*/
WITH Counter AS
(
SELECT Item, Price, Action, COUNT(*) AS countEvents
FROM input
WHERE Action = 'Purchased'
GROUP BY Item, Price, Action, TumblingWindow(second,30)
),
top5 AS
(
SELECT DISTINCT
CollectTop(5)  OVER (ORDER BY countEvents) AS topEvent
FROM Counter
GROUP BY TumblingWindow(second,30)
),
arrayselect AS
(
SELECT arrayElement.ArrayValue
FROM top5
CROSS APPLY GetArrayElements(top5.topevent) AS arrayElement
)
SELECT arrayvalue.value.item, arrayvalue.value.price,   arrayvalue.value.countEvents
INTO top5Output
FROM arrayselect

/*REVENUE*/
SELECT System.TimeStamp AS Time, SUM(Price)
INTO incomingRevenueOutput
FROM input
WHERE Action = 'Purchased'
GROUP BY TumblingWindow(hour, 1)

/*UNIQUE VISITORS*/
SELECT System.TimeStamp AS Time, COUNT(DISTINCT CartID) as uniqueVisitors
INTO uniqueVisitorCountOutput
FROM input
GROUP BY TumblingWindow(second, 5)
```

The TOP 5 query calculates the top 5 items, ranked by the number of times that they have been purchased. This metric can help e-commerce companies evaluate which items are most popular and can influence the company's advertising, pricing, and inventory decisions.

The REVENUE query calculates revenue by summing up the prices of all items purchased each minute. This metric can help e-commerce companies evaluate its financial performance and also understand what times of day contribute to most revenue. This can impact the overall company strategy, marketing in particular.

The UNIQUE VISITORS query calculates how many unique visitors are on the site every 5 seconds by detecting unique cart ID's. This metric can help e-commerce companies evaluate their site activity and strategize how to acquire more customers.

8. You can now add tiles for these datasets as well.

   - For Top 5, it would make sense to do a clustered column chart with the items as the axis and the count as the value.
   - For Revenue, it would make sense to do a line chart with time as the axis and the sum of the prices as the value. The time window to display should be the largest possible in order to deliver as much information as possible.
   - For Unique Visitors, it would make sense to do a card visualization with the number of unique visitors as the value.

   This is how a sample dashboard looks with these charts:

Optional: Visualize with an E-commerce site

You will now observe how you can use your new data analysis tool to connect with a real e-commerce site. In order to build the e-commerce site, use an Azure Cosmos database to store the list of product categories (Women's, Men's, Unisex), the product catalog, and a list of the most popular items.

1. Navigate back to the Azure portal, then to your **Cosmos DB account**, then to **Data Explorer**.

   Add two collections under **changefeedlabdatabase** - **products** and **categories** with Fixed storage capacity.

   Add another collection under **changefeedlabdatabase** named **topItems** and **/Item** as the partition key.

2. Select the **topItems** collection, and under **Scale and Settings** set the **Time to Live** to be **30 seconds** so that topItems updates every 30 seconds.

3. In order to populate the **topItems** collection with the most frequently purchased items, navigate back to **streamjob1** and add a new **Output**. Select **Cosmos DB**.

4. Fill in the required fields as pictured below.

5. If you added the optional TOP 5 query in the previous part of the lab, proceed to part 5a. If not, proceed to part 5b.

   5a. In **streamjob1**, select **Edit query** and paste the following query in your Azure Stream Analytics query editor below the TOP 5 query but above the rest of the queries.

```
SELECT arrayvalue.value.item AS Item, arrayvalue.value.price, arrayvalue.value.countEvents
INTO topItems
FROM arrayselect
```

   5b. In **streamjob1**, select **Edit query** and paste the following query in your Azure Stream Analytics query editor above all other queries.

```
/*TOP 5*/
WITH Counter AS
(
SELECT Item, Price, Action, COUNT(*) AS countEvents
FROM input
WHERE Action = 'Purchased'
GROUP BY Item, Price, Action, TumblingWindow(second,30)
),
top5 AS
(
SELECT DISTINCT
CollectTop(5)  OVER (ORDER BY countEvents) AS topEvent
FROM Counter
GROUP BY TumblingWindow(second,30)
),
arrayselect AS
(
SELECT arrayElement.ArrayValue
FROM top5
CROSS APPLY GetArrayElements(top5.topevent) AS arrayElement
)
SELECT arrayvalue.value.item AS Item, arrayvalue.value.price, arrayvalue.value.countEvents
INTO topItems
FROM arrayselect
```

6. Open **EcommerceWebApp.sln** and navigate to the **Web.config** file in the **Solution Explorer**.
7. Within the `<appSettings>` block, add the **URI** and **PRIMARY KEY** that you saved earlier where it says **your URI here** and **your primary key here**. Then add in your **database name** and **collection name** as indicated. (These names should be **changefeedlabdatabase** and **changefeedlabcollection** unless you chose to name yours differently.)

   Fill in your **products collection name**, **categories collection name**, and **top items collection name** as indicated. (These names should be **products, categories, and topItems** unless you chose to name yours differently.)
8. Navigate to and open the **Checkout folder** within **EcommerceWebApp.sln.** Then open the **Web.config** file within that folder.
9. Within the `<appSettings>` block, add the **URI** and **PRIMARY KEY** that you saved earlier where indicated. Then add in your **database name** and **collection name** as indicated. (These names should be **changefeedlabdatabase** and **changefeedlabcollection** unless you chose to name yours differently.)
10. Press **Start** at the top of the page to run the program.
11. Now you can play around on the e-commerce site. When you view an item, add an item to your cart, change the quantity of an item in your cart, or purchase an item, these events will be passed through the Cosmos DB change feed to Event Hub, ASA, and then Power BI. We recommend continuing to run DataGenerator to generate significant web traffic data and provide a realistic set of "Hot Products" on the e-commerce site.

Delete the resources

To delete the resources that you created during this lab, navigate to the resource group on Azure portal, then select **Delete resource group** from the menu at the top of the page and follow the instructions provided.

Next steps

- To learn more about change feed, see working with change feed support in Azure Cosmos DB

Deploy Azure Cosmos DB and Azure App Service with a web app from GitHub using an Azure Resource Manager Template

12/4/2020 • 5 minutes to read • Edit Online

APPLIES TO: SQL API

This tutorial shows you how to do a "no touch" deployment of a web application that connects to Azure Cosmos DB on first run without having to cut and paste any connection information from Azure Cosmos DB to `appsettings.json` or to the Azure App Services application settings in the Azure portal. All these actions are accomplished using an Azure Resource Manager template in a single operation. In the example here we will deploy the Azure Cosmos DB ToDo sample from a Web app tutorial.

Resource Manager templates, are quite flexible and allow you to compose complex deployments across any service in Azure. This includes advanced tasks such as deploying applications from GitHub and injecting connection information into Azure App Service's application settings in the Azure portal. This tutorial will show you how to do the following things using a single Resource Manager template.

- Deploy an Azure Cosmos account.
- Deploy an Azure App Service Hosting Plan.
- Deploy an Azure App Service.
- Inject the endpoint and keys from the Azure Cosmos account into the App Service application settings in the Azure portal.
- Deploy a web application from a GitHub repository to the App Service.

The resulting deployment has a fully functional web application that can connect to Azure Cosmos DB without having to cut and paste the Azure Cosmos DB's endpoint URL or authentication keys from the Azure portal.

Prerequisites

> **TIP**
> While this tutorial does not assume prior experience with Azure Resource Manager templates or JSON, should you wish to modify the referenced templates or deployment options, then knowledge of each of these areas is required.

Step 1: Deploy the template

First, select the **Deploy to Azure** button below to open the Azure portal to create a custom deployment. You can also view the Azure Resource Management template from the Azure Quickstart Templates Gallery

Deploy to Azure

Once in the Azure portal, select the subscription to deploy into and select or create a new resource group. Then fill in the following values.

- **Region** - This is required by the Resource Manager. Enter the same region used by the location parameter where your resources are located.
- **Application Name** - This name is used by all the resources for this deployment. Make sure to choose a unique name to avoid conflicts with existing Azure Cosmos DB and App Service accounts.
- **Location** - The region where your resources are deployed.
- **App Service Plan Tier** - App Service Plan's pricing tier.
- **App Service Plan Instances** - The number of workers for the app service plan.
- **Repository URL** - The repository to the web application on GitHub.
- **Branch** - The branch for the GitHub repository.
- **Database Name** - The Azure Cosmos database name.
- **Container Name** - The Azure Cosmos container name.

After filling in the values, select the **Create** button to start the deployment. This step should take between 5 and 10 minutes to complete.

> **TIP**
> The template does not validate that the Azure App Service name and Azure Cosmos account name entered in the template are valid and available. It is highly recommended that you verify the availability of the names you plan to supply prior to submitting the deployment.

Step 2: Explore the resources

View the deployed resources

After the template has deployed the resources, you can now see each of them in your resource group.

View Cosmos DB endpoint and keys

Next, open the Azure Cosmos account in the portal. The following screenshot shows the endpoint and keys for an Azure Cosmos account.

View the Azure Cosmos DB keys in application settings

Next, navigate to the Azure App Service in the resource group. Click the Configuration tab to view the Application Settings for the App Service. The Application Settings contains the Cosmos DB account and primary key values necessary to connect to Cosmos DB as well as the database and container names that were passed in from the template deployment.

View web app in Deployment Center

Next go to the Deployment Center for the App Service. Here you will see Repository points to the GitHub repository passed in to the template. Also the Status below indicates Success(Active), meaning the application successfully deployed and started.

Run the web application

Click **Browse** at the top of Deployment Center to open the web application. The web application will open up to the home screen. Click on **Create New** and enter some data into the fields and click Save. The resulting screen shows the data saved to Cosmos DB.

Step 3: How does it work

There are three elements necessary for this to work.

Reading app settings at runtime

First, the application needs to request the Cosmos DB endpoint and key in the `Startup` class in the ASP.NET MVC web application. The Cosmos DB To Do Sample can run locally where you can enter the connection information into appsettings.json. However, when deployed, this file does deploy with the app. If these lines in red cannot access the settings from appsettings.json, it will try from Application Settings in Azure App Service.

```
private static async Task<CosmosDbService> InitializeCosmosClientInstanceAsync(IConfigurationSection configurationSection)
{
    string databaseName = configurationSection.GetSection("DatabaseName").Value;
    string containerName = configurationSection.GetSection("ContainerName").Value;
    string account = configurationSection.GetSection("Account").Value;
    string key = configurationSection.GetSection("Key").Value;
    Microsoft.Azure.Cosmos.CosmosClient client = new Microsoft.Azure.Cosmos.CosmosClient(account, key);
    CosmosDbService cosmosDbService = new CosmosDbService(client, databaseName, containerName);
    Microsoft.Azure.Cosmos.DatabaseResponse database = await client.CreateDatabaseIfNotExistsAsync(databaseName);
    await database.Database.CreateContainerIfNotExistsAsync(containerName, "/id");

    return cosmosDbService;
}
```

Using special Azure Resource Management functions

For these values to be available to the application when deployed, the Azure Resource Manager template can ask for those values from the Cosmos DB account using special Azure Resource Management functions including reference and listKeys which grab the values from the Cosmos DB account and insert them into the application settings values with key names that match what is used in the application above in a '{section:key}' format. For example, `CosmosDb:Account`.

```
"appSettings": [
  {
    "name": "CosmosDb:Account",
    "value": "[reference(resourceId('Microsoft.DocumentDb/databaseAccounts/', variables('cosmosAccountName'))).documentEndpoint]"
  },
  {
    "name": "CosmosDb:Key",
    "value": "[listKeys(resourceId('Microsoft.DocumentDb/databaseAccounts', variables('cosmosAccountName')), '2020-04-01').primaryMasterKey]"
  },
  {
    "name": "CosmosDb:DatabaseName",
    "value": "[parameters('DatabaseName')]"
  },
  {
    "name": "CosmosDb:ContainerName",
    "value": "[parameters('ContainerName')]"
  }
]
```

Deploying web apps from GitHub

Lastly, we need to deploy the web application from GitHub into the App Service. This is done using the JSON below.

Two things to be careful with are the type and name for this resource. Both the `"type": "sourcecontrols"` and `"name": "web"` property values are hard coded and should not be changed.

```
"resources": [
  {
    "type": "sourcecontrols",
    "apiVersion": "2019-08-01",
    "name": "web",
    "location": "[parameters('location')]",
    "dependsOn": [
      "[resourceId('Microsoft.Web/sites', variables('webSiteName'))]"
    ],
    "properties": {
      "repoUrl": "[parameters('repositoryURL')]",
      "branch": "[parameters('branch')]",
      "isManualIntegration": true
    }
  }
]
```

Next steps

Congratulations! You've deployed Azure Cosmos DB, Azure App Service, and a sample web application that automatically has the connection info necessary to connect to Cosmos DB, all in a single operation and without having to cut and paste sensitive information. Using this template as a starting point, you can modify it to deploy your own web applications the same way.

- For the Azure Resource Manager Template for this sample go to Azure Quickstart Templates Gallery
- For the source code for the sample app go to Cosmos DB To Do App on GitHub.

How to index Cosmos DB data using an indexer in Azure Cognitive Search

12/4/2020 • 15 minutes to read • Edit Online

IMPORTANT

SQL API is generally available. MongoDB API, Gremlin API, and Cassandra API support are currently in public preview. Preview functionality is provided without a service level agreement, and is not recommended for production workloads. For more information, see Supplemental Terms of Use for Microsoft Azure Previews. You can request access to the previews by filling out this form. REST API preview versions provide these features. There is currently limited portal support, and no .NET SDK support.

WARNING

Only Cosmos DB collections with an indexing policy set to Consistent are supported by Azure Cognitive Search. Indexing collections with a Lazy indexing policy is not recommended and may result in missing data. Collections with indexing disabled are not supported.

This article shows you how to configure an Azure Cosmos DB indexer to extract content and make it searchable in Azure Cognitive Search. This workflow creates an Azure Cognitive Search index and loads it with existing text extracted from Azure Cosmos DB.

Because terminology can be confusing, it's worth noting that Azure Cosmos DB indexing and Azure Cognitive Search indexing are distinct operations, unique to each service. Before you start Azure Cognitive Search indexing, your Azure Cosmos DB database must already exist and contain data.

The Cosmos DB indexer in Azure Cognitive Search can crawl Azure Cosmos DB items accessed through different protocols.

- For SQL API, which is generally available, you can use the portal, REST API, or .NET SDK to create the data source and indexer.
- For MongoDB API (preview), you can use either the portal or the REST API version 2020-06-30-Preview to create the data source and indexer.
- For Cassandra API (preview) and Gremlin API (preview), you can only use the REST API version 2020-06-30-Preview to create the data source and indexer.

NOTE

You can cast a vote on User Voice for the Table API if you'd like to see it supported in Azure Cognitive Search.

Use the portal

NOTE

The portal currently supports the SQL API and MongoDB API (preview).

The easiest method for indexing Azure Cosmos DB items is to use a wizard in the Azure portal. By sampling data and reading metadata on the container, the **Import data** wizard in Azure Cognitive Search can create a default index, map source fields to target index fields, and load the index in a single operation. Depending on the size and complexity of source data, you could have an operational full text search index in minutes.

We recommend using the same region or location for both Azure Cognitive Search and Azure Cosmos DB for lower latency and to avoid bandwidth charges.

1 - Prepare source data

You should have a Cosmos DB account, an Azure Cosmos DB database mapped to the SQL API, MongoDB API (preview), or Gremlin API (preview), and content in the database.

Make sure your Cosmos DB database contains data. The Import data wizard reads metadata and performs data sampling to infer an index schema, but it also loads data from Cosmos DB. If the data is missing, the wizard stops with this error "Error detecting index schema from data source: Could not build a prototype index because datasource 'emptycollection' returned no data".

2 - Start Import data wizard

You can start the wizard from the command bar in the Azure Cognitive Search service page, or if you're connecting to Cosmos DB SQL API you can click **Add Azure Cognitive Search** in the **Settings** section of your Cosmos DB account's left navigation pane.

3 - Set the data source

In the **data source** page, the source must be **Cosmos DB**, with the following specifications:

- **Name** is the name of the data source object. Once created, you can choose it for other workloads.
- **Cosmos DB account** should be in one of the following formats:
  1. The primary or secondary connection string from Cosmos DB with the following format:
     `AccountEndpoint=https://<Cosmos DB account name>.documents.azure.com;AccountKey=<Cosmos DB auth key>;` .
     - For version 3.2 and version 3.6 **MongoDB collections** use the following format for the Cosmos DB account in the Azure portal:
       ```
       AccountEndpoint=https://<Cosmos DB account name>.documents.azure.com;AccountKey=<Cosmos DB auth key>;ApiKind=MongoDb
       ```
     - For **Gremlin graphs and Cassandra tables**, sign up for the gated indexer preview to get access to the preview and information about how to format the credentials.
  2. A managed identity connection string with the following format that does not include an account key:
     ```
     ResourceId=/subscriptions/<your subscription ID>/resourceGroups/<your resource group name>/providers/Microsoft.DocumentDB/databaseAccounts/<your cosmos db account name>/;(ApiKind=[api-kind];)
     ```
     . To use this connection string format, follow the instructions for Setting up an indexer connection to a Cosmos DB database using a managed identity.
- **Database** is an existing database from the account.
- **Collection** is a container of documents. Documents must exist in order for import to succeed.
- **Query** can be blank if you want all documents, otherwise you can input a query that selects a document subset. **Query** is only available for the SQL API.

4 - Skip the "Enrich content" page in the wizard

Adding cognitive skills (or enrichment) is not an import requirement. Unless you have a specific need to add AI enrichment to your indexing pipeline, you should skip this step.

To skip the step, click the blue buttons at the bottom of the page for "Next" and "Skip".

5 - Set index attributes

In the **Index** page, you should see a list of fields with a data type and a series of checkboxes for setting index attributes. The wizard can generate a fields list based on metadata and by sampling the source data.

You can bulk-select attributes by clicking the checkbox at the top of an attribute column. Choose **Retrievable** and **Searchable** for every field that should be returned to a client app and subject to full text search processing. You'll notice that integers are not full text or fuzzy searchable (numbers are evaluated verbatim and are often useful in filters).

Review the description of index attributes and language analyzers for more information.

Take a moment to review your selections. Once you run the wizard, physical data structures are created and you won't be able to edit these fields without dropping and recreating all objects.

6 - Create indexer

Fully specified, the wizard creates three distinct objects in your search service. A data source object and index object are saved as named resources in your Azure Cognitive Search service. The last step creates an indexer object. Naming the indexer allows it to exist as a standalone resource, which you can schedule and manage independently of the index and data source object, created in the same wizard sequence.

If you are not familiar with indexers, an *indexer* is a resource in Azure Cognitive Search that crawls an external data source for searchable content. The output of the **Import data** wizard is an indexer that crawls your Cosmos DB data source, extracts searchable content, and imports it into an index on Azure Cognitive Search.

The following screenshot shows the default indexer configuration. You can switch to **Once** if you want to run the indexer one time. Click **Submit** to run the wizard and create all objects. Indexing commences immediately.

You can monitor data import in the portal pages. Progress notifications indicate indexing status and how many documents are uploaded.

When indexing is complete, you can use Search explorer to query your index.

> **NOTE**
> If you don't see the data you expect, you might need to set more attributes on more fields. Delete the index and indexer you just created, and step through the wizard again, modifying your selections for index attributes in step 5.

Use REST APIs

You can use the REST API to index Azure Cosmos DB data, following a three-part workflow common to all indexers in Azure Cognitive Search: create a data source, create an index, create an indexer. Data extraction from Cosmos DB occurs when you submit the Create Indexer request. After this request is finished, you will have a queryable index.

> **NOTE**
> For indexing data from Cosmos DB Gremlin API or Cosmos DB Cassandra API you must first request access to the gated previews by filling out this form. Once your request is processed, you will receive instructions for how to use the REST API version 2020-06-30-Preview to create the data source.

Earlier in this article it is mentioned that Azure Cosmos DB indexing and Azure Cognitive Search indexing indexing

are distinct operations. For Cosmos DB indexing, by default all documents are automatically indexed except with the Cassandra API. If you turn off automatic indexing, documents can be accessed only through their self-links or by queries by using the document ID. Azure Cognitive Search indexing requires Cosmos DB automatic indexing to be turned on in the collection that will be indexed by Azure Cognitive Search. When signing up for the Cosmos DB Cassandra API indexer preview, you'll be given instructions on how set up Cosmos DB indexing.

> **WARNING**
>
> Azure Cosmos DB is the next generation of DocumentDB. Previously with API version `2017-11-11` you could use the `documentdb` syntax. This meant that you could specify your data source type as `cosmosdb` or `documentdb`. Starting with API version `2019-05-06` both the Azure Cognitive Search APIs and Portal only support the `cosmosdb` syntax as instructed in this article. This means that the data source type must `cosmosdb` if you would like to connect to a Cosmos DB endpoint.

1 - Assemble inputs for the request

For each request, you must provide the service name and admin key for Azure Cognitive Search (in the POST header), and the storage account name and key for blob storage. You can use Postman or Visual Studio Code to send HTTP requests to Azure Cognitive Search.

Copy the following four values into Notepad so that you can paste them into a request:

- Azure Cognitive Search service name
- Azure Cognitive Search admin key
- Cosmos DB connection string

You can find these values in the portal:

1. In the portal pages for Azure Cognitive Search, copy the search service URL from the Overview page.
2. In the left navigation pane, click **Keys** and then copy either the primary or secondary key (they are equivalent).
3. Switch to the portal pages for your Cosmos storage account. In the left navigation pane, under **Settings**, click **Keys**. This page provides a URI, two sets of connection strings, and two sets of keys. Copy one of the connection strings to Notepad.

2 - Create a data source

A **data source** specifies the data to index, credentials, and policies for identifying changes in the data (such as modified or deleted documents inside your collection). The data source is defined as an independent resource so that it can be used by multiple indexers.

To create a data source, formulate a POST request:

```
    POST https://[service name].search.windows.net/datasources?api-version=2020-06-30
    Content-Type: application/json
    api-key: [Search service admin key]

 {
        "name": "mycosmosdbdatasource",
        "type": "cosmosdb",
        "credentials": {
            "connectionString":
"AccountEndpoint=https://myCosmosDbEndpoint.documents.azure.com;AccountKey=myCosmosDbAuthKey;Database=myCosmos
DbDatabaseId"
        },
        "container": { "name": "myCollection", "query": null },
        "dataChangeDetectionPolicy": {
            "@odata.type": "#Microsoft.Azure.Search.HighWaterMarkChangeDetectionPolicy",
            "highWaterMarkColumnName": "_ts"
        }
    }
```

The body of the request contains the data source definition, which should include the following fields:

FIELD	DESCRIPTION
name	Required. Choose any name to represent your data source object.
type	Required. Must be `cosmosdb`.

FIELD	DESCRIPTION
credentials	Required. Must either follow the Cosmos DB connection string format or a managed identity connection string format. For **SQL collections**, connection strings can follow either of the below formats: • `AccountEndpoint=https://<Cosmos DB account name>.documents.azure.com;AccountKey=<Cosmos DB auth key>;Database=<Cosmos DB database id>` • A managed identity connection string with the following format that does not include an account key: `ResourceId=/subscriptions/<your subscription ID>/resourceGroups/<your resource group name>/providers/Microsoft.DocumentDB/databaseAccounts/<your cosmos db account name>/;` . To use this connection string format, follow the instructions for Setting up an indexer connection to a Cosmos DB database using a managed identity. For version 3.2 and version 3.6 **MongoDB collections** use either of the following formats for the connection string: • `AccountEndpoint=https://<Cosmos DB account name>.documents.azure.com;AccountKey=<Cosmos DB auth key>;Database=<Cosmos DB database id>;ApiKind=MongoDb` • A managed identity connection string with the following format that does not include an account key: `ResourceId=/subscriptions/<your subscription ID>/resourceGroups/<your resource group name>/providers/Microsoft.DocumentDB/databaseAccounts/<your cosmos db account name>/;ApiKind=MongoDb;` . To use this connection string format, follow the instructions for Setting up an indexer connection to a Cosmos DB database using a managed identity. For **Gremlin graphs and Cassandra tables**, sign up for the gated indexer preview to get access to the preview and information about how to format the credentials. Avoid port numbers in the endpoint url. If you include the port number, Azure Cognitive Search will be unable to index your Azure Cosmos DB database.
container	Contains the following elements: **name**: Required. Specify the ID of the database collection to be indexed. **query**: Optional. You can specify a query to flatten an arbitrary JSON document into a flat schema that Azure Cognitive Search can index. For the MongoDB API, Gremlin API, and Cassandra API, queries are not supported.
dataChangeDetectionPolicy	Recommended. See Indexing Changed Documents section.
dataDeletionDetectionPolicy	Optional. See Indexing Deleted Documents section.

Using queries to shape indexed data

You can specify a SQL query to flatten nested properties or arrays, project JSON properties, and filter the data to be indexed.

> **WARNING**
> Custom queries are not supported for **MongoDB API**, **Gremlin API**, and **Cassandra API**: `container.query` parameter must be set to null or omitted. If you need to use a custom query, please let us know on User Voice.

Example document:

```
{
    "userId": 10001,
    "contact": {
        "firstName": "andy",
        "lastName": "hoh"
    },
    "company": "microsoft",
    "tags": ["azure", "cosmosdb", "search"]
}
```

Filter query:

```
SELECT * FROM c WHERE c.company = "microsoft" and c._ts >= @HighWaterMark ORDER BY c._ts
```

Flattening query:

```
SELECT c.id, c.userId, c.contact.firstName, c.contact.lastName, c.company, c._ts FROM c WHERE c._ts >=
@HighWaterMark ORDER BY c._ts
```

Projection query:

```
SELECT VALUE { "id":c.id, "Name":c.contact.firstName, "Company":c.company, "_ts":c._ts } FROM c WHERE c._ts >=
@HighWaterMark ORDER BY c._ts
```

Array flattening query:

```
SELECT c.id, c.userId, tag, c._ts FROM c JOIN tag IN c.tags WHERE c._ts >= @HighWaterMark ORDER BY c._ts
```

3 - Create a target search index

Create a target Azure Cognitive Search index if you don't have one already. The following example creates an index with an ID and description field:

```
    POST https://[service name].search.windows.net/indexes?api-version=2020-06-30
    Content-Type: application/json
    api-key: [Search service admin key]

{
      "name": "mysearchindex",
      "fields": [{
        "name": "id",
        "type": "Edm.String",
        "key": true,
        "searchable": false
      }, {
        "name": "description",
        "type": "Edm.String",
        "filterable": false,
        "sortable": false,
        "facetable": false,
        "suggestions": true
      }]
    }
```

Ensure that the schema of your target index is compatible with the schema of the source JSON documents or the output of your custom query projection.

> **NOTE**
>
> For partitioned collections, the default document key is Azure Cosmos DB's `_rid` property, which Azure Cognitive Search automatically renames to `rid` because field names cannot start with an underscore character. Also, Azure Cosmos DB `_rid` values contain characters that are invalid in Azure Cognitive Search keys. For this reason, the `_rid` values are Base64 encoded.
>
> For MongoDB collections, Azure Cognitive Search automatically renames the `_id` property to `id`.

Mapping between JSON Data Types and Azure Cognitive Search Data Types

JSON DATA TYPE	COMPATIBLE TARGET INDEX FIELD TYPES
Bool	Edm.Boolean, Edm.String
Numbers that look like integers	Edm.Int32, Edm.Int64, Edm.String
Numbers that look like floating-points	Edm.Double, Edm.String
String	Edm.String
Arrays of primitive types, for example ["a", "b", "c"]	Collection(Edm.String)
Strings that look like dates	Edm.DateTimeOffset, Edm.String
GeoJSON objects, for example { "type": "Point", "coordinates": [long, lat] }	Edm.GeographyPoint
Other JSON objects	N/A

4 - Configure and run the indexer

Once the index and data source have been created, you're ready to create the indexer:

```
POST https://[service name].search.windows.net/indexers?api-version=2020-06-30
Content-Type: application/json
api-key: [admin key]

{
  "name" : "mycosmosdbindexer",
  "dataSourceName" : "mycosmosdbdatasource",
  "targetIndexName" : "mysearchindex",
  "schedule" : { "interval" : "PT2H" }
}
```

This indexer runs every two hours (schedule interval is set to "PT2H"). To run an indexer every 30 minutes, set the interval to "PT30M". The shortest supported interval is 5 minutes. The schedule is optional - if omitted, an indexer runs only once when it's created. However, you can run an indexer on-demand at any time.

For more details on the Create Indexer API, check out Create Indexer.

For more information about defining indexer schedules, see How to schedule indexers for Azure Cognitive Search.

Use .NET

The generally available .NET SDK has full parity with the generally available REST API. We recommend that you review the previous REST API section to learn concepts, workflow, and requirements. You can then refer to following .NET API reference documentation to implement a JSON indexer in managed code.

- azure.search.documents.indexes.models.searchindexerdatasourceconnection
- azure.search.documents.indexes.models.searchindexerdatasourcetype
- azure.search.documents.indexes.models.searchindex
- azure.search.documents.indexes.models.searchindexer

Indexing changed documents

The purpose of a data change detection policy is to efficiently identify changed data items. Currently, the only supported policy is the `HighWaterMarkChangeDetectionPolicy` using the `_ts` (timestamp) property provided by Azure Cosmos DB, which is specified as follows:

```
{
    "@odata.type" : "#Microsoft.Azure.Search.HighWaterMarkChangeDetectionPolicy",
    "highWaterMarkColumnName" : "_ts"
}
```

Using this policy is highly recommended to ensure good indexer performance.

If you are using a custom query, make sure that the `_ts` property is projected by the query.

Incremental progress and custom queries

Incremental progress during indexing ensures that if indexer execution is interrupted by transient failures or execution time limit, the indexer can pick up where it left off next time it runs, instead of having to reindex the entire collection from scratch. This is especially important when indexing large collections.

To enable incremental progress when using a custom query, ensure that your query orders the results by the `_ts` column. This enables periodic check-pointing that Azure Cognitive Search uses to provide incremental progress in the presence of failures.

In some cases, even if your query contains an `ORDER BY [collection alias]._ts` clause, Azure Cognitive Search may not infer that the query is ordered by the `_ts`. You can tell Azure Cognitive Search that results are ordered by

using the `assumeOrderByHighWaterMarkColumn` configuration property. To specify this hint, create or update your indexer as follows:

```
{
    ... other indexer definition properties
    "parameters" : {
           "configuration" : { "assumeOrderByHighWaterMarkColumn" : true } }
   }
```

Indexing deleted documents

When rows are deleted from the collection, you normally want to delete those rows from the search index as well. The purpose of a data deletion detection policy is to efficiently identify deleted data items. Currently, the only supported policy is the `Soft Delete` policy (deletion is marked with a flag of some sort), which is specified as follows:

```
   {
       "@odata.type" : "#Microsoft.Azure.Search.SoftDeleteColumnDeletionDetectionPolicy",
       "softDeleteColumnName" : "the property that specifies whether a document was deleted",
       "softDeleteMarkerValue" : "the value that identifies a document as deleted"
   }
```

If you are using a custom query, make sure that the property referenced by `softDeleteColumnName` is projected by the query.

The following example creates a data source with a soft-deletion policy:

```
 POST https://[service name].search.windows.net/datasources?api-version=2020-06-30
    Content-Type: application/json
    api-key: [Search service admin key]

    {
        "name": "mycosmosdbdatasource",
        "type": "cosmosdb",
        "credentials": {
            "connectionString":
"AccountEndpoint=https://myCosmosDbEndpoint.documents.azure.com;AccountKey=myCosmosDbAuthKey;Database=myCosmos
DbDatabaseId"
        },
        "container": { "name": "myCosmosDbCollectionId" },
        "dataChangeDetectionPolicy": {
            "@odata.type": "#Microsoft.Azure.Search.HighWaterMarkChangeDetectionPolicy",
            "highWaterMarkColumnName": "_ts"
        },
        "dataDeletionDetectionPolicy": {
            "@odata.type": "#Microsoft.Azure.Search.SoftDeleteColumnDeletionDetectionPolicy",
            "softDeleteColumnName": "isDeleted",
            "softDeleteMarkerValue": "true"
        }
    }
```

Next steps

Congratulations! You have learned how to integrate Azure Cosmos DB with Azure Cognitive Search using an indexer.

- To learn more about Azure Cosmos DB, see the Azure Cosmos DB service page.
- To learn more about Azure Cognitive Search, see the Search service page.

Serverless database computing using Azure Cosmos DB and Azure Functions

12/4/2020 • 9 minutes to read • Edit Online

APPLIES TO: SQL API

Serverless computing is all about the ability to focus on individual pieces of logic that are repeatable and stateless. These pieces require no infrastructure management and they consume resources only for the seconds, or milliseconds, they run for. At the core of the serverless computing movement are functions, which are made available in the Azure ecosystem by Azure Functions. To learn about other serverless execution environments in Azure see serverless in Azure page.

With the native integration between Azure Cosmos DB and Azure Functions, you can create database triggers, input bindings, and output bindings directly from your Azure Cosmos DB account. Using Azure Functions and Azure Cosmos DB, you can create and deploy event-driven serverless apps with low-latency access to rich data for a global user base.

Overview

Azure Cosmos DB and Azure Functions enable you to integrate your databases and serverless apps in the following ways:

- Create an event-driven **Azure Functions trigger for Cosmos DB**. This trigger relies on change feed streams to monitor your Azure Cosmos container for changes. When any changes are made to a container, the change feed stream is sent to the trigger, which invokes the Azure Function.
- Alternatively, bind an Azure Function to an Azure Cosmos container using an **input binding**. Input bindings read data from a container when a function executes.
- Bind a function to an Azure Cosmos container using an **output binding**. Output bindings write data to a container when a function completes.

> **NOTE**
>
> Currently, Azure Functions trigger, input bindings, and output bindings for Cosmos DB are supported for use with the SQL API only. For all other Azure Cosmos DB APIs, you should access the database from your function by using the static client for your API.

The following diagram illustrates each of these three integrations:

The Azure Functions trigger, input binding, and output binding for Azure Cosmos DB can be used in the following combinations:

- An Azure Functions trigger for Cosmos DB can be used with an output binding to a different Azure Cosmos container. After a function performs an action on an item in the change feed you can write it to another container (writing it to the same container it came from would effectively create a recursive loop). Or, you can use an Azure Functions trigger for Cosmos DB to effectively migrate all changed items from one container to a different container, with the use of an output binding.
- Input bindings and output bindings for Azure Cosmos DB can be used in the same Azure Function. This works well in cases when you want to find certain data with the input binding, modify it in the Azure Function, and then save it to the same container or a different container, after the modification.
- An input binding to an Azure Cosmos container can be used in the same function as an Azure Functions trigger

for Cosmos DB, and can be used with or without an output binding as well. You could use this combination to apply up-to-date currency exchange information (pulled in with an input binding to an exchange container) to the change feed of new orders in your shopping cart service. The updated shopping cart total, with the current currency conversion applied, can be written to a third container using an output binding.

Use cases

The following use cases demonstrate a few ways you can make the most of your Azure Cosmos DB data - by connecting your data to event-driven Azure Functions.

IoT use case - Azure Functions trigger and output binding for Cosmos DB

In IoT implementations, you can invoke a function when the check engine light is displayed in a connected car.

Implementation: Use an Azure Functions trigger and output binding for Cosmos DB

1. An **Azure Functions trigger for Cosmos DB** is used to trigger events related to car alerts, such as the check engine light coming on in a connected car.
2. When the check engine light comes, the sensor data is sent to Azure Cosmos DB.
3. Azure Cosmos DB creates or updates new sensor data documents, then those changes are streamed to the Azure Functions trigger for Cosmos DB.
4. The trigger is invoked on every data-change to the sensor data collection, as all changes are streamed via the change feed.
5. A threshold condition is used in the function to send the sensor data to the warranty department.
6. If the temperature is also over a certain value, an alert is also sent to the owner.
7. The **output binding** on the function updates the car record in another Azure Cosmos container to store information about the check engine event.

The following image shows the code written in the Azure portal for this trigger.

Financial use case - Timer trigger and input binding

In financial implementations, you can invoke a function when a bank account balance falls under a certain amount.

Implementation: A timer trigger with an Azure Cosmos DB input binding

1. Using a timer trigger, you can retrieve the bank account balance information stored in an Azure Cosmos

container at timed intervals using an **input binding**.

2. If the balance is below the low balance threshold set by the user, then follow up with an action from the Azure Function.
3. The output binding can be a SendGrid integration that sends an email from a service account to the email addresses identified for each of the low balance accounts.

The following images show the code in the Azure portal for this scenario.

index.js Save ▶ Save and run

```javascript
module.exports = function (context, myTimer) {
    var timeStamp = new Date().toISOString();
    context.log('JavaScript timer trigger function ran!', timeStamp);

    var accounts = context.bindings.accounts;
    var lowBalanceThreshold = 100;

    for (var i = 0; i < accounts.length; i++) {
        var account = accounts[i];
        var accountNumber = account['$v']['number']['$v'];
        var accountBalance = account['$v']['balance']['$v'];

        context.log('The account #' + accountNumber + 'has a balance of: $' + accountBalance);

        if (accountBalance < lowBalanceThreshold){
            // Notify account owners about low balance.
        }
    }
    context.done();
};
```

run.csx Save ▶ Save and run

```csharp
#r "Microsoft.Azure.Documents.Client"
using System;
using System.Collections.Generic;
using Microsoft.Azure.Documents;

public static void Run(IReadOnlyList<Document> documents,  TraceWriter log)
{
    double lowBalanceThreshold = 100;
    foreach (var doc in documents) {
        if (doc.GetPropertyValue<double>("balance") < lowBalanceThreshold) {
            //Send the mail to user
            log.Verbose ("Balance low");
        }
    }
}

```

Gaming use case - Azure Functions trigger and output binding for Cosmos DB

In gaming, when a new user is created you can search for other users who might know them by using the Azure Cosmos DB Gremlin API. You can then write the results to an Azure Cosmos DB or SQL database for easy retrieval.

Implementation: Use an Azure Functions trigger and output binding for Cosmos DB

1. Using an Azure Cosmos DB graph database to store all users, you can create a new function with an Azure Functions trigger for Cosmos DB.
2. Whenever a new user is inserted, the function is invoked, and then the result is stored using an **output binding**.
3. The function queries the graph database to search for all the users that are directly related to the new user and returns that dataset to the function.
4. This data is then stored in an Azure Cosmos DB which can then be easily retrieved by any front-end application that shows the new user their connected friends.

Retail use case - Multiple functions

In retail implementations, when a user adds an item to their basket you now have the flexibility to create and invoke functions for optional business pipeline components.

Implementation: Multiple Azure Functions triggers for Cosmos DB listening to one container

1. You can create multiple Azure Functions by adding Azure Functions triggers for Cosmos DB to each - all of which listen to the same change feed of shopping cart data. Note that when multiple functions listen to the same change feed, a new lease collection is required for each function. For more information about lease collections, see Understanding the Change Feed Processor library.
2. Whenever a new item is added to a users shopping cart, each function is independently invoked by the change feed from the shopping cart container.
   - One function may use the contents of the current basket to change the display of other items the user might be interested in.
   - Another function may update inventory totals.
   - Another function may send customer information for certain products to the marketing department, who sends them a promotional mailer.

     Any department can create an Azure Functions for Cosmos DB by listening to the change feed, and be sure they won't delay critical order processing events in the process.

In all of these use cases, because the function has decoupled the app itself, you don't need to spin up new app instances all the time. Instead, Azure Functions spins up individual functions to complete discrete processes as needed.

Tooling

Native integration between Azure Cosmos DB and Azure Functions is available in the Azure portal and in Visual Studio 2019.

- In the Azure Functions portal, you can create a trigger. For quickstart instructions, see Create an Azure Functions trigger for Cosmos DB in the Azure portal.
- In the Azure Cosmos DB portal, you can add an Azure Functions trigger for Cosmos DB to an existing Azure Function app in the same resource group.
- In Visual Studio 2019, you can create the trigger using the Azure Functions Tools:

Why choose Azure Functions integration for serverless computing?

Azure Functions provides the ability to create scalable units of work, or concise pieces of logic that can be run on demand, without provisioning or managing infrastructure. By using Azure Functions, you don't have to create a full-blown app to respond to changes in your Azure Cosmos database, you can create small reusable functions for specific tasks. In addition, you can also use Azure Cosmos DB data as the input or output to an Azure Function in response to event such as an HTTP requests or a timed trigger.

Azure Cosmos DB is the recommended database for your serverless computing architecture for the following reasons:

- **Instant access to all your data**: You have granular access to every value stored because Azure Cosmos DB automatically indexes all data by default, and makes those indexes immediately available. This means you are able to constantly query, update, and add new items to your database and have instant access via Azure Functions.

- **Schemaless**. Azure Cosmos DB is schemaless - so it's uniquely able to handle any data output from an Azure Function. This "handle anything" approach makes it straightforward to create a variety of Functions that all output to Azure Cosmos DB.
- **Scalable throughput**. Throughput can be scaled up and down instantly in Azure Cosmos DB. If you have hundreds or thousands of Functions querying and writing to the same container, you can scale up your RU/s to handle the load. All functions can work in parallel using your allocated RU/s and your data is guaranteed to be consistent.
- **Global replication**. You can replicate Azure Cosmos DB data around the globe to reduce latency, geo-locating your data closest to where your users are. As with all Azure Cosmos DB queries, data from event-driven triggers is read data from the Azure Cosmos DB closest to the user.

If you're looking to integrate with Azure Functions to store data and don't need deep indexing or if you need to store attachments and media files, the Azure Blob Storage trigger may be a better option.

Benefits of Azure Functions:

- **Event-driven**. Azure Functions are event-driven and can listen to a change feed from Azure Cosmos DB. This means you don't need to create listening logic, you just keep an eye out for the changes you're listening for.
- **No limits**. Functions execute in parallel and the service spins up as many as you need. You set the parameters.
- **Good for quick tasks**. The service spins up new instances of functions whenever an event fires and closes them as soon as the function completes. You only pay for the time your functions are running.

If you're not sure whether Flow, Logic Apps, Azure Functions, or WebJobs are best for your implementation, see Choose between Flow, Logic Apps, Functions, and WebJobs.

Next steps

Now let's connect Azure Cosmos DB and Azure Functions for real:

- Create an Azure Functions trigger for Cosmos DB in the Azure portal
- Create an Azure Functions HTTP trigger with an Azure Cosmos DB input binding
- Azure Cosmos DB bindings and triggers

Azure Cosmos DB bindings for Azure Functions 1.x

11/2/2020 • 26 minutes to read • Edit Online

This article explains how to work with Azure Cosmos DB bindings in Azure Functions. Azure Functions supports trigger, input, and output bindings for Azure Cosmos DB.

> **NOTE**
>
> This article is for Azure Functions 1.x. For information about how to use these bindings in Functions 2.x and higher, see Azure Cosmos DB bindings for Azure Functions 2.x.
>
> This binding was originally named DocumentDB. In Functions version 1.x, only the trigger was renamed Cosmos DB; the input binding, output binding, and NuGet package retain the DocumentDB name.

This is reference information for Azure Functions developers. If you're new to Azure Functions, start with the following resources:

- Create your first function: C#, JavaScript, Java, or Python.
- Azure Functions developer reference.
- Language-specific reference: C#, C# script, F#, Java, JavaScript, or Python.
- Azure Functions triggers and bindings concepts.
- Code and test Azure Functions locally.

> **NOTE**
>
> Azure Cosmos DB bindings are only supported for use with the SQL API. For all other Azure Cosmos DB APIs, you should access the database from your function by using the static client for your API, including Azure Cosmos DB's API for MongoDB, Cassandra API, Gremlin API, and Table API.

Packages - Functions 1.x

The Azure Cosmos DB bindings for Functions version 1.x are provided in the Microsoft.Azure.WebJobs.Extensions.DocumentDB NuGet package, version 1.x. Source code for the bindings is in the azure-webjobs-sdk-extensions GitHub repository.

The following table tells how to add support for this binding in each development environment.

DEVELOPMENT ENVIRONMENT	TO ADD SUPPORT IN FUNCTIONS 1.X
Local development - C# class library	Install the package
Local development - C# script, JavaScript, F#	Automatic
Portal development	Automatic

Trigger

The Azure Cosmos DB Trigger uses the Azure Cosmos DB Change Feed to listen for inserts and updates across partitions. The change feed publishes inserts and updates, not deletions.

Trigger - example

- C#
- C# Script
- JavaScript

The following example shows a C# function that is invoked when there are inserts or updates in the specified database and collection.

```
using Microsoft.Azure.Documents;
using Microsoft.Azure.WebJobs;
using Microsoft.Azure.WebJobs.Host;
using System.Collections.Generic;

namespace CosmosDBSamplesV1
{
    public static class CosmosTrigger
    {
        [FunctionName("CosmosTrigger")]
        public static void Run([CosmosDBTrigger(
            databaseName: "ToDoItems",
            collectionName: "Items",
            ConnectionStringSetting = "CosmosDBConnection",
            LeaseCollectionName = "leases",
            CreateLeaseCollectionIfNotExists = true)]IReadOnlyList<Document> documents,
            TraceWriter log)
        {
            if (documents != null && documents.Count > 0)
            {
                log.Info($"Documents modified: {documents.Count}");
                log.Info($"First document Id: {documents[0].Id}");
            }
        }
    }
}
```

Trigger - attributes

- C#
- C# Script
- JavaScript

In C# class libraries, use the CosmosDBTrigger attribute.

The attribute's constructor takes the database name and collection name. For information about those settings and other properties that you can configure, see Trigger - configuration. Here's a `CosmosDBTrigger` attribute example in a method signature:

```
    [FunctionName("DocumentUpdates")]
    public static void Run(
        [CosmosDBTrigger("database", "collection", ConnectionStringSetting = "myCosmosDB")]
    IReadOnlyList<Document> documents,
        TraceWriter log)
    {
        ...
    }
```

For a complete example, see Trigger - C# example.

Trigger - configuration

The following table explains the binding configuration properties that you set in the *function.json* file and the `CosmosDBTrigger` attribute.

FUNCTION.JSON PROPERTY	ATTRIBUTE PROPERTY	DESCRIPTION
type	n/a	Must be set to `cosmosDBTrigger`.
direction	n/a	Must be set to `in`. This parameter is set automatically when you create the trigger in the Azure portal.
name	n/a	The variable name used in function code that represents the list of documents with changes.
connectionStringSetting	**ConnectionStringSetting**	The name of an app setting that contains the connection string used to connect to the Azure Cosmos DB account being monitored.
databaseName	**DatabaseName**	The name of the Azure Cosmos DB database with the collection being monitored.
collectionName	**CollectionName**	The name of the collection being monitored.
leaseConnectionStringSetting	**LeaseConnectionStringSetting**	(Optional) The name of an app setting that contains the connection string to the service which holds the lease collection. When not set, the `connectionStringSetting` value is used. This parameter is automatically set when the binding is created in the portal. The connection string for the leases collection must have write permissions.
leaseDatabaseName	**LeaseDatabaseName**	(Optional) The name of the database that holds the collection used to store leases. When not set, the value of the `databaseName` setting is used. This parameter is automatically set when the binding is created in the portal.
leaseCollectionName	**LeaseCollectionName**	(Optional) The name of the collection used to store leases. When not set, the value `leases` is used.
createLeaseCollectionIfNotExists	**CreateLeaseCollectionIfNotExists**	(Optional) When set to `true`, the leases collection is automatically created when it doesn't already exist. The default value is `false`.

FUNCTION.JSON PROPERTY	ATTRIBUTE PROPERTY	DESCRIPTION
leasesCollectionThroughput	**LeasesCollectionThroughput**	(Optional) Defines the amount of Request Units to assign when the leases collection is created. This setting is only used When `createLeaseCollectionIfNotExists` is set to `true`. This parameter is automatically set when the binding is created using the portal.
leaseCollectionPrefix	**LeaseCollectionPrefix**	(Optional) When set, it adds a prefix to the leases created in the Lease collection for this Function, effectively allowing two separate Azure Functions to share the same Lease collection by using different prefixes.
feedPollDelay	**FeedPollDelay**	(Optional) When set, it defines, in milliseconds, the delay in between polling a partition for new changes on the feed, after all current changes are drained. Default is 5000 (5 seconds).
leaseAcquireInterval	**LeaseAcquireInterval**	(Optional) When set, it defines, in milliseconds, the interval to kick off a task to compute if partitions are distributed evenly among known host instances. Default is 13000 (13 seconds).
leaseExpirationInterval	**LeaseExpirationInterval**	(Optional) When set, it defines, in milliseconds, the interval for which the lease is taken on a lease representing a partition. If the lease is not renewed within this interval, it will cause it to expire and ownership of the partition will move to another instance. Default is 60000 (60 seconds).
leaseRenewInterval	**LeaseRenewInterval**	(Optional) When set, it defines, in milliseconds, the renew interval for all leases for partitions currently held by an instance. Default is 17000 (17 seconds).
checkpointFrequency	**CheckpointFrequency**	(Optional) When set, it defines, in milliseconds, the interval between lease checkpoints. Default is always after each Function call.
maxItemsPerInvocation	**MaxItemsPerInvocation**	(Optional) When set, it customizes the maximum amount of items received per Function call.

FUNCTION.JSON PROPERTY	ATTRIBUTE PROPERTY	DESCRIPTION
startFromBeginning	**StartFromBeginning**	(Optional) When set, it tells the Trigger to start reading changes from the beginning of the history of the collection instead of the current time. This only works the first time the Trigger starts, as in subsequent runs, the checkpoints are already stored. Setting this to `true` when there are leases already created has no effect.

When you're developing locally, app settings go into the local.settings.json file.

Trigger - usage

The trigger requires a second collection that it uses to store *leases* over the partitions. Both the collection being monitored and the collection that contains the leases must be available for the trigger to work.

> **IMPORTANT**
> If multiple functions are configured to use a Cosmos DB trigger for the same collection, each of the functions should use a dedicated lease collection or specify a different `LeaseCollectionPrefix` for each function. Otherwise, only one of the functions will be triggered. For information about the prefix, see the Configuration section.

The trigger doesn't indicate whether a document was updated or inserted, it just provides the document itself. If you need to handle updates and inserts differently, you could do that by implementing timestamp fields for insertion or update.

Input

The Azure Cosmos DB input binding uses the SQL API to retrieve one or more Azure Cosmos DB documents and passes them to the input parameter of the function. The document ID or query parameters can be determined based on the trigger that invokes the function.

Input - example

- C#
- C# Script
- JavaScript

This section contains the following examples:

- Queue trigger, look up ID from JSON
- HTTP trigger, look up ID from query string
- HTTP trigger, look up ID from route data
- HTTP trigger, look up ID from route data, using SqlQuery
- HTTP trigger, get multiple docs, using SqlQuery
- HTTP trigger, get multiple docs, using DocumentClient

The examples refer to a simple `ToDoItem` type:

```
namespace CosmosDBSamplesV1
{
    public class ToDoItem
    {
        public string Id { get; set; }
        public string Description { get; set; }
    }
}
```

Queue trigger, look up ID from JSON

The following example shows a C# function that retrieves a single document. The function is triggered by a queue message that contains a JSON object. The queue trigger parses the JSON into an object named `ToDoItemLookup`, which contains the ID to look up. That ID is used to retrieve a `ToDoItem` document from the specified database and collection.

```
namespace CosmosDBSamplesV1
{
    public class ToDoItemLookup
    {
        public string ToDoItemId { get; set; }
    }
}
```

```
using Microsoft.Azure.WebJobs;
using Microsoft.Azure.WebJobs.Host;

namespace CosmosDBSamplesV1
{
    public static class DocByIdFromJSON
    {
        [FunctionName("DocByIdFromJSON")]
        public static void Run(
            [QueueTrigger("todoqueueforlookup")] ToDoItemLookup toDoItemLookup,
            [DocumentDB(
                databaseName: "ToDoItems",
                collectionName: "Items",
                ConnectionStringSetting = "CosmosDBConnection",
                Id = "{ToDoItemId}")]ToDoItem toDoItem,
            TraceWriter log)
        {
            log.Info($"C# Queue trigger function processed Id={toDoItemLookup?.ToDoItemId}");

            if (toDoItem == null)
            {
                log.Info($"ToDo item not found");
            }
            else
            {
                log.Info($"Found ToDo item, Description={toDoItem.Description}");
            }
        }
    }
}
```

HTTP trigger, look up ID from query string

The following example shows a C# function that retrieves a single document. The function is triggered by an HTTP request that uses a query string to specify the ID to look up. That ID is used to retrieve a `ToDoItem` document from the specified database and collection.

```
using Microsoft.Azure.WebJobs;
using Microsoft.Azure.WebJobs.Extensions.Http;
using Microsoft.Azure.WebJobs.Host;
using System.Net;
using System.Net.Http;

namespace CosmosDBSamplesV1
{
    public static class DocByIdFromQueryString
    {
        [FunctionName("DocByIdFromQueryString")]
        public static HttpResponseMessage Run(
            [HttpTrigger(AuthorizationLevel.Anonymous, "get", "post", Route = null)]HttpRequestMessage req,
            [DocumentDB(
                databaseName: "ToDoItems",
                collectionName: "Items",
                ConnectionStringSetting = "CosmosDBConnection",
                Id = "{Query.id}")] ToDoItem toDoItem,
            TraceWriter log)
        {
            log.Info("C# HTTP trigger function processed a request.");
            if (toDoItem == null)
            {
                log.Info($"ToDo item not found");
            }
            else
            {
                log.Info($"Found ToDo item, Description={toDoItem.Description}");
            }
            return req.CreateResponse(HttpStatusCode.OK);
        }
    }
}
```

HTTP trigger, look up ID from route data

The following example shows a C# function that retrieves a single document. The function is triggered by an HTTP request that uses route data to specify the ID to look up. That ID is used to retrieve a `ToDoItem` document from the specified database and collection.

```
using Microsoft.Azure.WebJobs;
using Microsoft.Azure.WebJobs.Extensions.Http;
using Microsoft.Azure.WebJobs.Host;
using System.Net;
using System.Net.Http;

namespace CosmosDBSamplesV1
{
    public static class DocByIdFromRouteData
    {
        [FunctionName("DocByIdFromRouteData")]
        public static HttpResponseMessage Run(
            [HttpTrigger(
                AuthorizationLevel.Anonymous, "get", "post",
                Route = "todoitems/{id}")]HttpRequestMessage req,
            [DocumentDB(
                databaseName: "ToDoItems",
                collectionName: "Items",
                ConnectionStringSetting = "CosmosDBConnection",
                Id = "{id}")] ToDoItem toDoItem,
            TraceWriter log)
        {
            log.Info("C# HTTP trigger function processed a request.");

            if (toDoItem == null)
            {
                log.Info($"ToDo item not found");
            }
            else
            {
                log.Info($"Found ToDo item, Description={toDoItem.Description}");
            }
            return req.CreateResponse(HttpStatusCode.OK);
        }
    }
}
```

Skip input examples

HTTP trigger, look up ID from route data, using SqlQuery

The following example shows a C# function that retrieves a single document. The function is triggered by an HTTP request that uses route data to specify the ID to look up. That ID is used to retrieve a `ToDoItem` document from the specified database and collection.

```
using Microsoft.Azure.WebJobs;
using Microsoft.Azure.WebJobs.Extensions.Http;
using Microsoft.Azure.WebJobs.Host;
using System.Collections.Generic;
using System.Net;
using System.Net.Http;

namespace CosmosDBSamplesV1
{
    public static class DocByIdFromRouteDataUsingSqlQuery
    {
        [FunctionName("DocByIdFromRouteDataUsingSqlQuery")]
        public static HttpResponseMessage Run(
            [HttpTrigger(AuthorizationLevel.Anonymous, "get", "post",
                Route = "todoitems2/{id}")]HttpRequestMessage req,
            [DocumentDB(
                databaseName: "ToDoItems",
                collectionName: "Items",
                ConnectionStringSetting = "CosmosDBConnection",
                SqlQuery = "select * from ToDoItems r where r.id = {id}")] IEnumerable<ToDoItem> toDoItems,
            TraceWriter log)
        {
            log.Info("C# HTTP trigger function processed a request.");
            foreach (ToDoItem toDoItem in toDoItems)
            {
                log.Info(toDoItem.Description);
            }
            return req.CreateResponse(HttpStatusCode.OK);
        }
    }
}
```

Skip input examples

HTTP trigger, get multiple docs, using SqlQuery

The following example shows a C# function that retrieves a list of documents. The function is triggered by an HTTP request. The query is specified in the `SqlQuery` attribute property.

```
using Microsoft.Azure.WebJobs;
using Microsoft.Azure.WebJobs.Extensions.Http;
using Microsoft.Azure.WebJobs.Host;
using System.Collections.Generic;
using System.Net;
using System.Net.Http;

namespace CosmosDBSamplesV1
{
    public static class DocsBySqlQuery
    {
        [FunctionName("DocsBySqlQuery")]
        public static HttpResponseMessage Run(
            [HttpTrigger(AuthorizationLevel.Anonymous, "get", "post", Route = null)]
                HttpRequestMessage req,
            [DocumentDB(
                databaseName: "ToDoItems",
                collectionName: "Items",
                ConnectionStringSetting = "CosmosDBConnection",
                SqlQuery = "SELECT top 2 * FROM c order by c._ts desc")]
                IEnumerable<ToDoItem> toDoItems,
            TraceWriter log)
        {
            log.Info("C# HTTP trigger function processed a request.");
            foreach (ToDoItem toDoItem in toDoItems)
            {
                log.Info(toDoItem.Description);
            }
            return req.CreateResponse(HttpStatusCode.OK);
        }
    }
}
```

Skip input examples

HTTP trigger, get multiple docs, using DocumentClient (C#)

The following example shows a C# function that retrieves a list of documents. The function is triggered by an HTTP request. The code uses a `DocumentClient` instance provided by the Azure Cosmos DB binding to read a list of documents. The `DocumentClient` instance could also be used for write operations.

```
using Microsoft.Azure.Documents.Client;
using Microsoft.Azure.Documents.Linq;
using Microsoft.Azure.WebJobs;
using Microsoft.Azure.WebJobs.Extensions.Http;
using Microsoft.Azure.WebJobs.Host;
using System;
using System.Linq;
using System.Net;
using System.Net.Http;
using System.Threading.Tasks;

namespace CosmosDBSamplesV1
{
    public static class DocsByUsingDocumentClient
    {
        [FunctionName("DocsByUsingDocumentClient")]
        public static async Task<HttpResponseMessage> Run(
            [HttpTrigger(AuthorizationLevel.Anonymous, "get", "post", Route = null)]HttpRequestMessage req,
            [DocumentDB(
                databaseName: "ToDoItems",
                collectionName: "Items",
                ConnectionStringSetting = "CosmosDBConnection")] DocumentClient client,
            TraceWriter log)
        {
            log.Info("C# HTTP trigger function processed a request.");

            Uri collectionUri = UriFactory.CreateDocumentCollectionUri("ToDoItems", "Items");
            string searchterm = req.GetQueryNameValuePairs()
                .FirstOrDefault(q => string.Compare(q.Key, "searchterm", true) == 0)
                .Value;

            if (searchterm == null)
            {
                return req.CreateResponse(HttpStatusCode.NotFound);
            }

            log.Info($"Searching for word: {searchterm} using Uri: {collectionUri.ToString()}");
            IDocumentQuery<ToDoItem> query = client.CreateDocumentQuery<ToDoItem>(collectionUri)
                .Where(p => p.Description.Contains(searchterm))
                .AsDocumentQuery();

            while (query.HasMoreResults)
            {
                foreach (ToDoItem result in await query.ExecuteNextAsync())
                {
                    log.Info(result.Description);
                }
            }
            return req.CreateResponse(HttpStatusCode.OK);
        }
    }
}
```

Input - attributes

- C#
- C# Script
- JavaScript

In C# class libraries, use the DocumentDB attribute.

The attribute's constructor takes the database name and collection name. For information about those settings and other properties that you can configure, see the following configuration section.

Input - configuration

The following table explains the binding configuration properties that you set in the *function.json* file and the `DocumentDB` attribute.

FUNCTION.JSON PROPERTY	ATTRIBUTE PROPERTY	DESCRIPTION
type	n/a	Must be set to `documentdb`.
direction	n/a	Must be set to `in`.
name	n/a	Name of the binding parameter that represents the document in the function.
databaseName	**DatabaseName**	The database containing the document.
collectionName	**CollectionName**	The name of the collection that contains the document.
id	**Id**	The ID of the document to retrieve. This property supports binding expressions. Don't set both the **id** and **sqlQuery** properties. If you don't set either one, the entire collection is retrieved.
sqlQuery	**SqlQuery**	An Azure Cosmos DB SQL query used for retrieving multiple documents. The property supports runtime bindings, as in this example: `SELECT * FROM c where c.departmentId = {departmentId}`. Don't set both the **id** and **sqlQuery** properties. If you don't set either one, the entire collection is retrieved.
connection	**ConnectionStringSetting**	The name of the app setting containing your Azure Cosmos DB connection string.
partitionKey	**PartitionKey**	Specifies the partition key value for the lookup. May include binding parameters.

When you're developing locally, app settings go into the local.settings.json file.

Input - usage

- C#
- C# Script
- JavaScript

When the function exits successfully, any changes made to the input document via named input parameters are automatically persisted.

Output

The Azure Cosmos DB output binding lets you write a new document to an Azure Cosmos DB database using the SQL API.

Output - example

- C#
- C# Script
- JavaScript

This section contains the following examples:

- Queue trigger, write one doc
- Queue trigger, write docs using `IAsyncCollector`

The examples refer to a simple `ToDoItem` type:

```
namespace CosmosDBSamplesV1
{
    public class ToDoItem
    {
        public string Id { get; set; }
        public string Description { get; set; }
    }
}
```

Queue trigger, write one doc

The following example shows a C# function that adds a document to a database, using data provided in message from Queue storage.

```
using Microsoft.Azure.WebJobs;
using Microsoft.Azure.WebJobs.Host;
using System;

namespace CosmosDBSamplesV1
{
    public static class WriteOneDoc
    {
        [FunctionName("WriteOneDoc")]
        public static void Run(
            [QueueTrigger("todoqueueforwrite")] string queueMessage,
            [DocumentDB(
                databaseName: "ToDoItems",
                collectionName: "Items",
                ConnectionStringSetting = "CosmosDBConnection")]out dynamic document,
            TraceWriter log)
        {
            document = new { Description = queueMessage, id = Guid.NewGuid() };

            log.Info($"C# Queue trigger function inserted one row");
            log.Info($"Description={queueMessage}");
        }
    }
}
```

Queue trigger, write docs using IAsyncCollector

The following example shows a C# function that adds a collection of documents to a database, using data provided in a queue message JSON.

```
using Microsoft.Azure.WebJobs;
using Microsoft.Azure.WebJobs.Host;
using System.Threading.Tasks;

namespace CosmosDBSamplesV1
{
    public static class WriteDocsIAsyncCollector
    {
        [FunctionName("WriteDocsIAsyncCollector")]
        public static async Task Run(
            [QueueTrigger("todoqueueforwritemulti")] ToDoItem[] toDoItemsIn,
            [DocumentDB(
                databaseName: "ToDoItems",
                collectionName: "Items",
                ConnectionStringSetting = "CosmosDBConnection")]
                IAsyncCollector<ToDoItem> toDoItemsOut,
            TraceWriter log)
        {
            log.Info($"C# Queue trigger function processed {toDoItemsIn?.Length} items");

            foreach (ToDoItem toDoItem in toDoItemsIn)
            {
                log.Info($"Description={toDoItem.Description}");
                await toDoItemsOut.AddAsync(toDoItem);
            }
        }
    }
}
```

Output - attributes

- C#
- C# Script
- JavaScript

In C# class libraries, use the DocumentDB attribute.

The attribute's constructor takes the database name and collection name. For information about those settings and other properties that you can configure, see Output - configuration. Here's a `DocumentDB` attribute example in a method signature:

```
    [FunctionName("QueueToDocDB")]
    public static void Run(
        [QueueTrigger("myqueue-items", Connection = "AzureWebJobsStorage")] string myQueueItem,
        [DocumentDB("ToDoList", "Items", Id = "id", ConnectionStringSetting = "myCosmosDB")] out dynamic
document)
    {
        ...
    }
```

For a complete example, see Output.

Output - configuration

The following table explains the binding configuration properties that you set in the *function.json* file and the `DocumentDB` attribute.

FUNCTION.JSON PROPERTY	ATTRIBUTE PROPERTY	DESCRIPTION
type	n/a	Must be set to `documentdb`.
direction	n/a	Must be set to `out`.
name	n/a	Name of the binding parameter that represents the document in the function.
databaseName	**DatabaseName**	The database containing the collection where the document is created.
collectionName	**CollectionName**	The name of the collection where the document is created.
createIfNotExists	**CreateIfNotExists**	A boolean value to indicate whether the collection is created when it doesn't exist. The default is *false* because new collections are created with reserved throughput, which has cost implications. For more information, see the pricing page.
partitionKey	**PartitionKey**	When `CreateIfNotExists` is true, defines the partition key path for the created collection.
collectionThroughput	**CollectionThroughput**	When `CreateIfNotExists` is true, defines the throughput of the created collection.
connection	**ConnectionStringSetting**	The name of the app setting containing your Azure Cosmos DB connection string.

When you're developing locally, app settings go into the local.settings.json file.

Output - usage

By default, when you write to the output parameter in your function, a document is created in your database. This document has an automatically generated GUID as the document ID. You can specify the document ID of the output document by specifying the `id` property in the JSON object passed to the output parameter.

> **NOTE**
> When you specify the ID of an existing document, it gets overwritten by the new output document.

Exceptions and return codes

BINDING	REFERENCE
CosmosDB	CosmosDB Error Codes

Next steps

- Learn more about serverless database computing with Cosmos DB
- Learn more about Azure functions triggers and bindings

Copy and transform data in Azure Cosmos DB (SQL API) by using Azure Data Factory

11/2/2020 • 10 minutes to read • Edit Online

APPLIES TO: Azure Data Factory Azure Synapse Analytics

This article outlines how to use Copy Activity in Azure Data Factory to copy data from and to Azure Cosmos DB (SQL API), and use Data Flow to transform data in Azure Cosmos DB (SQL API). To learn about Azure Data Factory, read the introductory article.

> **NOTE**
> This connector only support Cosmos DB SQL API. For MongoDB API, refer to connector for Azure Cosmos DB's API for MongoDB. Other API types are not supported now.

Supported capabilities

This Azure Cosmos DB (SQL API) connector is supported for the following activities:

- Copy activity with supported source/sink matrix
- Mapping data flow
- Lookup activity

For Copy activity, this Azure Cosmos DB (SQL API) connector supports:

- Copy data from and to the Azure Cosmos DB SQL API.
- Write to Azure Cosmos DB as **insert** or **upsert**.
- Import and export JSON documents as-is, or copy data from or to a tabular dataset. Examples include a SQL database and a CSV file. To copy documents as-is to or from JSON files or to or from another Azure Cosmos DB collection, see Import and export JSON documents.

Data Factory integrates with the Azure Cosmos DB bulk executor library to provide the best performance when you write to Azure Cosmos DB.

> **TIP**
> The Data Migration video walks you through the steps of copying data from Azure Blob storage to Azure Cosmos DB. The video also describes performance-tuning considerations for ingesting data to Azure Cosmos DB in general.

Get started

To perform the Copy activity with a pipeline, you can use one of the following tools or SDKs:

- The Copy Data tool
- The Azure portal
- The .NET SDK
- The Python SDK
- Azure PowerShell
- The REST API

- The Azure Resource Manager template

The following sections provide details about properties you can use to define Data Factory entities that are specific to Azure Cosmos DB (SQL API).

Linked service properties

The following properties are supported for the Azure Cosmos DB (SQL API) linked service:

PROPERTY	DESCRIPTION	REQUIRED
type	The **type** property must be set to **CosmosDb**.	Yes
connectionString	Specify information that's required to connect to the Azure Cosmos DB database. **Note**: You must specify database information in the connection string as shown in the examples that follow. You can also put account key in Azure Key Vault and pull the `accountKey` configuration out of the connection string. Refer to the following samples and Store credentials in Azure Key Vault article with more details.	Yes
connectVia	The Integration Runtime to use to connect to the data store. You can use the Azure Integration Runtime or a self-hosted integration runtime (if your data store is located in a private network). If this property isn't specified, the default Azure Integration Runtime is used.	No

Example

```
{
    "name": "CosmosDbSQLAPILinkedService",
    "properties": {
        "type": "CosmosDb",
        "typeProperties": {
            "connectionString": "AccountEndpoint=<EndpointUrl>;AccountKey=<AccessKey>;Database=<Database>"
        },
        "connectVia": {
            "referenceName": "<name of Integration Runtime>",
            "type": "IntegrationRuntimeReference"
        }
    }
}
```

Example: store account key in Azure Key Vault

```
{
    "name": "CosmosDbSQLAPILinkedService",
    "properties": {
        "type": "CosmosDb",
        "typeProperties": {
            "connectionString": "AccountEndpoint=<EndpointUrl>;Database=<Database>",
            "accountKey": {
                "type": "AzureKeyVaultSecret",
                "store": {
                    "referenceName": "<Azure Key Vault linked service name>",
                    "type": "LinkedServiceReference"
                },
                "secretName": "<secretName>"
            }
        },
        "connectVia": {
            "referenceName": "<name of Integration Runtime>",
            "type": "IntegrationRuntimeReference"
        }
    }
}
```

Dataset properties

For a full list of sections and properties that are available for defining datasets, see Datasets and linked services.

The following properties are supported for Azure Cosmos DB (SQL API) dataset:

PROPERTY	DESCRIPTION	REQUIRED
type	The **type** property of the dataset must be set to **CosmosDbSqlApiCollection**.	Yes
collectionName	The name of the Azure Cosmos DB document collection.	Yes

If you use "DocumentDbCollection" type dataset, it is still supported as-is for backward compatibility for Copy and Lookup activity, it's not supported for Data Flow. You are suggested to use the new model going forward.

Example

```
{
    "name": "CosmosDbSQLAPIDataset",
    "properties": {
        "type": "CosmosDbSqlApiCollection",
        "linkedServiceName":{
            "referenceName": "<Azure Cosmos DB linked service name>",
            "type": "LinkedServiceReference"
        },
        "schema": [],
        "typeProperties": {
            "collectionName": "<collection name>"
        }
    }
}
```

Copy Activity properties

This section provides a list of properties that the Azure Cosmos DB (SQL API) source and sink support. For a full

list of sections and properties that are available for defining activities, see Pipelines.

Azure Cosmos DB (SQL API) as source

To copy data from Azure Cosmos DB (SQL API), set the **source** type in Copy Activity to **DocumentDbCollectionSource**.

The following properties are supported in the Copy Activity **source** section:

PROPERTY	DESCRIPTION	REQUIRED
type	The **type** property of the copy activity source must be set to **CosmosDbSqlApiSource**.	Yes
query	Specify the Azure Cosmos DB query to read data. Example: `SELECT c.BusinessEntityID, c.Name.First AS FirstName, c.Name.Middle AS MiddleName, c.Name.Last AS LastName, c.Suffix, c.EmailPromotion FROM c WHERE c.ModifiedDate > \"2009-01-01T00:00:00\"`	No If not specified, this SQL statement is executed: `select <columns defined in structure> from mycollection`
preferredRegions	The preferred list of regions to connect to when retrieving data from Cosmos DB.	No
pageSize	The number of documents per page of the query result. Default is "-1" which means uses the service side dynamic page size up to 1000.	No

If you use "DocumentDbCollectionSource" type source, it is still supported as-is for backward compatibility. You are suggested to use the new model going forward which provide richer capabilities to copy data from Cosmos DB.

Example

```
"activities":[
    {
        "name": "CopyFromCosmosDBSQLAPI",
        "type": "Copy",
        "inputs": [
            {
                "referenceName": "<Cosmos DB SQL API input dataset name>",
                "type": "DatasetReference"
            }
        ],
        "outputs": [
            {
                "referenceName": "<output dataset name>",
                "type": "DatasetReference"
            }
        ],
        "typeProperties": {
            "source": {
                "type": "CosmosDbSqlApiSource",
                "query": "SELECT c.BusinessEntityID, c.Name.First AS FirstName, c.Name.Middle AS MiddleName,
c.Name.Last AS LastName, c.Suffix, c.EmailPromotion FROM c WHERE c.ModifiedDate > \"2009-01-01T00:00:00\"",
                "preferredRegions": [
                    "East US"
                ]
            },
            "sink": {
                "type": "<sink type>"
            }
        }
    }
]
```

When copy data from Cosmos DB, unless you want to export JSON documents as-is, the best practice is to specify the mapping in copy activity. Data Factory honors the mapping you specified on the activity - if a row doesn't contain a value for a column, a null value is provided for the column value. If you don't specify a mapping, Data Factory infers the schema by using the first row in the data. If the first row doesn't contain the full schema, some columns will be missing in the result of the activity operation.

Azure Cosmos DB (SQL API) as sink

To copy data to Azure Cosmos DB (SQL API), set the **sink** type in Copy Activity to **DocumentDbCollectionSink**.

The following properties are supported in the Copy Activity **sink** section:

PROPERTY	DESCRIPTION	REQUIRED
type	The **type** property of the Copy Activity sink must be set to **CosmosDbSqlApiSink**.	Yes

PROPERTY	DESCRIPTION	REQUIRED
writeBehavior	Describes how to write data to Azure Cosmos DB. Allowed values: **insert** and **upsert**. The behavior of **upsert** is to replace the document if a document with the same ID already exists; otherwise, insert the document. **Note**: Data Factory automatically generates an ID for a document if an ID isn't specified either in the original document or by column mapping. This means that you must ensure that, for **upsert** to work as expected, your document has an ID.	No (the default is **insert**)
writeBatchSize	Data Factory uses the Azure Cosmos DB bulk executor library to write data to Azure Cosmos DB. The **writeBatchSize** property controls the size of documents that ADF provides to the library. You can try increasing the value for **writeBatchSize** to improve performance and decreasing the value if your document size being large - see below tips.	No (the default is **10,000**)
disableMetricsCollection	Data Factory collects metrics such as Cosmos DB RUs for copy performance optimization and recommendations. If you are concerned with this behavior, specify `true` to turn it off.	No (default is `false`)

> **TIP**
> To import JSON documents as-is, refer to Import or export JSON documents section; to copy from tabular-shaped data, refer to Migrate from relational database to Cosmos DB.

> **TIP**
> Cosmos DB limits single request's size to 2MB. The formula is Request Size = Single Document Size * Write Batch Size. If you hit error saying **"Request size is too large."**, **reduce the** `writeBatchSize` **value** in copy sink configuration.

If you use "DocumentDbCollectionSink" type source, it is still supported as-is for backward compatibility. You are suggested to use the new model going forward which provide richer capabilities to copy data from Cosmos DB.

Example

```
"activities":[
    {
        "name": "CopyToCosmosDBSQLAPI",
        "type": "Copy",
        "inputs": [
            {
                "referenceName": "<input dataset name>",
                "type": "DatasetReference"
            }
        ],
        "outputs": [
            {
                "referenceName": "<Document DB output dataset name>",
                "type": "DatasetReference"
            }
        ],
        "typeProperties": {
            "source": {
                "type": "<source type>"
            },
            "sink": {
                "type": "CosmosDbSqlApiSink",
                "writeBehavior": "upsert"
            }
        }
    }
]
```

Schema mapping

To copy data from Azure Cosmos DB to tabular sink or reversed, refer to schema mapping.

Mapping data flow properties

When transforming data in mapping data flow, you can read and write to collections in Cosmos DB. For more information, see the source transformation and sink transformation in mapping data flows.

Source transformation

Settings specific to Azure Cosmos DB are available in the **Source Options** tab of the source transformation.

Include system columns: If true, `id`, `_ts`, and other system columns will be included in your data flow metadata from CosmosDB. When updating collections, it is important to include this so that you can grab the existing row id.

Page size: The number of documents per page of the query result. Default is "-1" which uses the service dynamic page up to 1000.

Throughput: Set an optional value for the number of RUs you'd like to apply to your CosmosDB collection for each execution of this data flow during the read operation. Minimum is 400.

Preferred regions: Choose the preferred read regions for this process.

JSON Settings

Single document: Select this option if ADF is to treat the entire file as a single JSON doc.

Unquoted column names: Select this option if column names in the JSON as not quoted.

Has comments: Use this selection if your JSON documents have comments in the data.

Single quoted: This should be selected if the columns and values in your document are quoted with single quotes.

Backslash escaped: If using backslashes to escape characters in your JSON, choose this option.

Sink transformation

Settings specific to Azure Cosmos DB are available in the **Settings** tab of the sink transformation.

Update method: Determines what operations are allowed on your database destination. The default is to only allow inserts. To update, upsert, or delete rows, an alter-row transformation is required to tag rows for those actions. For updates, upserts and deletes, a key column or columns must be set to determine which row to alter.

Collection action: Determines whether to recreate the destination collection prior to writing.

- None: No action will be done to the collection.
- Recreate: The collection will get dropped and recreated

Batch size: Controls how many rows are being written in each bucket. Larger batch sizes improve compression and memory optimization, but risk out of memory exceptions when caching data.

Partition Key: Enter a string that represents the partition key for your collection. Example: `/movies/title`

Throughput: Set an optional value for the number of RUs you'd like to apply to your CosmosDB collection for each execution of this data flow. Minimum is 400.

Write throughput budget: An integer that represents the number of RUs you want to allocate to the bulk ingestion Spark job. This number is out of the total throughput allocated to the collection.

Lookup activity properties

To learn details about the properties, check Lookup activity.

Import and export JSON documents

You can use this Azure Cosmos DB (SQL API) connector to easily:

- Copy documents between two Azure Cosmos DB collections as-is.
- Import JSON documents from various sources to Azure Cosmos DB, including from Azure Blob storage, Azure Data Lake Store, and other file-based stores that Azure Data Factory supports.
- Export JSON documents from an Azure Cosmos DB collection to various file-based stores.

To achieve schema-agnostic copy:

- When you use the Copy Data tool, select the **Export as-is to JSON files or Cosmos DB collection** option.
- When you use activity authoring, choose JSON format with the corresponding file store for source or sink.

Migrate from relational database to Cosmos DB

When migrating from a relational database e.g. SQL Server to Azure Cosmos DB, copy activity can easily map tabular data from source to flatten JSON documents in Cosmos DB. In some cases, you may want to redesign the data model to optimize it for the NoSQL use-cases according to Data modeling in Azure Cosmos DB, for example, to denormalize the data by embedding all of the related sub-items within one JSON document. For such case, refer to this article with a walkthrough on how to achieve it using Azure Data Factory copy activity.

Next steps

For a list of data stores that Copy Activity supports as sources and sinks in Azure Data Factory, see supported data stores.

Azure Stream Analytics output to Azure Cosmos DB

11/2/2020 • 8 minutes to read • Edit Online

Azure Stream Analytics can target Azure Cosmos DB for JSON output, enabling data archiving and low-latency queries on unstructured JSON data. This document covers some best practices for implementing this configuration. We recommend that you set your job to compatability level 1.2 when using Azure Cosmos DB as output.

If you're unfamiliar with Azure Cosmos DB, see the Azure Cosmos DB documentation to get started.

> **NOTE**
> At this time, Stream Analytics supports connection to Azure Cosmos DB only through the *SQL API*. Other Azure Cosmos DB APIs are not yet supported. If you point Stream Analytics to Azure Cosmos DB accounts created with other APIs, the data might not be properly stored.

Basics of Azure Cosmos DB as an output target

The Azure Cosmos DB output in Stream Analytics enables writing your stream processing results as JSON output into your Azure Cosmos DB containers.

Stream Analytics doesn't create containers in your database. Instead, it requires you to create them up front. You can then control the billing costs of Azure Cosmos DB containers. You can also tune the performance, consistency, and capacity of your containers directly by using the Azure Cosmos DB APIs.

> **NOTE**
> You must add 0.0.0.0 to the list of allowed IPs from your Azure Cosmos DB firewall.

The following sections detail some of the container options for Azure Cosmos DB.

Tuning consistency, availability, and latency

To match your application requirements, Azure Cosmos DB allows you to fine-tune the database and containers and make trade-offs between consistency, availability, latency, and throughput.

Depending on what levels of read consistency your scenario needs against read and write latency, you can choose a consistency level on your database account. You can improve throughput by scaling up Request Units (RUs) on the container.

Also by default, Azure Cosmos DB enables synchronous indexing on each CRUD operation to your container. This is another useful option to control write/read performance in Azure Cosmos DB.

For more information, review the Change your database and query consistency levels article.

Upserts from Stream Analytics

Stream Analytics integration with Azure Cosmos DB allows you to insert or update records in your container based on a given **Document ID** column. This is also called an *upsert*.

Stream Analytics uses an optimistic upsert approach. Updates happen only when an insert fails with a document ID conflict.

With compatibility level 1.0, Stream Analytics performs this update as a PATCH operation, so it enables partial updates to the document. Stream Analytics adds new properties or replaces an existing property incrementally. However, changes in the values of array properties in your JSON document result in overwriting the entire array. That is, the array isn't merged.

With 1.2, upsert behavior is modified to insert or replace the document. The later section about compatibility level 1.2 further describes this behavior.

If the incoming JSON document has an existing ID field, that field is automatically used as the **Document ID** column in Azure Cosmos DB. Any subsequent writes are handled as such, leading to one of these situations:

- Unique IDs lead to insert.
- Duplicate IDs and **Document ID** set to **ID** lead to upsert.
- Duplicate IDs and **Document ID** not set lead to error, after the first document.

If you want to save *all* documents, including the ones that have a duplicate ID, rename the ID field in your query (by using the **AS** keyword). Let Azure Cosmos DB create the ID field or replace the ID with another column's value (by using the **AS** keyword or by using the **Document ID** setting).

Data partitioning in Azure Cosmos DB

Azure Cosmos DB automatically scales partitions based on your workload. So we recommend unlimited containers as the approach for partitioning your data. When Stream Analytics writes to unlimited containers, it uses as many parallel writers as the previous query step or input partitioning scheme.

> **NOTE**
> Azure Stream Analytics supports only unlimited containers with partition keys at the top level. For example, `/region` is supported. Nested partition keys (for example, `/region/name`) are not supported.

Depending on your choice of partition key, you might receive this *warning*:

```
CosmosDB Output contains multiple rows and just one row per partition key. If the output latency is higher than
expected, consider choosing a partition key that contains at least several hundred records per partition key.
```

It's important to choose a partition key property that has a number of distinct values, and that lets you distribute your workload evenly across these values. As a natural artifact of partitioning, requests that involve the same partition key are limited by the maximum throughput of a single partition.

The storage size for documents that belong to the same partition key value is limited to 20 GB (the physical partition size limit is 50 GB). An ideal partition key is one that appears frequently as a filter in your queries and has sufficient cardinality to ensure that your solution is scalable.

Partition keys used for Stream Analytics queries and Cosmos DB don't need to be identical. Fully parallel topologies recommend using *Input Partition key*, `PartitionId`, as the Stream Analytics query's partition key but that may not be the recommended choice for a Cosmos DB container's partition key.

A partition key is also the boundary for transactions in stored procedures and triggers for Azure Cosmos DB. You should choose the partition key so that documents that occur together in transactions share the same partition key value. The article Partitioning in Azure Cosmos DB gives more details on choosing a partition key.

For fixed Azure Cosmos DB containers, Stream Analytics allows no way to scale up or out after they're full. They have an upper limit of 10 GB and 10,000 RU/s of throughput. To migrate the data from a fixed container to an unlimited container (for example, one with at least 1,000 RU/s and a partition key), use the data migration tool or the change feed library.

The ability to write to multiple fixed containers is being deprecated. We don't recommend it for scaling out your

Stream Analytics job.

Improved throughput with compatibility level 1.2

With compatibility level 1.2, Stream Analytics supports native integration to bulk write into Azure Cosmos DB. This integration enables writing effectively to Azure Cosmos DB while maximizing throughput and efficiently handling throttling requests.

The improved writing mechanism is available under a new compatibility level because of a difference in upsert behavior. With levels before 1.2, the upsert behavior is to insert or merge the document. With 1.2, upsert behavior is modified to insert or replace the document.

With levels before 1.2, Stream Analytics uses a custom stored procedure to bulk upsert documents per partition key into Azure Cosmos DB. There, a batch is written as a transaction. Even when a single record hits a transient error (throttling), the whole batch has to be retried. This makes scenarios with even reasonable throttling relatively slow.

The following example shows two identical Stream Analytics jobs reading from the same Azure Event Hubs input. Both Stream Analytics jobs are fully partitioned with a passthrough query and write to identical Azure Cosmos DB containers. Metrics on the left are from the job configured with compatibility level 1.0. Metrics on the right are configured with 1.2. An Azure Cosmos DB container's partition key is a unique GUID that comes from the input event.

The incoming event rate in Event Hubs is two times higher than Azure Cosmos DB containers (20,000 RUs) are configured to take in, so throttling is expected in Azure Cosmos DB. However, the job with 1.2 is consistently writing at a higher throughput (output events per minute) and with a lower average SU% utilization. In your environment, this difference will depend on few more factors. These factors include choice of event format, input event/message size, partition keys, and query.

With 1.2, Stream Analytics is more intelligent in utilizing 100 percent of the available throughput in Azure Cosmos DB with very few resubmissions from throttling or rate limiting. This provides a better experience for other workloads like queries running on the container at the same time. If you want to see how Stream Analytics scales out with Azure Cosmos DB as a sink for 1,000 to 10,000 messages per second, try this Azure sample project.

Throughput of Azure Cosmos DB output is identical with 1.0 and 1.1. We *strongly recommend* that you use compatibility level 1.2 in Stream Analytics with Azure Cosmos DB.

Azure Cosmos DB settings for JSON output

Using Azure Cosmos DB as an output in Stream Analytics generates the following prompt for information.

FIELD	DESCRIPTION
Output alias	An alias to refer to this output in your Stream Analytics query.

FIELD	DESCRIPTION
Subscription	The Azure subscription.
Account ID	The name or endpoint URI of the Azure Cosmos DB account.
Account key	The shared access key for the Azure Cosmos DB account.
Database	The Azure Cosmos DB database name.
Container name	The container name, such as `MyContainer`. One container named `MyContainer` must exist.
Document ID	Optional. The column name in output events used as the unique key on which insert or update operations must be based. If you leave it empty, all events will be inserted, with no update option.

After you configure the Azure Cosmos DB output, you can use it in the query as the target of an INTO statement. When you're using an Azure Cosmos DB output that way, a partition key needs to be set explicitly.

The output record must contain a case-sensitive column named after the partition key in Azure Cosmos DB. To achieve greater parallelization, the statement might require a PARTITION BY clause that uses the same column.

Here's a sample query:

```
SELECT TollBoothId, PartitionId
INTO CosmosDBOutput
FROM Input1 PARTITION BY PartitionId
```

Error handling and retries

If a transient failure, service unavailability, or throttling happens while Stream Analytics is sending events to Azure Cosmos DB, Stream Analytics retries indefinitely to finish the operation successfully. But it doesn't attempt retries for the following failures:

- Unauthorized (HTTP error code 401)
- NotFound (HTTP error code 404)
- Forbidden (HTTP error code 403)
- BadRequest (HTTP error code 400)

Common issues

1. A unique index constraint is added to the collection and the output data from Stream Analytics violates this constraint. Ensure the output data from Stream Analytics doesn't violate unique constraints or remove constraints. For more information, see Unique key constraints in Azure Cosmos DB.
2. The `PartitionKey` column does not exists.
3. The `Id` column does not exist.

Next steps

- Understand outputs from Azure Stream Analytics

- Azure Stream Analytics output to Azure SQL Database
- Azure Stream Analytics custom blob output partitioning

Connect to Azure Cosmos DB using BI analytics tools with the ODBC driver

12/4/2020 • 14 minutes to read • Edit Online

APPLIES TO: SQL API

The Azure Cosmos DB ODBC driver enables you to connect to Azure Cosmos DB using BI analytics tools such as SQL Server Integration Services, Power BI Desktop, and Tableau so that you can analyze and create visualizations of your Azure Cosmos DB data in those solutions.

The Azure Cosmos DB ODBC driver is ODBC 3.8 compliant and supports ANSI SQL-92 syntax. The driver offers rich features to help you renormalize data in Azure Cosmos DB. Using the driver, you can represent data in Azure Cosmos DB as tables and views. The driver enables you to perform SQL operations against the tables and views including group by queries, inserts, updates, and deletes.

> **NOTE**
>
> Connecting to Azure Cosmos DB with the ODBC driver is currently supported for Azure Cosmos DB SQL API accounts only.

Why do I need to normalize my data?

Azure Cosmos DB is a schemaless database, which enables rapid application development and the ability to iterate on data models without being confined to a strict schema. A single Azure Cosmos database can contain JSON documents of various structures. This is great for rapid application development, but when you want to analyze and create reports of your data using data analytics and BI tools, the data often needs to be flattened and adhere to a specific schema.

This is where the ODBC driver comes in. By using the ODBC driver, you can now renormalize data in Azure Cosmos DB into tables and views that fit your data analytics and reporting needs. The renormalized schemas have no impact on the underlying data and do not confine developers to adhere to them. Rather, they enable you to leverage ODBC-compliant tools to access the data. So, now your Azure Cosmos database will not only be a favorite for your development team, but your data analysts will love it too.

Let's get started with the ODBC driver.

Step 1: Install the Azure Cosmos DB ODBC driver

1. Download the drivers for your environment:

INSTALLER	SUPPORTED OPERATING SYSTEMS
Microsoft Azure Cosmos DB ODBC 64-bit.msi for 64-bit Windows	64-bit versions of Windows 8.1 or later, Windows 8, Windows 7, Windows Server 2012 R2, Windows Server 2012, and Windows Server 2008 R2.
Microsoft Azure Cosmos DB ODBC 32x64-bit.msi for 32-bit on 64-bit Windows	64-bit versions of Windows 8.1 or later, Windows 8, Windows 7, Windows XP, Windows Vista, Windows Server 2012 R2, Windows Server 2012, Windows Server 2008 R2, and Windows Server 2003.

INSTALLER	SUPPORTED OPERATING SYSTEMS
Microsoft Azure Cosmos DB ODBC 32-bit.msi for 32-bit Windows	32-bit versions of Windows 8.1 or later, Windows 8, Windows 7, Windows XP, and Windows Vista.

Run the msi file locally, which starts the **Microsoft Azure Cosmos DB ODBC Driver Installation Wizard**.

2. Complete the installation wizard using the default input to install the ODBC driver.

3. Open the **ODBC Data source Administrator** app on your computer. You can do this by typing **ODBC Data sources** in the Windows search box. You can confirm the driver was installed by clicking the **Drivers** tab and ensuring **Microsoft Azure Cosmos DB ODBC Driver** is listed.

Step 2: Connect to your Azure Cosmos database

1. After Installing the Azure Cosmos DB ODBC driver, in the **ODBC Data Source Administrator** window, click **Add**. You can create a User or System DSN. In this example, you are creating a User DSN.

2. In the **Create New Data Source** window, select **Microsoft Azure Cosmos DB ODBC Driver**, and then click **Finish**.

3. In the **Azure Cosmos DB ODBC Driver SDN Setup** window, fill in the following information:

- **Data Source Name**: Your own friendly name for the ODBC DSN. This name is unique to your Azure Cosmos DB account, so name it appropriately if you have multiple accounts.
- **Description**: A brief description of the data source.
- **Host**: URI for your Azure Cosmos DB account. You can retrieve this from the Azure Cosmos DB Keys page in the Azure portal, as shown in the following screenshot.
- **Access Key**: The primary or secondary, read-write or read-only key from the Azure Cosmos DB Keys page in the Azure portal as shown in the following screenshot. We recommend you use the read-only key if the DSN is used for read-only data processing and reporting.

- **Encrypt Access Key for**: Select the best choice based on the users of this machine.

4. Click the **Test** button to make sure you can connect to your Azure Cosmos DB account.

5. Click **Advanced Options** and set the following values:

- **REST API Version**: Select the REST API version for your operations. The default 2015-12-16. If you have containers with large partition keys and require REST API version 2018-12-31:
  - Type in **2018-12-31** for REST API version
  - In the **Start** menu, type "regedit" to find and open the **Registry Editor** application.
  - In Registry Editor, navigate to the path: **Computer\HKEY_LOCAL_MACHINE\SOFTWARE\ODBC\ODBC.INI**
  - Create a new subkey with the same name as your DSN, e.g. "Contoso Account ODBC DSN".
  - Navigate to the "Contoso Account ODBC DSN" subkey.
  - Right-click to add a new **String** value:
    - Value Name: **IgnoreSessionToken**
    - Value data: **1**

- **Query Consistency**: Select the consistency level for your operations. The default is Session.
- **Number of Retries**: Enter the number of times to retry an operation if the initial request does not complete due to service rate limiting.
- **Schema File**: You have a number of options here.
  - By default, leaving this entry as is (blank), the driver scans the first page of data for all containers to determine the schema of each container. This is known as Container Mapping. Without a schema file defined, the driver has to perform the scan for each driver session and could result in a higher startup time of an application using the DSN. We recommend that you always associate a schema file for a DSN.
  - If you already have a schema file (possibly one that you created using the Schema Editor), you can click **Browse**, navigate to your file, click **Save**, and then click **OK**.
  - If you want to create a new schema, click **OK**, and then click **Schema Editor** in the main window.

Then proceed to the Schema Editor information. After creating the new schema file, remember to go back to the **Advanced Options** window to include the newly created schema file.

6. Once you complete and close the **Azure Cosmos DB ODBC Driver DSN Setup** window, the new User DSN is added to the User DSN tab.

Step 3: Create a schema definition using the container mapping method

There are two types of sampling methods that you can use: **container mapping** or **table-delimiters**. A sampling session can utilize both sampling methods, but each container can only use a specific sampling method. The steps below create a schema for the data in one or more containers using the container mapping method. This sampling method retrieves the data in the page of a container to determine the structure of the data. It transposes a container to a table on the ODBC side. This sampling method is efficient and fast when the data in a container is homogenous. If a container contains heterogeneous type of data, we recommend you use the table-delimiters mapping method as it provides a more robust sampling method to determine the data structures in the container.

1. After completing steps 1-4 in Connect to your Azure Cosmos database, click **Schema Editor** in the **Azure Cosmos DB ODBC Driver DSN Setup** window.

2. In the **Schema Editor** window, click **Create New**. The **Generate Schema** window displays all the containers in the Azure Cosmos DB account.

3. Select one or more containers to sample, and then click **Sample**.

4. In the **Design View** tab, the database, schema, and table are represented. In the table view, the scan displays the set of properties associated with the column names (SQL Name, Source Name, etc.). For each column, you can modify the column SQL name, the SQL type, SQL length (if applicable), Scale (if applicable), Precision (if applicable) and Nullable.

   - You can set **Hide Column** to **true** if you want to exclude that column from query results. Columns marked Hide Column = true are not returned for selection and projection, although they are still part of the schema. For example, you can hide all of the Azure Cosmos DB system required properties starting with "_".
   - The **id** column is the only field that cannot be hidden as it is used as the primary key in the normalized schema.

5. Once you have finished defining the schema, click **File | Save**, navigate to the directory to save the schema, and then click **Save**.

6. To use this schema with a DSN, open the **Azure Cosmos DB ODBC Driver DSN Setup window** (via the ODBC Data Source Administrator), click **Advanced Options**, and then in the **Schema File** box, navigate to the saved schema. Saving a schema file to an existing DSN modifies the DSN connection to scope to the data and structure defined by schema.

Step 4: Create a schema definition using the table-delimiters mapping method

There are two types of sampling methods that you can use: **container mapping** or **table-delimiters**. A sampling session can utilize both sampling methods, but each container can only use a specific sampling method.

The following steps create a schema for the data in one or more containers using the **table-delimiters** mapping method. We recommend that you use this sampling method when your containers contain heterogeneous type of data. You can use this method to scope the sampling to a set of attributes and its corresponding values. For example, if a document contains a "Type" property, you can scope the sampling to the values of this property. The end result of the sampling would be a set of tables for each of the values for Type you have specified. For example, Type = Car will produce a Car table while Type = Plane would produce a Plane table.

1. After completing steps 1-4 in Connect to your Azure Cosmos database, click **Schema Editor** in the Azure Cosmos DB ODBC Driver DSN Setup window.

2. In the **Schema Editor** window, click **Create New**. The **Generate Schema** window displays all the containers in the Azure Cosmos DB account.

3. Select a container on the **Sample View** tab, in the **Mapping Definition** column for the container, click **Edit**. Then in the **Mapping Definition** window, select **Table Delimiters** method. Then do the following:

   a. In the **Attributes** box, type the name of a delimiter property. This is a property in your document that you want to scope the sampling to, for instance, City and press enter.

   b. If you only want to scope the sampling to certain values for the attribute you entered above, select the attribute in the selection box, enter a value in the **Value** box (e.g. Seattle), and press enter. You can continue to add multiple values for attributes. Just ensure that the correct attribute is selected when you're entering values.

   For example, if you include an **Attributes** value of City, and you want to limit your table to only include rows with a city value of New York and Dubai, you would enter City in the Attributes box, and New York and then Dubai in the **Values** box.

4. Click **OK**.

5. After completing the mapping definitions for the containers you want to sample, in the **Schema Editor** window, click **Sample**. For each column, you can modify the column SQL name, the SQL type, SQL length (if applicable), Scale (if applicable), Precision (if applicable) and Nullable.
   - You can set **Hide Column** to **true** if you want to exclude that column from query results. Columns marked Hide Column = true are not returned for selection and projection, although they are still part of the schema. For example, you can hide all the Azure Cosmos DB system required properties starting with `_`.
   - The **id** column is the only field that cannot be hidden as it is used as the primary key in the normalized schema.
6. Once you have finished defining the schema, click **File | Save**, navigate to the directory to save the schema, and then click **Save**.
7. Back in the **Azure Cosmos DB ODBC Driver DSN Setup** window, click **Advanced Options**. Then, in the **Schema File** box, navigate to the saved schema file and click **OK**. Click **OK** again to save the DSN. This saves the schema you created to the DSN.

(Optional) Set up linked server connection

You can query Azure Cosmos DB from SQL Server Management Studio (SSMS) by setting up a linked server connection.

1. Create a system data source as described in Step 2, named for example `SDS Name`.
2. Install SQL Server Management Studio and connect to the server.
3. In the SSMS query editor, create a linked server object `DEMOCOSMOS` for the data source with the following commands. Replace `DEMOCOSMOS` with the name for your linked server, and `SDS Name` with the name of your system data source.

```
USE [master]
GO

EXEC master.dbo.sp_addlinkedserver @server = N'DEMOCOSMOS', @srvproduct=N'', @provider=N'MSDASQL',
@datasrc=N'SDS Name'

EXEC master.dbo.sp_addlinkedsrvlogin @rmtsrvname=N'DEMOCOSMOS', @useself=N'False', @locallogin=NULL,
@rmtuser=NULL, @rmtpassword=NULL

GO
```

To see the new linked server name, refresh the Linked Servers list.

Query linked database

To query the linked database, enter an SSMS query. In this example, the query selects from the table in the container named `customers`:

```
SELECT * FROM OPENQUERY(DEMOCOSMOS, 'SELECT *  FROM [customers].[customers]')
```

Execute the query. The result should be similar to this:

```
attachments/  1507476156    521 Bassett Avenue, Wikieup, Missouri, 5422   "2602bc56-0000-0000-0000-
59da42bc0000"   2015-02-06T05:32:32 +05:00 f1ca3044f17149f3bc61f7b9c78a26df
attachments/  1507476156    167 Nassau Street, Tuskahoma, Illinois, 5998   "2602bd56-0000-0000-0000-
59da42bc0000"   2015-06-16T08:54:17 +04:00 f75f949ea8de466a9ef2bdb7ce065ac8
attachments/  1507476156    885 Strong Place, Cassel, Montana, 2069       "2602be56-0000-0000-0000-
59da42bc0000"   2015-03-20T07:21:47 +04:00 ef0365fb40c04bb6a3ffc4bc77c905fd
attachments/  1507476156    515 Barwell Terrace, Defiance, Tennessee, 6439     "2602c056-0000-0000-0000-
59da42bc0000"   2014-10-16T06:49:04 +04:00      e913fe543490432f871bc42019663518
attachments/  1507476156    570 Ruby Street, Spokane, Idaho, 9025       "2602c156-0000-0000-0000-59da42bc0000"
2014-10-30T05:49:33 +04:00 e53072057d314bc9b36c89a8350048f3
```

> **NOTE**
>
> The linked Cosmos DB server does not support four-part naming. An error is returned similar to the following message:

```
Msg 7312, Level 16, State 1, Line 44

Invalid use of schema or catalog for OLE DB provider "MSDASQL" for linked server "DEMOCOSMOS". A four-part
name was supplied, but the provider does not expose the necessary interfaces to use a catalog or schema.
```

(Optional) Creating views

You can define and create views as part of the sampling process. These views are equivalent to SQL views. They are read-only and are scope the selections and projections of the Azure Cosmos DB SQL query defined.

To create a view for your data, in the **Schema Editor** window, in the **View Definitions** column, click **Add** on the row of the container to sample.

Then in the **View Definitions** window, do the following:

1. Click **New**, enter a name for the view, for example, EmployeesfromSeattleView and then click **OK**.

2. In the **Edit view** window, enter an Azure Cosmos DB query. This must be an Azure Cosmos DB SQL query, for example `SELECT c.City, c.EmployeeName, c.Level, c.Age, c.Manager FROM c WHERE c.City = "Seattle"`, and then click **OK**.

You can create a many views as you like. Once you are done defining the views, you can then sample the data.

Step 5: View your data in BI tools such as Power BI Desktop

You can use your new DSN to connect to Azure Cosmos DB with any ODBC-compliant tools - this step simply shows you how to connect to Power BI Desktop and create a Power BI visualization.

1. Open Power BI Desktop.
2. Click **Get Data**.

3. In the **Get Data** window, click **Other** | **ODBC** | **Connect**.

4. In the **From ODBC** window, select the data source name you created, and then click **OK**. You can leave the **Advanced Options** entries blank.

5. In the **Access a data source using an ODBC driver** window, select **Default or Custom** and then click **Connect**. You do not need to include the **Credential connection string properties**.

6. In the **Navigator** window, in the left pane, expand the database, the schema, and then select the table. The results pane includes the data using the schema you created.

7. To visualize the data in Power BI desktop, check the box in front of the table name, and then click **Load**.

8. In Power BI Desktop, on the far left, select the Data tab to confirm your data was imported.

9. You can now create visuals using Power BI by clicking on the Report tab , clicking **New Visual**, and then customizing your tile. For more information about creating visualizations in Power BI Desktop, see Visualization types in Power BI.

Troubleshooting

If you receive the following error, ensure the **Host** and **Access Key** values you copied the Azure portal in Step 2 are correct and then retry. Use the copy buttons to the right of the **Host** and **Access Key** values in the Azure portal to copy the values error free.

```
[HY000]: [Microsoft][Azure Cosmos DB] (401) HTTP 401 Authentication Error:
{"code":"Unauthorized","message":"The input authorization token can't serve the request. Please check that the
expected payload is built as per the protocol, and check the key being used. Server used the following payload
to sign: 'get\ndbs\n\nfri, 20 jan 2017 03:43:55 gmt\n\n'\r\nActivityId: 9acb3c0d-cb31-4b78-ac0a-413c8d33e373"}
```

Next steps

To learn more about Azure Cosmos DB, see Welcome to Azure Cosmos DB.

Azure Cosmos DB .NET SDK v3 for SQL API: Download and release notes

11/2/2020 • 17 minutes to read • Edit Online

APPLIES TO: SQL API

SDK download	NuGet
API documentation	.NET API reference documentation
Samples	.NET code samples
Get started	Get started with the Azure Cosmos DB .NET SDK
Web app tutorial	Web application development with Azure Cosmos DB
Entity Framework Core tutorial	Entity Framework Core with Azure Cosmos DB Provider
Current supported framework	Microsoft .NET Standard 2.0

Preview features are treated as a separate branch and will not be included in the official release until the feature is ready. Each preview release lists all the additional features that are enabled.

The format is based on Keep a Changelog, and this project adheres to Semantic Versioning.

3.15.0 - 2020-11-17

Added

- #1926 Query: Adds multiple arguments in IN clause support to c# query parser when service interop is not available.
- #1933 ChangeFeed: Adds adoption of pagination library
- #1943 Performance: Adds query optimization by LazyCosmosElement Cache Improvements
- #1944 Performance: Adds direct version to get response header improvement
- #1947 ReadFeed: Adds pagination library adoption
- #1949 Performance: Adds optimized request headers
- #1974 Performance: Adds Bulk optimization by reducing lock contention in TimerWheel
- #1977 Performance diagnostics: Adds static timer and caches handler name

Fixed

- #1930 Change Feed: Fixes estimator diagnostics
- #1939 LINQ: Fixes ArgumentNullException with StringComparison sensitive case (Thanks to ylabade)
- #1940 LINQ: Fixes CancellationToken bug in CosmosLinqQuery.AggregateResultAsync (Thanks to ylabade)
- #1960 CosmosClientOptions and ClientBuilder: Fixes ArgumentException when setting null value on HttpClientFactory or WebProxy
- #1961 RequestOption.Properties: Fixes RequestOption.Properties for CreateContainerIfNotExistsAsync
- #1967 Query: Fixes CancellationToken logic in pagination library
- #1988 Query: Fixes split proofing logic for queries with logical partition key

- #1999 Performance: Fixes exception serialization when tracing is not enabled
- #2004 Query: Fixes SplitHandling bug caused by caches not getting refreshed. Introduced in 3.14.0 PR #1812

3.15.2-preview - 2020-11-17

Fixed

- #2004 Query: Fixes SplitHandling bug caused by caches not getting refreshed. Introduced in 3.14.0 PR #1812

Unlisted see #2004 - 3.15.1-preview - 2020-11-05

Fixed

- #1972 Private preview Azure Active Directory: Fixes TokenCredentialCache timeout logic and ports tests from master
- #1984 Private preview Azure Active Directory: Fixes issue with using wrong scope value

Unlisted see #2004 - 3.15.0-preview - 2020-10-21

Added

- #1944 Performance: Adds direct version to get response header improvement
- #1933 Change Feed: Adds new continuation token format which can be migrated via new EmitOldContinuationToken.
- #1933 Change Feed: Adds the ability to retry on 304s and no longer modifies HasMoreResults
- #1926 Query: Adds multiple arguments in IN clause support to c# query parser when service interop is not available.
- #1798 Private Preview Azure Active Directory: Adds Azure Active Directory support to the SDK

Fixed

- #1933 Change Feed: Fixes StartFrom bug where the value was not honored

Unlisted see #2004 - 3.14.0-preview - 2020-10-09

Added

- #1830 Change Feed Estimator: Adds support for detailed estimation per lease

Unlisted see #2004 - 3.14.0 - 2020-10-09

Added

- #1876 Performance: Adds session token optimization
- #1879 Performance: Adds AuthorizationHelper improvements
- #1882 Performance: Adds SessionContainer optimizations and style fixes
- #1812 Query: Adds adoption of pagination library
- #1920 Query: Adds RegexMatch system function support

Fixed

- #1875 HttpClient: Fixes HttpResponseMessage.RequestMessage is null in WASM
- #1886 Change Feed Processor: Fixes failures during initialization
- #1892 GatewayAddressCache: Fixes high CPU from HashSet usage on Address refresh path
- #1909 Authorization: Fixes DocumentClientException being thrown on write operations

3.13.0 - 2020-09-21

Added

- #1743 Query Performance: Adds skipping getting query plan for non-aggregate single partition queries on non-Windows x64 systems when FeedOptions.PartitionKey is set
- #1768 Performance: Adds SessionToken optimization to reduce header size by removing session token for CRUD on stored procedure, triggers, and UDFs
- #1781 Performance: Adds headers optimization which can reduce response allocation by 10 KB per a

request.

- #1825 RequestOptions.Properties: Adds the ability for applications to specify request context
- #1835 Performance: Add HttpClient optimization to avoid double buffering gateway responses
- #1837 Query SystemFunctions : Adds DateTime System Functions
- #1842 Query Performance: Adds Singleton QueryPartitionProvider. Helps when Container is getting recreated.
- #1857 Performance: Adds finalizer optimizations in a few places (Thanks to pentp)
- #1843 Performance: Adds Transport serialization, SessionTokenMismatchRetryPolicy, and store response dictionary optimizations

Fixed

- #1757 Batch API: Fixes the size limit to reduce timeouts
- #1758 Connectivity: Fixes address resolution calls when using EnableTcpConnectionEndpointRediscovery
- #1788 Transient HTTP exceptions: Adds retry logic to all http requests
- #1863 HttpClientHandler: Fixes HttpClientHandler PlatformNotSupportedException

3.13.0-preview - 2020-08-12

Added

- #1725 ChangeFeed : Adds ChangeFeedStartFrom to support StartTimes x FeedRanges. WARNING: This is breaking change for preview SDK
- #1764 Performance: Adds compiler optimize flag
- #1768 SessionToken: Adds optimization to reduce header size by removing session token for CRUD on stored procedure, triggers, and UDFs

Fixed

- #1757 Batch API: Fixes the size limit to reduce timeouts
- #1758 Connectivity: Fixes address resolution calls when using EnableTcpConnectionEndpointRediscovery

3.12.0 - 2020-08-06

Added

- #1548 Transport: Adds an optimization to unify HttpClient usage across Gateway classes
- #1569 Batch API: Adds support of request options for transactional batch
- #1693 Performance: Reduces lock contention on GlobalAddress Resolver
- #1712 Performance: Adds optimization to reduce AuthorizationHelper memory allocations
- #1715 Availability: Adds cross-region retry mechanism on transient connectivity issues
- #1721 LINQ : Adds support for case-insensitive searches (Thanks to jeffpardy)
- #1733 Change Feed Processor: Adds backward compatibility of lease store

Fixed

- #1720 Gateway Trace: Fixes a bug where the ActivityId is being set to Guid.Empty
- #1728 Diagnostics: Fixes ActivityScope by moving it to operation level
- #1740 Connection limits: Fixes .NET core to honor gateway connection limit
- #1744 Transport: Fixes use of PortReuseMode and other Direct configuration settings

3.11.1-preview - 2020-10-01

- #1892 Performance: Fixes High CPU caused by EnableTcpConnectionEndpointRediscovery by reducing HashSet lock contention

3.11.0 - 2020-07-07

3.11.0-preview - 2020-07-07

Added

- #1587 & 1643 & 1667 Diagnostics: Adds synchronization context tracing to all request

- #1617 Performance: Fixes Object Model hierarchy to use strings for relative paths instead of URI
- #1639 CosmosClient: Adds argument check for empty key to prevent ambiguous 401 not authorized exception
- #1640 Bulk: Adds TimerWheel to Bulk to improve latency
- #1678 Autoscale: Adds to container builder

Fixed

- #1638 Documentation : Fixes all examples to add using statement to FeedIterator
- #1666 CosmosOperationCanceledException: Fixes handler to catch all operation cancelled exceptions
- #1682 Performance: Fixes high CPU consumption caused by EnableTcpConnectionEndpointRediscovery

3.10.1 - 2020-06-18

- #1637 TransportHandler : Removes stack trace print. Introduced in 3.10.0 PR 1587

3.10.0 - 2020-06-18

Added

- #1613 Query FeedIterator: Adds IDisposable to fix memory leak. WARNING: This will require changes to fix static anlysis tools checking for dispose.
- #1550 CosmosOperationCanceledException: This enables users to access the diagnsotics when an operation is canceled via the cancellation token. The new type extends OperationCanceledException so it does not break current exception handling and includes the CosmosDiagnostic in the ToString().
- #1578 Query: Adds memory optimization to prevent coping the buffer
- #1578 Query: Adds support for ignore case for Contains and StartsWith functions.
- #1602 Diagnostics: Adds CPU usage to all operations
- #1603 Documentation: Adds new exception handling documentation

Fixed

- #1530 ContainerDefinition : Fixes WithDefaultTimeToLive argument spelling (Thanks to tony-xia)
- #1547 & #1582 Query and Readfeed: Fix exceptions caused by not properly handling splits
- #1578 ApplicationRegion: Fixes ApplicationRegion to ensure the correct order is being used for failover scenarios
- #1585 Query : Fixes Multi- ORDER BY continuation token support with QueryExecutionInfo response headers

3.9.1 - 2020-05-19

3.9.1-preview - 2020-05-19

Fixed

- #1539 CosmosException and Diagnostics: Fixes ToString() to not grow exponentially with retries. Introduced in 3.7.0 in PR #1189.

3.9.0 - 2020-05-18

Added

- #1356 & #1407 & #1428 & #1407 Adds autoscale support.
- #1398 Diagnostics: Adds CPU monitoring for .NET Core.
- #1441 Transport: Adds `HttpClientFactory` support on `CosmosClientOptions`.
- #1457 Container: Adds database reference to the container.
- #1455 Serializer: Adds SDK serializer to `Client.ClientOptions.Serializer`.
- #1397 CosmosClientBuilder: Adds preferred regions and `WithConnectionModeDirect()`.
- #1439 No content on Response: Adds the ability to have operations return no content from Azure Cosmos DB.
- #1398 & #1516 Read feed and change feed: Adds serialization optimization to reduce memory and CPU

utilization up to 90%. Objects are now passed as an array to the serializer.

- #1516 Query: Adds serialization optimization to reduce memory up to %50 and CPU utilization up to 25%. Objects are now passed as an array to the serializer.

Fixed

- #1401 & #1437: Response type: Fix deadlock on scenarios with `SynchronizationContext` when using `Response.Container`.
- #1445 Transport: Fix `ServicePoint` for `WebAssembly`.
- #1462 UserAgent: Fix feature usage tracking.
- #1469 Diagnostics: Fix `InvalidOperationException` and converts elapsed time to millisecond.
- #1512 PartitionRoutingHelper: Fix ReadFeed `ArgumentNullException` due to container cache miss.
- #1530 CosmosClientBuilder: Fix WithDefaultTimeToLive parameter spelling.

3.9.0-preview3 - 2020-05-11

Added

- #1356 & #1407 & #1428 Autoscale preview release.
- #1407 Autoscale: Adds `CreateDatabaseIfNotExistsAsync` and `CreateContainerIfNotExistsAsync` methods.
- #1410 FeedRange: Adds Json serialization support.
- #1398 Diagnostics: Adds CPU monitoring for .NET Core.
- #1441 Transport: Adds `HttpClientFactory` support on `CosmosClientOptions`.
- #1457 Container: Adds database reference to the container.
- #1453 Response factory: Adds a response factory to the public API.
- #1455 Serializer: Adds SDK serializer to `Client.ClientOptions.Serializer`.
- #1397 CosmosClientBuilder: Adds preferred regions and public internal func `WithConnectionModeDirect()`.
- #1439 No content on response: Adds the ability to have operation return no content from Azure Cosmos DB.
- #1469 Diagnostics: Fixes the `InvalidOperationException` and converts elapsed time to millisecond.

Fixed

- #1398 Reduced memory allocations on query deserialization.
- #1401 & #1437: Response type: Fixes deadlock on scenarios with `SynchronizationContext` when using `Response.Container`.
- #1445 Transport: Fixes `ServicePoint` for WebAssembly.
- #1462 UserAgent: Fixes feature usage tracking.

3.9.0-preview - 2020-04-17

Added

- #1356 Autoscale: Adds to public preview release

3.8.0-preview - 2020-04-16

Added

- #1331 Enabled client encryption / decryption for transactional batch requests

Fixed

- #1369 Fixes decryption for 'order by' query results

3.8.0 - 2020-04-07

Added

- #1314 Adds configuration for proactive TCP end-of-connection detection.
- #1305 Adds support for preferred region customization.

Fixed

- #1312 Fixes null reference when using default(PartitionKey).
- #1296 Decrypts the encrypted properties before returning query result.
- #1345 Fixes get query plan diagnostics.

3.7.1-preview - 2020-03-30

- #1210 Adds change feed pull model.
- #1242 Client encryption - fixes bug in read path without encrypted properties.
- #1314 Adds configuration for proactive TCP end-of-connection detection.
- #1312 Fixes null reference when using default(PartitionKey).
- #1296 Decrypts the encrypted properties before returning query result.

3.7.0 - 2020-03-26

Added

- #1268 Adds `GetElapsedClientLatency` to `CosmosDiagnostics`.
- #1239 Made `MultiPolygon` and `PolygonCoordinates` classes public.
- #1233 Partition key now supports operators `==, !=` for equality comparison.
- #1285 Add query plan retrieval to diagnostics.
- #1289 Query `ORDER BY` resume optimization.
- #1074 Bulk API congestion control.

Fixed

- #1213 `CosmosException` now returns the original stack trace.
- #1213 `ResponseMessage.ErrorMessage` is now always correctly populated. There was bug in some scenarios where the error message was left in the content stream.
- #1298 `CosmosException.Message` contains the same information as `CosmosException.ToString()` to ensure all the information is being tracked.
- #1242 Client encryption - Fixes bug in read path without encrypted properties.
- #1189 Query diagnostics shows correct overall time.
- #1189 Fixes a bug that caused duplicate information in diagnostic context.
- #1263 Fixes a bug where retry after interval did not get set on query stream responses.
- #1198 Fixes null reference exception when calling a disposed `CosmosClient`.
- #1274 `ObjectDisposedException` is thrown when calling all SDK objects like Database and Container that reference a disposed client.
- #1268 Fixes bug where Request Options was getting lost for `Database.ReadStreamAsync` and `Database.DeleteStreamAsync` methods.
- #1304 Fixes XML documentation so it now is visible in Visual Studio.

3.7.0-preview2 - 2020-03-09

- #1210 Change feed pull model.
- #1242 Client encryption - fixes bug in read path without encrypted properties.

3.7.0-preview - 2020-02-25

- #1074 Bulk API congestion control.
- #1210 Change feed pull model.

3.6.0 - 2020-01-23

- #1105 CosmosClient Immutability + Disposable Fixes

Added

- #1097 `GeospatialConfig` to `ContainerProperties`, `BoundingBoxProperties` to `SpatialPath`.
- #1061 Stream payload to `ExecuteStoredProcedureStreamAsync`.

- #1062 Additional diagnostic information including the ability to track time through the different SDK layers.
- #1107 Source Link support.
- #1121 `StandByFeedIterator` breath-first read strategy.

Fixed

- #1105 Custom serializer no longer calls SDK owned types that would cause serialization exceptions.
- #1112 Fixes SDK properties like `DatabaseProperties` to have the same JSON attributes.
- #1116 Fixes a deadlock on scenarios with `SynchronizationContext` while executing async query operations.
- #1143 Fixes permission resource link and authorization issue when doing a query with resource token for a specific partition key.
- #1150 Fixes `NullReferenceException` when using a non-existent Lease Container.

3.5.1 - 2019-12-11

Fixed

- #1060 Fixes unicode encoding bug in DISTINCT queries.
- #1070 `CreateItem` will only retry for auto-extracted partition key in-case of collection re-creation.
- #1075 Including header size details for bad request with large headers.
- #1078 Fixes a deadlock on scenarios with `SynchronizationContext` while executing async SDK API.
- #1081 Fixes race condition in serializer caused by null reference exception.
- #1086 Fix possible `NullReferenceException` on a `TransactionalBatch` code path.
- #1091 Fixes a bug in query when a partition split occurs that causes a `NotImplementedException` to be thrown.
- #1089 Fixes a `NullReferenceException` when using Bulk with items without partition key.

3.5.0 - 2019-12-03

Added

- #979 Make `SessionToken` on `QueryRequestOptions` public.
- #995 Included session token in diagnostics.
- #1000 Adds `PortReuseMode` to `CosmosClientOptions`.
- #1017 Adds `ClientSideRequestStatistics` to gateway calls and making end time nullable.
- #1038 Adds self-link to resource properties.

Fixed

- #921 Fixes error handling to preserve stack trace in certain scenarios.
- #944 Change feed processor won't use user serializer for internal operations.
- #988 Fixes query mutating due to retry of gone / name cache is stale.
- #954 Support start from beginning for change feed processor in multi master accounts.
- #999 Fixes grabbing extra page, updated continuation token on exception path, and non-Ascii character in order by continuation token.
- #1013 Gateway `OperationCanceledExceptions` are now returned as request timeouts.
- #1020 Direct package update removes debug statements.
- #1023 Fixes `ThroughputResponse.IsReplacePending` header mapping.
- #1036 Fixes query responses to return null Content if it is a failure.
- #1045 Adds stack trace and inner exception to `CosmosException`.
- #1050 Adds mocking constructors to `TransactionalBatchOperationResult`.

3.4.1 - 2019-11-06

Fixed

- #978 Fixes mocking for `FeedIterator` and response classes.

3.4.0 - 2019-11-04

Added

- #853 ORDER BY arrays and object support.
- #877 Query diagnostics now contains client-side request diagnostics information.
- #923 Bulk support is now public.
- #922 Included information of bulk support usage in user agent.
- #934 Preserved the ordering of projections in a `GROUP BY` query.
- #952 ORDER BY undefined and mixed type `ORDER BY` support.
- #965 Batch API is now public.

Fixed

- #901 Fixes a bug causing query response to create a new stream for each content call.
- #918 Fixes serializer being used for scripts, permissions, and conflict-related iterators.
- #936 Fixes bulk requests with large resources to have natural exception.

3.3.3 - 2019-10-30

- #837 Fixes group by bug for non-Windows platforms.
- #927 Fixes query returning partial results instead of error.

3.3.2 - 2019-10-16

Fixed

- #905 Fixes lINQ camel case bug.

3.3.1 - 2019-10-11

Fixed

- #895 Fixes user agent bug that caused format exceptions on non-Windows platforms.

3.3.0 - 2019-10-09

Added

- #801 Enabled LINQ `ThenBy` operator after `OrderBy`.
- #814 Ability to limit to configured endpoint only.
- #822 `GROUP BY` query support.
- #844 Adds `PartitionKeyDefinitionVersion` to container builder.

Fixed

- #835 Fixes a bug that caused sorted ranges exceptions.
- #846 Statistics not getting populated correctly on `CosmosException`.
- #857 Fixes reusability of the bulk support across container instances.
- #860 Fixes base user agent string.
- #876 Default connection time out reduced from 60 s to 10 s.

3.2.0 - 2019-09-17

Added

- #100 Configurable Tcp settings to `CosmosClientOptions`.
- #615, #775 Adds request diagnostics to response.
- #622 Adds CRUD and query operations for users and permissions, which enables ResourceToken support.
- #716 Added camel case serialization on LINQ query generation.
- #729, #776 Adds aggregate (CountAsync/SumAsync etc.) extensions for LINQ query.
- #743 Adds `WebProxy` to `CosmosClientOptions`.

Fixed

- #726 Query iterator `HasMoreResults` now returns false if an exception is hit.

- #705 User agent suffix gets truncated.
- #753 Reason was not being propagated for conflict exceptions.
- #756 Change feed processor with `WithStartTime` would execute the delegate the first time with no items.
- #761 `CosmosClient` deadlocks when using a custom task scheduler like Orleans.
- #769 Session consistency and gateway mode session-token bug fix- under few rare non-success cases session token might be in-correct.
- #772 Fixes throughput throwing when custom serializer used or offer doesn't exists.
- #785 Incorrect key to throw `CosmosExceptions` with `HttpStatusCode.Unauthorized` status code.

3.2.0-preview2 - 2019-09-10

- #585, #741 Bulk execution support.
- #427 Transactional batch support (Item CRUD).

3.2.0-preview - 2019-08-09

- #427 Transactional batch support (Item CRUD).

3.1.1 - 2019-08-12

Added

- #650 `CosmosSerializerOptions` to customize serialization.

Fixed

- #612 Bug fix for `ReadFeed` with partition key.
- #614 Fixes spatial path serialization and compatibility with older index versions.
- #619 Fixes `PInvokeStackImbalance` exception for .NET framework.
- #626 `FeedResponse<T>` status codes now return OK for success instead of the invalid status code 0 or Accepted.
- #629 Fixes `CreateContainerIfNotExistsAsync` validation to limited to partition key path only.
- #630 Fixes user agent to contain environment and package information.

3.1.0 - 2019-07-29 - Unlisted

Added

- #541 Adds consistency level to client and query options.
- #544 Adds continuation token support for LINQ.
- #557 Adds trigger options to item request options.
- #572 Adds partition key validation on `CreateContainerIfNotExistsAsync`.
- #581 Adds LINQ to `QueryDefinition` API.
- #592 Adds `CreateIfNotExistsAsync` to container builder.
- #597 Adds continuation token property to `ResponseMessage`.
- #604 Adds LINQ `ToStreamIterator` extension method.

Fixed

- #548 Fixes mis-typed message in `CosmosException.ToString()`.
- #558 Location cache `ConcurrentDict` lock contentions fix.
- #561 `GetItemLinqQueryable` now works with null query.
- #567 Query correctly handles different language cultures.
- #574 Fixes empty error message if query parsing fails from unexpected exception.
- #576 Query correctly serializes the input into a stream.

3.0.0 - 2019-07-15

- General availability of Version 3.0.0 of the .NET SDK.
- Targets .NET Standard 2.0, which supports .NET framework 4.6.1+ and .NET Core 2.0+.

- New object model, with top level `CosmosClient` and methods split across relevant database and container classes.
- New stream APIs that have high performance.
- Built-in support for change feed processor APIs.
- Fluent builder APIs for `CosmosClient`, container, and change feed processor.
- Idiomatic throughput management APIs.
- Granular `RequestOptions` and `ResponseTypes` for database, container, item, query, and throughput requests.
- Ability to scale non-partitioned containers.
- Extensible and customizable serializer.
- Extensible request pipeline with support for custom handlers.

Release & Retirement dates

Microsoft provides notification at least **12 months** in advance of retiring an SDK in order to smooth the transition to a newer/supported version. New features and functionality and optimizations are only added to the current SDK, as such it is recommended that you always upgrade to the latest SDK version as early as possible.

After **31 August 2022**, Azure Cosmos DB will no longer make bug fixes, add new features, and provide support for versions 1.x of the Azure Cosmos DB .NET or .NET Core SDK for SQL API. If you prefer not to upgrade, requests sent from version 1.x of the SDK will continue to be served by the Azure Cosmos DB service.

VERSION	RELEASE DATE	RETIREMENT DATE
3.6.0	January 23, 2020	---
3.5.1	December 11, 2019	---
3.5.0	December 03, 2019	---
3.4.1	November 06, 2019	---
3.4.0	November 04, 2019	---
3.3.3	October 30, 2019	---
3.3.2	October 16, 2019	---
3.3.1	October 11, 2019	---
3.3.0	October 8, 2019	---
3.2.0	September 18, 2019	---
3.1.1	August 12, 2019	---
3.1.0	July 29, 2019	---
3.0.0	July 15, 2019	---

FAQ

How will I be notified of the retiring SDK?

Microsoft will provide 12 month's advance notice before the end of support of the retiring SDK to facilitate a smooth transition to a supported SDK. We'll notify you through various communication channels: the Azure portal, Azure updates, and direct communication to assigned service administrators.

Can I author applications by using a to-be-retired Azure Cosmos DB SDK during the 12-month period?

Yes, you'll be able to author, deploy, and modify applications by using the to-be-retired Azure Cosmos DB SDK during the 12-month notice period. We recommend that you migrate to a newer supported version of the Azure Cosmos DB SDK during the 12-month notice period, as appropriate.

After the retirement date, what happens to applications that use the unsupported Azure Cosmos DB SDK?

After the retirement date, Azure Cosmos DB will no longer make bug fixes, add new features, or provide support to the retired SDK versions. If you prefer not to upgrade, requests sent from the retired versions of the SDK will continue to be served by the Azure Cosmos DB service.

Which SDK versions will have the latest features and updates?

New features and updates will be added only to the latest minor version of the latest supported major SDK version. We recommend that you always use the latest version to take advantage of new features, performance improvements, and bug fixes. If you're using an old, non-retired version of the SDK, your requests to Azure Cosmos DB will still function, but you won't have access to any new capabilities.

What should I do if I can't update my application before a cutoff date?

We recommend that you upgrade to the latest SDK as early as possible. After an SDK is tagged for retirement, you'll have 12 months to update your application. If you're not able to update by the retirement date, requests sent from the retired versions of the SDK will continue to be served by Azure Cosmos DB, so your running applications will continue to function. But Azure Cosmos DB will no longer make bug fixes, add new features, or provide support to the retired SDK versions.

If you have a support plan and require technical support, contact us by filing a support ticket.

See also

To learn more about Cosmos DB, see Microsoft Azure Cosmos DB service page.

Azure Cosmos DB .NET SDK v2 for SQL API: Download and release notes

11/2/2020 • 18 minutes to read • Edit Online

APPLIES TO: SQL API

SDK download	NuGet
API documentation	.NET API reference documentation
Samples	.NET code samples
Get started	Get started with the Azure Cosmos DB .NET SDK
Web app tutorial	Web application development with Azure Cosmos DB
Current supported framework	Microsoft .NET Framework 4.5

Release notes

> **NOTE**
> If you are using .NET Framework, please see the latest version 3.x of the .NET SDK, which targets .NET Standard.

The following sub versions of .NET SDKs are available under the 2.x.x version:

2.12.0

- Improved detection of regional outages for query operations that require obtaining a query plan.
- Improved CPU utilization during connectivity events when using ConnectionPolicy.EnableTcpConnectionEndpointRediscovery.
- Added RegexMatch system function for queries.
- Improved performance for Direct + TCP connection by adding pooling to frequently allocated objects

2.11.6

- Fixes "request headers is too long" for CRUD operations on stored procedures, triggers, and user defined functions

2.11.5

- Improved detection of regional failover scenarios during query operations.
- Improved resiliency during transient ReadSessionNotAvailable scenarios.

2.11.4

- Fix PortReuseMode setting in connection policy to be honored by DocumentClient.

2.11.3

- Added a cross-region retry mechanism for requests with transient connectivity issues to a particular

region. This retry mechanism uses (and requires) the user's defined `ConnectionPolicy.PreferredLocations` preference list.

- Added diagnostics information to operation canceled exceptions.

2.11.2

- Fix user session token in query FeedOption for request level session consistency

2.11.1

- Fix CPU issues related to usage of ConnectionPolicy.EnableTcpConnectionEndpointRediscovery
- Fix PartitionKey not being correctly passed on GROUP BY queries
- Fix possible StackOverflowException in retry scenario by adding Task.Yeild. Related fix for .NET Core 3

2.11.0

- Add RequestDiagnosticsString property to the StoredProcedureResponse
- Fix to improve the accurancy of heuristics applied to ConnectionPolicy.SetCurrentLocation
- Connectivity errors on .NET core using Direct + TCP mode now includes CPU usage history.

2.10.3

- Fix socket exception thrown from TCP connection closure under edge cases for closed connection

2.10.1

- Fix null reference exceptions when doing a query. This is fixes a bug in the fallback logic when the Microsoft.Azure.Documents.ServiceInterop.dll is not found.

2.10.0

- Add support for creating geometry collections
- Add support to specify bounding box for geometry spatial index
- Add EnableTcpConnectionEndpointRediscovery to ConnectionPolicy which enables invalidation of addresses based upon connection close notifications
- Fixed aggregate query RuntimeBinderException 'Cannot convert type'
- Fixed permission serialization to include the Token
- Improve retry logic for transient region failures where the SDK cannot connect to a specific endpoint and gets HttpException.
- Improve latency by reducing default RequestTimeout from 60 seconds to 10 seconds

2.9.4

- Fixed partition key not being honored for non windows x64 clients

2.9.3

- Fixed timer pool leak in Direct TCP mode
- Fixed broken links in documentation
- Too large of header now traces the header sizes
- Reduced header size by excluding session token in get query plan calls

2.9.2

- Fixed non ascii character in order by continuation token

2.9.1

- Fix Microsoft.Azure.Documents.ServiceInterop.dll graceful fallback bug Issue #750

- Add support for GROUP BY queries
- Query now retrieves query plan before execution in order to ensure consistent behavior between single partition and cross partition queries.

2.8.1

- Added RequestDiagnosticsString to FeedResponse
- Fixed serialization settings for upsert and replace document

2.7.0

- Added support for arrays and objects in order by queries
- Handle effective partition key collisions
- Added LINQ support for multiple OrderBy operators with ThenBy operator
- Fixed AysncCache deadlock issue so that it will work with a single-threaded task scheduler

2.6.0

- Added PortReusePolicy to ConnectionPolicy
- Fixed ntdll!RtlGetVersion TypeLoadException issue when SDK is used in a UWP app

2.5.1

- SDK's System.Net.Http version now matches what is defined in the NuGet package
- Allow write requests to fallback to a different region if the original one fails
- Add session retry policy for write request

2.4.2

- Made assembly version and file version same as nuget package version.

2.4.1

- Fixes tracing race condition for queries which caused empty pages

2.4.0

- Increased decimal precision size for LINQ queries.
- Added new classes CompositePath, CompositePathSortOrder, SpatialSpec, SpatialType and PartitionKeyDefinitionVersion
- Added TimeToLivePropertyPath to DocumentCollection
- Added CompositeIndexes and SpatialIndexes to IndexPolicy
- Added Version to PartitionKeyDefinition
- Added None to PartitionKey
- Fix a bug to properly handle non-JSON payload that would cause JsonReaderException

2.3.0

- Added IdleTcpConnectionTimeout, OpenTcpConnectionTimeout, MaxRequestsPerTcpConnection and MaxTcpConnectionsPerEndpoint to ConnectionPolicy.

2.2.3

- Diagnostics improvements

2.2.2

- Added environment variable setting "POCOSerializationOnly".
- Removed DocumentDB.Spatial.Sql.dll and now included in Microsoft.Azure.Documents.ServiceInterop.dll

2.2.1

- Improvement in retry logic during failover for StoredProcedure execute calls.
- Made DocumentClientEventSource singleton.
- Fix GatewayAddressCache timeout not honoring ConnectionPolicy RequestTimeout.

2.2.0

- For direct/TCP transport diagnostics, added TransportException, an internal exception type of the SDK. When present in exception messages, this type prints additional information for troubleshooting client connectivity problems.
- Added new constuctor overload which takes a HttpMessageHandler, a HTTP handler stack to use for sending HttpClient requests (e.g., HttpClientHandler).
- Fix bug where header with null values were not being handled properly.
- Improved collection cache validation.

2.1.3

- Updated System.Net.Security to 4.3.2.

2.1.2

- Diagnostic tracing improvements.

2.1.1

- Added more resilience to Multi-region request transient failures.

2.1.0

- Added Multi-region write support.
- Cross partition query performance improvements with TOP.
- Fixed bug where MaxBufferedItemCount was not being honored causing out of memory issues.

2.0.0

- Added request cancellation support.
- Added SetCurrentLocation to ConnectionPolicy, which automatically populates the preferred locations based on the region.
- Fixed Bug in Cross Partition Queries with Min/Max and a filter that matches no documents on an individual partition.
- DocumentClient methods now have parity with IDocumentClient.
- Updated direct TCP transport stack to reduce the number of connections established.
- Added support for Direct Mode TCP for non-Windows clients.

2.0.0-preview2

- Added request cancellation support.
- Added SetCurrentLocation to ConnectionPolicy, which automatically populates the preferred locations based on the region.
- Fixed Bug in Cross Partition Queries with Min/Max and a filter that matches no documents on an individual partition.

2.0.0-preview

- DocumentClient methods now have parity with IDocumentClient.
- Updated direct TCP transport stack to reduce the number of connections established.
- Added support for Direct Mode TCP for non-Windows clients.

1.22.0

- Added ConsistencyLevel Property to FeedOptions.
- Added JsonSerializerSettings to RequestOptions and FeedOptions.
- Added EnableReadRequestsFallback to ConnectionPolicy.

1.21.1

- Fixed KeyNotFoundException for cross partition order by queries in corner cases.
- Fixed bug where JsonPropery attribute in select clause for LINQ queries was not being honored.

1.20.2

- Fixed bug that is hit under certain race conditions, that results in intermittent "Microsoft.Azure.Documents.NotFoundException: The read session is not available for the input session token" errors when using Session consistency level.

1.20.1

- Improved cross partition query performance when the MaxDegreeOfParallelism property is set to -1 in FeedOptions.
- Added a new ToString() function to QueryMetrics.
- Exposed partition statistics on reading collections.
- Added PartitionKey property to ChangeFeedOptions.
- Minor bug fixes.

1.19.1

- Adds the ability to specify unique indexes for the documents by using UniqueKeyPolicy property on the DocumentCollection.
- Fixed a bug in which the custom JsonSerializer settings were not being honored for some queries and stored procedure execution.

1.19.0

- Branding change from Azure DocumentDB to Azure Cosmos DB in the API Reference documentation, metadata information in assemblies, and the NuGet package.
- Expose diagnostic information and latency from the response of requests sent with direct connectivity mode. The property names are RequestDiagnosticsString and RequestLatency on ResourceResponse class.
- This SDK version requires the latest version of Azure Cosmos DB Emulator available for download from https://aka.ms/cosmosdb-emulator.

1.18.1

- Internal changes for Microsoft friends assemblies.

1.18.0

- Added several reliability fixes and improvements.

1.17.0

- Added support for PartitionKeyRangeId as a FeedOption for scoping query results to a specific partition key range value.
- Added support for StartTime as a ChangeFeedOption to start looking for the changes after that time.

1.16.1

- Fixed an issue in the JsonSerializable class that may cause a stack overflow exception.

1.16.0

- Fixed an issue that required recompiling of the application due to the introduction of JsonSerializerSettings as an optional parameter in the DocumentClient constructor.
- Marked the DocumentClient constructor obsolete that required JsonSerializerSettings as the last parameter to allow for default values of ConnectionPolicy and ConsistencyLevel parameters when passing in JsonSerializerSettings parameter.

1.15.0

- Added support for specifying custom JsonSerializerSettings while instantiating DocumentClient.

1.14.1

- Fixed an issue that affected x64 machines that don't support SSE4 instruction and throw an SEHException when running Azure Cosmos DB DocumentDB API queries.

1.14.0

- Added support for Request Unit per Minute (RU/m) feature.
- Added support for a new consistency level called ConsistentPrefix.
- Added support for query metrics for individual partitions.
- Added support for limiting the size of the continuation token for queries.
- Added support for more detailed tracing for failed requests.
- Made some performance improvements in the SDK.

1.13.4

- Functionally same as 1.13.3. Made some internal changes.

1.13.3

- Functionally same as 1.13.2. Made some internal changes.

1.13.2

- Fixed an issue that ignored the PartitionKey value provided in FeedOptions for aggregate queries.
- Fixed an issue in transparent handling of partition management during mid-flight cross-partition Order By query execution.

1.13.1

- Fixed an issue which caused deadlocks in some of the async APIs when used inside ASP.NET context.

1.13.0

- Fixes to make SDK more resilient to automatic failover under certain conditions.

1.15.0

- Fix for an issue that occasionally causes a WebException: The remote name could not be resolved.
- Added the support for directly reading a typed document by adding new overloads to ReadDocumentAsync API.

1.12.1

- Added LINQ support for aggregation queries (COUNT, MIN, MAX, SUM, and AVG).
- Fix for a memory leak issue for the ConnectionPolicy object caused by the use of event handler.
- Fix for an issue wherein UpsertAttachmentAsync was not working when ETag was used.
- Fix for an issue wherein cross partition order-by query continuation was not working when sorting on string field.

1.12.0

- Added support for aggregation queries (COUNT, MIN, MAX, SUM, and AVG). See Aggregation support.
- Lowered minimum throughput on partitioned collections from 10,100 RU/s to 2500 RU/s.

1.11.4

- Fix for an issue wherein some of the cross-partition queries were failing in the 32-bit host process.
- Fix for an issue wherein the session container was not being updated with the token for failed requests in Gateway mode.
- Fix for an issue wherein a query with UDF calls in projection was failing in some cases.

1.11.3

- Fix for an issue wherein the session container was not being updated with the token for failed requests.
- Added support for the SDK to work in a 32-bit host process. Note that if you use cross partition queries, 64-bit host processing is recommended for improved performance.
- Improved performance for scenarios involving queries with a large number of partition key values in an IN expression.

1.11.1

- Minor performance fix for the CreateDocumentCollectionIfNotExistsAsync API introduced in 1.11.0.
- Peformance fix in the SDK for scenarios that involve high degree of concurrent requests.

1.11.0

- Support for new classes and methods to process the change feed of documents within a collection.
- Support for cross-partition query continuation and some perf improvements for cross-partition queries.
- Addition of CreateDatabaseIfNotExistsAsync and CreateDocumentCollectionIfNotExistsAsync methods.
- LINQ support for system functions: IsDefined, IsNull and IsPrimitive.
- Fix for automatic binplacing of Microsoft.Azure.Documents.ServiceInterop.dll and DocumentDB.Spatial.Sql.dll assemblies to application's bin folder when using the Nuget package with projects that have project.json tooling.
- Support for emitting client side ETW traces which could be helpful in debugging scenarios.

1.10.0

- Added direct connectivity support for partitioned collections.
- Improved performance for the Bounded Staleness consistency level.
- Added LINQ support for StringEnumConverter, IsoDateTimeConverter and UnixDateTimeConverter while translating predicates.
- Various SDK bug fixes.

1.9.5

- Fixed an issue that caused the following NotFoundException: The read session is not available for the input session token. This exception occurred in some cases when querying for the read-region of a geo-distributed account.
- Exposed the ResponseStream property in the ResourceResponse class, which enables direct access to the underlying stream from a response.

1.9.4

- Modified the ResourceResponse, FeedResponse, StoredProcedureResponse and MediaResponse classes to implement the corresponding public interface so that they can be mocked for test driven deployment (TDD).
- Fixed an issue that caused a malformed partition key header when using a custom JsonSerializerSettings object for serializing data.

1.9.3

- Fixed an issue that caused long running queries to fail with error: Authorization token is not valid at the current time.
- Fixed an issue that removed the original SqlParameterCollection from cross partition top/order-by queries.

1.9.2

- Added support for parallel queries for partitioned collections.

- Added support for cross partition ORDER BY and TOP queries for partitioned collections.
- Fixed the missing references to DocumentDB.Spatial.Sql.dll and Microsoft.Azure.Documents.ServiceInterop.dll that are required when referencing a DocumentDB project with a reference to the DocumentDB Nuget package.
- Fixed the ability to use parameters of different types when using user defined functions in LINQ.
- Fixed a bug for globally replicated accounts where Upsert calls were being directed to read locations instead of write locations.
- Added methods to the IDocumentClient interface that were missing:
  - UpsertAttachmentAsync method that takes mediaStream and options as parameters.
  - CreateAttachmentAsync method that takes options as a parameter.
  - CreateOfferQuery method that takes querySpec as a parameter.
- Unsealed public classes that are exposed in the IDocumentClient interface.

1.8.0

- Added the support for multi-region database accounts.
- Added support for retry on throttled requests.User can customize the number of retries and the max wait time by configuring the ConnectionPolicy.RetryOptions property.
- Added a new IDocumentClient interface that defines the signatures of all DocumenClient properties and methods. As part of this change, also changed extension methods that create IQueryable and IOrderedQueryable to methods on the DocumentClient class itself.
- Added configuration option to set the ServicePoint.ConnectionLimit for a given DocumentDB endpoint Uri. Use ConnectionPolicy.MaxConnectionLimit to change the default value, which is set to 50.
- Deprecated IPartitionResolver and its implementation. Support for IPartitionResolver is now obsolete. It's recommended that you use Partitioned Collections for higher storage and throughput.

1.7.1

- Added an overload to Uri based ExecuteStoredProcedureAsync method that takes RequestOptions as a parameter.

1.7.0

- Added time to live (TTL) support for documents.

1.6.3

- Fixed a bug in Nuget packaging of .NET SDK for packaging it as part of a Azure Cloud Service solution.

1.6.2

- Implemented partitioned collections and user-defined performance levels.

1.5.3

- **[Fixed]** Querying DocumentDB endpoint throws: 'System.Net.Http.HttpRequestException: Error while copying content to a stream.

1.5.2

- Expanded LINQ support including new operators for paging, conditional expressions and range comparison.
  - Take operator to enable SELECT TOP behavior in LINQ.
  - CompareTo operator to enable string range comparisons.
  - Conditional (?) and coalesce operators (??).
- **[Fixed]** ArgumentOutOfRangeException when combining Model projection with Where-In in linq

query. #81

1.5.1

- **[Fixed]** If Select is not the last expression the LINQ Provider assumed no projection and produced SELECT * incorrectly. #58

1.5.0

- Implemented Upsert, Added UpsertXXXAsync methods.
- Performance improvements for all requests.
- LINQ Provider support for conditional, coalesce and CompareTo methods for strings.
- **[Fixed]** LINQ provider --> Implement Contains method on List to generate the same SQL as on IEnumerable and Array.
- **[Fixed]** BackoffRetryUtility uses the same HttpRequestMessage again instead of creating a new one on retry.
- **[Obsolete]** UriFactory.CreateCollection --> should now use UriFactory.CreateDocumentCollection.

1.4.1

- **[Fixed]** Localization issues when using non en culture info such as nl-NL etc.

1.4.0

- ID Based Routing
  - New UriFactory helper to assist with constructing ID based resource links.
  - New overloads on DocumentClient to take in URI.
- Added IsValid() and IsValidDetailed() in LINQ for geospatial.
- LINQ Provider support enhanced.
  - **Math** - Abs, Acos, Asin, Atan, Ceiling, Cos, Exp, Floor, Log, Log10, Pow, Round, Sign, Sin, Sqrt, Tan, Truncate.
  - **String** - Concat, Contains, EndsWith, IndexOf, Count, ToLower, TrimStart, Replace, Reverse, TrimEnd, StartsWith, SubString, ToUpper.
  - **Array** - Concat, Contains, Count.
  - **IN** operator.

1.3.0

- Added support for modifying indexing policies.
  - New ReplaceDocumentCollectionAsync method in DocumentClient.
  - New IndexTransformationProgress property in ResourceResponse for tracking percent progress of index policy changes.
  - DocumentCollection.IndexingPolicy is now mutable.
- Added support for spatial indexing and query
  - New Microsoft.Azure.Documents.Spatial namespace for serializing/deserializing spatial types like Point and Polygon.
  - New SpatialIndex class for indexing GeoJSON data stored in DocumentDB.
- **[Fixed]** : Incorrect SQL query generated from linq expression. #38

1.2.0

- Dependency on Newtonsoft.Json v5.0.7.
- Changes to support Order By:
  - LINQ provider support for OrderBy() or OrderByDescending().

- IndexingPolicy to support Order By.

NB: Possible breaking change

If you have existing code that provisions collections with a custom indexing policy, then your existing code will need to be updated to support the new IndexingPolicy class. If you have no custom indexing policy, then this change does not affect you.

1.1.0

- Support for partitioning data by using the new HashPartitionResolver and RangePartitionResolver classes and the IPartitionResolver.
- DataContract serialization.
- Guid support in LINQ provider.
- UDF support in LINQ.

1.0.0

- GA SDK.

Release & Retirement dates

Microsoft provides notification at least **12 months** in advance of retiring an SDK in order to smooth the transition to a newer/supported version. New features and functionality and optimizations are only added to the current SDK, as such it is recommended that you always upgrade to the latest SDK version as early as possible.

After 31 August 2022, Azure Cosmos DB will no longer make bug fixes, add new features, and provide support to versions 1.x of the Azure Cosmos DB .NET or .NET Core SDK for SQL API. If you prefer not to upgrade, requests sent from version 1.x of the SDK will continue to be served by the Azure Cosmos DB service.

VERSION	RELEASE DATE	RETIREMENT DATE
2.12.0	October 7, 2020	---
2.11.6	August 12, 2020	---
2.11.5	August 4, 2020	---
2.11.4	July 30, 2020	---
2.11.3	July 29, 2020	---
2.11.2	July 14, 2020	---
2.11.1	July 1, 2020	---
2.9.2	November 15, 2019	---
2.9.1	November 13, 2019	---
2.9.0	October 30, 2019	---
2.8.1	October 11, 2019	---

VERSION	RELEASE DATE	RETIREMENT DATE
2.7.0	September 23, 2019	---
2.6.0	August 30, 2019	---
2.5.1	July 02, 2019	---
2.4.1	June 20, 2019	---
2.4.0	May 05, 2019	---
2.3.0	April 04, 2019	---
2.2.3	February 11, 2019	---
2.2.2	February 06, 2019	---
2.2.1	December 24, 2018	---
2.2.0	December 07, 2018	---
2.1.3	October 15, 2018	---
2.1.2	October 04, 2018	---
2.1.1	September 27, 2018	---
2.1.0	September 21, 2018	---
2.0.0	September 07, 2018	---
1.22.0	April 19, 2018	August 31, 2022
1.21.1	March 09, 2018	August 31, 2022
1.20.2	February 21, 2018	August 31, 2022
1.20.1	February 05, 2018	August 31, 2022
1.19.1	November 16, 2017	August 31, 2022
1.19.0	November 10, 2017	August 31, 2022
1.18.1	November 07, 2017	August 31, 2022
1.18.0	October 17, 2017	August 31, 2022
1.17.0	August 10, 2017	August 31, 2022
1.16.1	August 07, 2017	August 31, 2022

VERSION	RELEASE DATE	RETIREMENT DATE
1.16.0	August 02, 2017	August 31, 2022
1.15.0	June 30, 2017	August 31, 2022
1.14.1	May 23, 2017	August 31, 2022
1.14.0	May 10, 2017	August 31, 2022
1.13.4	May 09, 2017	August 31, 2022
1.13.3	May 06, 2017	August 31, 2022
1.13.2	April 19, 2017	August 31, 2022
1.13.1	March 29, 2017	August 31, 2022
1.13.0	March 24, 2017	August 31, 2022
1.12.1	March 14, 2017	August 31, 2022
1.12.0	February 15, 2017	August 31, 2022
1.11.4	February 06, 2017	August 31, 2022
1.11.3	January 26, 2017	August 31, 2022
1.11.1	December 21, 2016	August 31, 2022
1.11.0	December 08, 2016	August 31, 2022
1.10.0	September 27, 2016	August 31, 2022
1.9.5	September 01, 2016	August 31, 2022
1.9.4	August 24, 2016	August 31, 2022
1.9.3	August 15, 2016	August 31, 2022
1.9.2	July 23, 2016	August 31, 2022
1.8.0	June 14, 2016	August 31, 2022
1.7.1	May 06, 2016	August 31, 2022
1.7.0	April 26, 2016	August 31, 2022
1.6.3	April 08, 2016	August 31, 2022
1.6.2	March 29, 2016	August 31, 2022

VERSION	RELEASE DATE	RETIREMENT DATE
1.5.3	February 19, 2016	August 31, 2022
1.5.2	December 14, 2015	August 31, 2022
1.5.1	November 23, 2015	August 31, 2022
1.5.0	October 05, 2015	August 31, 2022
1.4.1	August 25, 2015	August 31, 2022
1.4.0	August 13, 2015	August 31, 2022
1.3.0	August 05, 2015	August 31, 2022
1.2.0	July 06, 2015	August 31, 2022
1.1.0	April 30, 2015	August 31, 2022
1.0.0	April 08, 2015	August 31, 2022

FAQ

How will I be notified of the retiring SDK?

Microsoft will provide 12 month's advance notice before the end of support of the retiring SDK to facilitate a smooth transition to a supported SDK. We'll notify you through various communication channels: the Azure portal, Azure updates, and direct communication to assigned service administrators.

Can I author applications by using a to-be-retired Azure Cosmos DB SDK during the 12-month period?

Yes, you'll be able to author, deploy, and modify applications by using the to-be-retired Azure Cosmos DB SDK during the 12-month notice period. We recommend that you migrate to a newer supported version of the Azure Cosmos DB SDK during the 12-month notice period, as appropriate.

After the retirement date, what happens to applications that use the unsupported Azure Cosmos DB SDK?

After the retirement date, Azure Cosmos DB will no longer make bug fixes, add new features, or provide support to the retired SDK versions. If you prefer not to upgrade, requests sent from the retired versions of the SDK will continue to be served by the Azure Cosmos DB service.

Which SDK versions will have the latest features and updates?

New features and updates will be added only to the latest minor version of the latest supported major SDK version. We recommend that you always use the latest version to take advantage of new features, performance improvements, and bug fixes. If you're using an old, non-retired version of the SDK, your requests to Azure Cosmos DB will still function, but you won't have access to any new capabilities.

What should I do if I can't update my application before a cutoff date?

We recommend that you upgrade to the latest SDK as early as possible. After an SDK is tagged for retirement, you'll have 12 months to update your application. If you're not able to update by the retirement date, requests sent from the retired versions of the SDK will continue to be served by Azure Cosmos DB, so

your running applications will continue to function. But Azure Cosmos DB will no longer make bug fixes, add new features, or provide support to the retired SDK versions.

If you have a support plan and require technical support, contact us by filing a support ticket.

See also

To learn more about Cosmos DB, see Microsoft Azure Cosmos DB service page.

Azure Cosmos DB .NET Core SDK v2 for SQL API: Release notes and resources

11/2/2020 • 16 minutes to read • Edit Online

APPLIES TO: SQL API

SDK download	NuGet
API documentation	.NET API reference documentation
Samples	.NET code samples
Get started	Get started with the Azure Cosmos DB .NET
Web app tutorial	Web application development with Azure Cosmos DB
Current supported framework	.NET Standard 1.6 and .NET Standard 1.5

Release Notes

> **NOTE**
> If you are using .NET Core, please see the latest version 3.x of the .NET SDK, which targets .NET Standard.

The following sub versions of .NET SDKs are available under the 2.x.x version:

2.12.0

- Improved detection of regional outages for query operations that require obtaining a query plan.
- Improved CPU utilization during connectivity events when using ConnectionPolicy.EnableTcpConnectionEndpointRediscovery.
- Added RegexMatch system function for queries.
- Improved performance for Direct + TCP connection by adding pooling to frequently allocated objects

2.11.6

- Fixes "request headers is too long" for CRUD operations on stored procedures, triggers, and user defined functions

2.11.5

- Improved detection of regional failover scenarios during query operations.
- Improved resiliency during transient ReadSessionNotAvailable scenarios.

2.11.4

- Fix PortReuseMode setting in connection policy to be honored by DocumentClient.

2.11.3

- Added a cross-region retry mechanism for requests with transient connectivity issues to a particular region.

This retry mechanism uses (and requires) the user's defined `ConnectionPolicy.PreferredLocations` preference list.
- Added diagnostics information to operation canceled exceptions.

2.11.2

- Fix user session token in query FeedOption for request level session consistency

2.11.1

- Fix CPU issues related to usage of ConnectionPolicy.EnableTcpConnectionEndpointRediscovery
- Fix PartitionKey not being correctly passed on GROUP BY queries
- Fix possible StackOverflowException in retry scenario by adding Task.Yeild. Related fix for .NET Core 3

2.11.0

- Add RequestDiagnosticsString property to the StoredProcedureResponse
- Fix to improve the accurancy of heuristics applied to ConnectionPolicy.SetCurrentLocation
- Connectivity errors on .NET core using Direct + TCP mode now includes CPU usage history.

2.10.3

- Fix socket exception thrown from TCP connection closure under edge cases for closed connection

2.10.1

- Fix null reference exceptions when doing a query. This is fixes a bug in the fallback logic when the Microsoft.Azure.Documents.ServiceInterop.dll is not found.

2.10.0

- Add support for creating geometry collections
- Add support to specify bounding box for geometry spatial index
- Add EnableTcpConnectionEndpointRediscovery to ConnectionPolicy which enables invalidation of addresses based upon connection close notifications
- Fixed aggregate query RuntimeBinderException 'Cannot convert type'
- Fixed permission serialization to include the Token
- Improve retry logic for transient region failures where the SDK cannot connect to a specific endpoint and gets HttpException.
- Improve latency by reducing default RequestTimeout from 60 seconds to 10 seconds

2.9.4

- Fixed partition key not being honored for non windows x64 clients

2.9.3

- Fixed timer pool leak in Direct TCP mode
- Fixed broken links in documentation
- Too large of header now traces the header sizes
- Reduced header size by excluding session token in get query plan calls

2.9.2

- Fixed non ascii character in order by continuation token

2.9.1

- Fix Microsoft.Azure.Documents.ServiceInterop.dll graceful fallback bug Issue #750

- Add support for GROUP BY queries
- Query now retrieves query plan before execution in order to ensure consistent behavior between single partition and cross partition queries.

2.8.1

- Added RequestDiagnosticsString to FeedResponse
- Fixed serialization settings for upsert and replace document

2.7.0

- Added support for arrays and objects in order by queries
- Handle effective partition key collisions
- Added LINQ support for multiple OrderBy operators with ThenBy operator
- Fixed AysncCache deadlock issue so that it will work with a single-threaded task scheduler

2.6.0

- Added PortReusePolicy to ConnectionPolicy
- Fixed ntdll!RtlGetVersion TypeLoadException issue when SDK is used in a UWP app

2.5.1

- SDK's System.Net.Http version now matches what is defined in the NuGet package
- Allow write requests to fallback to a different region if the original one fails
- Add session retry policy for write request

2.4.2

- Made assembly version and file version same as nuget package version.

2.4.1

- Fixes tracing race condition for queries which caused empty pages

2.4.0

- Increased decimal precision size for LINQ queries.
- Added new classes CompositePath, CompositePathSortOrder, SpatialSpec, SpatialType and PartitionKeyDefinitionVersion
- Added TimeToLivePropertyPath to DocumentCollection
- Added CompositeIndexes and SpatialIndexes to IndexPolicy
- Added Version to PartitionKeyDefinition
- Added None to PartitionKey
- Fix a bug to properly handle non-JSON payload that would cause JsonReaderException

2.3.0

- Added IdleTcpConnectionTimeout, OpenTcpConnectionTimeout, MaxRequestsPerTcpConnection and MaxTcpConnectionsPerEndpoint to ConnectionPolicy.

2.2.3

- Diagnostics improvements

2.2.2

- Added environment variable setting "POCOSerializationOnly".
- Removed DocumentDB.Spatial.Sql.dll and now included in Microsoft.Azure.Documents.ServiceInterop.dll

2.2.1

- Improvement in retry logic during failover for StoredProcedure execute calls.
- Made DocumentClientEventSource singleton.
- Fix GatewayAddressCache timeout not honoring ConnectionPolicy RequestTimeout.

2.2.0

- For direct/TCP transport diagnostics, added TransportException, an internal exception type of the SDK. When present in exception messages, this type prints additional information for troubleshooting client connectivity problems.
- Added new constuctor overload which takes a HttpMessageHandler, a HTTP handler stack to use for sending HttpClient requests (e.g., HttpClientHandler).
- Fix bug where header with null values were not being handled properly.
- Improved collection cache validation.

2.1.3

- Updated System.Net.Security to 4.3.2.

2.1.2

- Diagnostic tracing improvements.

2.1.1

- Added more resilience to Multi-region request transient failures.

2.1.0

- Added Multi-region write support.
- Cross partition query performance improvements with TOP.
- Fixed bug where MaxBufferedItemCount was not being honored causing out of memory issues.

2.0.0

- Added request cancellation support.
- Added SetCurrentLocation to ConnectionPolicy, which automatically populates the preferred locations based on the region.
- Fixed Bug in Cross Partition Queries with Min/Max and a filter that matches no documents on an individual partition.
- DocumentClient methods now have parity with IDocumentClient.
- Updated direct TCP transport stack to reduce the number of connections established.
- Added support for Direct Mode TCP for non-Windows clients.

2.0.0-preview2

- Added request cancellation support.
- Added SetCurrentLocation to ConnectionPolicy, which automatically populates the preferred locations based on the region.
- Fixed Bug in Cross Partition Queries with Min/Max and a filter that matches no documents on an individual partition.

2.0.0-preview

- DocumentClient methods now have parity with IDocumentClient.
- Updated direct TCP transport stack to reduce the number of connections established.
- Added support for Direct Mode TCP for non-Windows clients.

1.22.0

- Added ConsistencyLevel Property to FeedOptions.
- Added JsonSerializerSettings to RequestOptions and FeedOptions.
- Added EnableReadRequestsFallback to ConnectionPolicy.

1.21.1

- Fixed KeyNotFoundException for cross partition order by queries in corner cases.

- Fixed bug where JsonPropery attribute in select clause for LINQ queries was not being honored.

1.20.2

- Fixed bug that is hit under certain race conditions, that results in intermittent "Microsoft.Azure.Documents.NotFoundException: The read session is not available for the input session token" errors when using Session consistency level.

1.20.1

- Improved cross partition query performance when the MaxDegreeOfParallelism property is set to -1 in FeedOptions.
- Added a new ToString() function to QueryMetrics.
- Exposed partition statistics on reading collections.
- Added PartitionKey property to ChangeFeedOptions.
- Minor bug fixes.

1.19.1

- Adds the ability to specify unique indexes for the documents by using UniqueKeyPolicy property on the DocumentCollection.
- Fixed a bug in which the custom JsonSerializer settings were not being honored for some queries and stored procedure execution.

1.19.0

- Branding change from Azure DocumentDB to Azure Cosmos DB in the API Reference documentation, metadata information in assemblies, and the NuGet package.
- Expose diagnostic information and latency from the response of requests sent with direct connectivity mode. The property names are RequestDiagnosticsString and RequestLatency on ResourceResponse class.
- This SDK version requires the latest version of Azure Cosmos DB Emulator available for download from https://aka.ms/cosmosdb-emulator.

1.18.1

- Internal changes for Microsoft friends assemblies.

1.18.0

- Added several reliability fixes and improvements.

1.17.0

- Added support for PartitionKeyRangeId as a FeedOption for scoping query results to a specific partition key range value.
- Added support for StartTime as a ChangeFeedOption to start looking for the changes after that time.

1.16.1

- Fixed an issue in the JsonSerializable class that may cause a stack overflow exception.

1.16.0

- Fixed an issue that required recompiling of the application due to the introduction of JsonSerializerSettings as an optional parameter in the DocumentClient constructor.
- Marked the DocumentClient constructor obsolete that required JsonSerializerSettings as the last parameter to allow for default values of ConnectionPolicy and ConsistencyLevel parameters when passing in JsonSerializerSettings parameter.

1.15.0

- Added support for specifying custom JsonSerializerSettings while instantiating DocumentClient.

1.14.1

- Fixed an issue that affected x64 machines that don't support SSE4 instruction and throw an SEHException when running Azure Cosmos DB DocumentDB API queries.

1.14.0

- Added support for Request Unit per Minute (RU/m) feature.
- Added support for a new consistency level called ConsistentPrefix.
- Added support for query metrics for individual partitions.
- Added support for limiting the size of the continuation token for queries.
- Added support for more detailed tracing for failed requests.
- Made some performance improvements in the SDK.

1.13.4

- Functionally same as 1.13.3. Made some internal changes.

1.13.3

- Functionally same as 1.13.2. Made some internal changes.

1.13.2

- Fixed an issue that ignored the PartitionKey value provided in FeedOptions for aggregate queries.
- Fixed an issue in transparent handling of partition management during mid-flight cross-partition Order By query execution.

1.13.1

- Fixed an issue which caused deadlocks in some of the async APIs when used inside ASP.NET context.

1.13.0

- Fixes to make SDK more resilient to automatic failover under certain conditions.

1.15.0

- Fix for an issue that occasionally causes a WebException: The remote name could not be resolved.
- Added the support for directly reading a typed document by adding new overloads to ReadDocumentAsync API.

1.12.1

- Added LINQ support for aggregation queries (COUNT, MIN, MAX, SUM, and AVG).
- Fix for a memory leak issue for the ConnectionPolicy object caused by the use of event handler.
- Fix for an issue wherein UpsertAttachmentAsync was not working when ETag was used.
- Fix for an issue wherein cross partition order-by query continuation was not working when sorting on string field.

1.12.0

- Added support for aggregation queries (COUNT, MIN, MAX, SUM, and AVG). See Aggregation support.
- Lowered minimum throughput on partitioned collections from 10,100 RU/s to 2500 RU/s.

1.11.4

- Fix for an issue wherein some of the cross-partition queries were failing in the 32-bit host process.
- Fix for an issue wherein the session container was not being updated with the token for failed requests in Gateway mode.
- Fix for an issue wherein a query with UDF calls in projection was failing in some cases.

1.11.3

- Fix for an issue wherein the session container was not being updated with the token for failed requests.
- Added support for the SDK to work in a 32-bit host process. Note that if you use cross partition queries, 64-

bit host processing is recommended for improved performance.

- Improved performance for scenarios involving queries with a large number of partition key values in an IN expression.

1.11.1

- Minor performance fix for the CreateDocumentCollectionIfNotExistsAsync API introduced in 1.11.0.
- Peformance fix in the SDK for scenarios that involve high degree of concurrent requests.

1.11.0

- Support for new classes and methods to process the change feed of documents within a collection.
- Support for cross-partition query continuation and some perf improvements for cross-partition queries.
- Addition of CreateDatabaseIfNotExistsAsync and CreateDocumentCollectionIfNotExistsAsync methods.
- LINQ support for system functions: IsDefined, IsNull and IsPrimitive.
- Fix for automatic binplacing of Microsoft.Azure.Documents.ServiceInterop.dll and DocumentDB.Spatial.Sql.dll assemblies to application's bin folder when using the Nuget package with projects that have project.json tooling.
- Support for emitting client side ETW traces which could be helpful in debugging scenarios.

1.10.0

- Added direct connectivity support for partitioned collections.
- Improved performance for the Bounded Staleness consistency level.
- Added LINQ support for StringEnumConverter, IsoDateTimeConverter and UnixDateTimeConverter while translating predicates.
- Various SDK bug fixes.

1.9.5

- Fixed an issue that caused the following NotFoundException: The read session is not available for the input session token. This exception occurred in some cases when querying for the read-region of a geo-distributed account.
- Exposed the ResponseStream property in the ResourceResponse class, which enables direct access to the underlying stream from a response.

1.9.4

- Modified the ResourceResponse, FeedResponse, StoredProcedureResponse and MediaResponse classes to implement the corresponding public interface so that they can be mocked for test driven deployment (TDD).
- Fixed an issue that caused a malformed partition key header when using a custom JsonSerializerSettings object for serializing data.

1.9.3

- Fixed an issue that caused long running queries to fail with error: Authorization token is not valid at the current time.
- Fixed an issue that removed the original SqlParameterCollection from cross partition top/order-by queries.

1.9.2

- Added support for parallel queries for partitioned collections.
- Added support for cross partition ORDER BY and TOP queries for partitioned collections.
- Fixed the missing references to DocumentDB.Spatial.Sql.dll and Microsoft.Azure.Documents.ServiceInterop.dll that are required when referencing a DocumentDB project with a reference to the DocumentDB Nuget package.
- Fixed the ability to use parameters of different types when using user defined functions in LINQ.

- Fixed a bug for globally replicated accounts where Upsert calls were being directed to read locations instead of write locations.
- Added methods to the IDocumentClient interface that were missing:
  - UpsertAttachmentAsync method that takes mediaStream and options as parameters.
  - CreateAttachmentAsync method that takes options as a parameter.
  - CreateOfferQuery method that takes querySpec as a parameter.
- Unsealed public classes that are exposed in the IDocumentClient interface.

1.8.0

- Added the support for multi-region database accounts.
- Added support for retry on throttled requests.User can customize the number of retries and the max wait time by configuring the ConnectionPolicy.RetryOptions property.
- Added a new IDocumentClient interface that defines the signatures of all DocumenClient properties and methods. As part of this change, also changed extension methods that create IQueryable and IOrderedQueryable to methods on the DocumentClient class itself.
- Added configuration option to set the ServicePoint.ConnectionLimit for a given DocumentDB endpoint Uri. Use ConnectionPolicy.MaxConnectionLimit to change the default value, which is set to 50.
- Deprecated IPartitionResolver and its implementation. Support for IPartitionResolver is now obsolete. It's recommended that you use Partitioned Collections for higher storage and throughput.

1.7.1

- Added an overload to Uri based ExecuteStoredProcedureAsync method that takes RequestOptions as a parameter.

1.7.0

- Added time to live (TTL) support for documents.

1.6.3

- Fixed a bug in Nuget packaging of .NET SDK for packaging it as part of a Azure Cloud Service solution.

1.6.2

- Implemented partitioned collections and user-defined performance levels.

1.5.3

- **[Fixed]** Querying DocumentDB endpoint throws: 'System.Net.Http.HttpRequestException: Error while copying content to a stream.

1.5.2

- Expanded LINQ support including new operators for paging, conditional expressions and range comparison.
  - Take operator to enable SELECT TOP behavior in LINQ.
  - CompareTo operator to enable string range comparisons.
  - Conditional (?) and coalesce operators (??).
- **[Fixed]** ArgumentOutOfRangeException when combining Model projection with Where-In in linq query. #81

1.5.1

- **[Fixed]** If Select is not the last expression the LINQ Provider assumed no projection and produced SELECT * incorrectly. #58

1.5.0

- Implemented Upsert, Added UpsertXXXAsync methods.
- Performance improvements for all requests.
- LINQ Provider support for conditional, coalesce and CompareTo methods for strings.
- **[Fixed]** LINQ provider --> Implement Contains method on List to generate the same SQL as on IEnumerable and Array.
- **[Fixed]** BackoffRetryUtility uses the same HttpRequestMessage again instead of creating a new one on retry.
- **[Obsolete]** UriFactory.CreateCollection --> should now use UriFactory.CreateDocumentCollection.

1.4.1

- **[Fixed]** Localization issues when using non en culture info such as nl-NL etc.

1.4.0

- ID Based Routing
  - New UriFactory helper to assist with constructing ID based resource links.
  - New overloads on DocumentClient to take in URI.
- Added IsValid() and IsValidDetailed() in LINQ for geospatial.
- LINQ Provider support enhanced.
  - **Math** - Abs, Acos, Asin, Atan, Ceiling, Cos, Exp, Floor, Log, Log10, Pow, Round, Sign, Sin, Sqrt, Tan, Truncate.
  - **String** - Concat, Contains, EndsWith, IndexOf, Count, ToLower, TrimStart, Replace, Reverse, TrimEnd, StartsWith, SubString, ToUpper.
  - **Array** - Concat, Contains, Count.
  - **IN** operator.

1.3.0

- Added support for modifying indexing policies.
  - New ReplaceDocumentCollectionAsync method in DocumentClient.
  - New IndexTransformationProgress property in ResourceResponse for tracking percent progress of index policy changes.
  - DocumentCollection.IndexingPolicy is now mutable.
- Added support for spatial indexing and query
  - New Microsoft.Azure.Documents.Spatial namespace for serializing/deserializing spatial types like Point and Polygon.
  - New SpatialIndex class for indexing GeoJSON data stored in DocumentDB.
- **[Fixed]** : Incorrect SQL query generated from linq expression. #38

1.2.0

- Dependency on Newtonsoft.Json v5.0.7.
- Changes to support Order By:
  - LINQ provider support for OrderBy() or OrderByDescending().
  - IndexingPolicy to support Order By.

  NB: Possible breaking change

  If you have existing code that provisions collections with a custom indexing policy, then your existing code will need to be updated to support the new IndexingPolicy class. If you have no custom indexing policy, then this change does not affect you.

1.1.0

- Support for partitioning data by using the new HashPartitionResolver and RangePartitionResolver classes and the IPartitionResolver.
- DataContract serialization.
- Guid support in LINQ provider.
- UDF support in LINQ.

1.0.0

- GA SDK.

Release & Retirement dates

Microsoft provides notification at least **12 months** in advance of retiring an SDK in order to smooth the transition to a newer/supported version. New features and functionality and optimizations are only added to the current SDK, as such it is recommended that you always upgrade to the latest SDK version as early as possible.

After 31 August 2022, Azure Cosmos DB will no longer make bug fixes, add new features, and provide support to versions 1.x of the Azure Cosmos DB .NET or .NET Core SDK for SQL API. If you prefer not to upgrade, requests sent from version 1.x of the SDK will continue to be served by the Azure Cosmos DB service.

VERSION	RELEASE DATE	RETIREMENT DATE
2.12.0	October 7, 2020	---
2.11.6	August 12, 2020	---
2.11.5	August 4, 2020	---
2.11.4	July 30, 2020	---
2.11.3	July 29, 2020	---
2.11.2	July 14, 2020	---
2.11.1	July 1, 2020	---
2.9.2	November 15, 2019	---
2.9.1	November 13, 2019	---
2.9.0	October 30, 2019	---
2.8.1	October 11, 2019	---
2.7.0	September 23, 2019	---
2.6.0	August 30, 2019	---
2.5.1	July 02, 2019	---
2.4.1	June 20, 2019	---

VERSION	RELEASE DATE	RETIREMENT DATE
2.4.0	May 05, 2019	---
2.3.0	April 04, 2019	---
2.2.3	February 11, 2019	---
2.2.2	February 06, 2019	---
2.2.1	December 24, 2018	---
2.2.0	December 07, 2018	---
2.1.3	October 15, 2018	---
2.1.2	October 04, 2018	---
2.1.1	September 27, 2018	---
2.1.0	September 21, 2018	---
2.0.0	September 07, 2018	---
1.22.0	April 19, 2018	August 31, 2022
1.21.1	March 09, 2018	August 31, 2022
1.20.2	February 21, 2018	August 31, 2022
1.20.1	February 05, 2018	August 31, 2022
1.19.1	November 16, 2017	August 31, 2022
1.19.0	November 10, 2017	August 31, 2022
1.18.1	November 07, 2017	August 31, 2022
1.18.0	October 17, 2017	August 31, 2022
1.17.0	August 10, 2017	August 31, 2022
1.16.1	August 07, 2017	August 31, 2022
1.16.0	August 02, 2017	August 31, 2022
1.15.0	June 30, 2017	August 31, 2022
1.14.1	May 23, 2017	August 31, 2022
1.14.0	May 10, 2017	August 31, 2022

VERSION	RELEASE DATE	RETIREMENT DATE
1.13.4	May 09, 2017	August 31, 2022
1.13.3	May 06, 2017	August 31, 2022
1.13.2	April 19, 2017	August 31, 2022
1.13.1	March 29, 2017	August 31, 2022
1.13.0	March 24, 2017	August 31, 2022
1.12.1	March 14, 2017	August 31, 2022
1.12.0	February 15, 2017	August 31, 2022
1.11.4	February 06, 2017	August 31, 2022
1.11.3	January 26, 2017	August 31, 2022
1.11.1	December 21, 2016	August 31, 2022
1.11.0	December 08, 2016	August 31, 2022
1.10.0	September 27, 2016	August 31, 2022
1.9.5	September 01, 2016	August 31, 2022
1.9.4	August 24, 2016	August 31, 2022
1.9.3	August 15, 2016	August 31, 2022
1.9.2	July 23, 2016	August 31, 2022
1.8.0	June 14, 2016	August 31, 2022
1.7.1	May 06, 2016	August 31, 2022
1.7.0	April 26, 2016	August 31, 2022
1.6.3	April 08, 2016	August 31, 2022
1.6.2	March 29, 2016	August 31, 2022
1.5.3	February 19, 2016	August 31, 2022
1.5.2	December 14, 2015	August 31, 2022
1.5.1	November 23, 2015	August 31, 2022
1.5.0	October 05, 2015	August 31, 2022

VERSION	RELEASE DATE	RETIREMENT DATE
1.4.1	August 25, 2015	August 31, 2022
1.4.0	August 13, 2015	August 31, 2022
1.3.0	August 05, 2015	August 31, 2022
1.2.0	July 06, 2015	August 31, 2022
1.1.0	April 30, 2015	August 31, 2022
1.0.0	April 08, 2015	August 31, 2022

See Also

To learn more about Cosmos DB, see Microsoft Azure Cosmos DB service page.

.NET Change Feed Processor SDK: Download and release notes

11/2/2020 • 9 minutes to read • Edit Online

APPLIES TO: SQL API

SDK download	NuGet
API documentation	Change Feed Processor library API reference documentation
Get started	Get started with the Change Feed Processor .NET SDK
Current supported framework	Microsoft .NET Framework 4.5 Microsoft .NET Core

> **NOTE**
> If you are using change feed processor, please see the latest version 3.x of the .NET SDK, which has change feed built into the SDK.

Release notes

v2 builds

2.3.2

- Added lease store compatibility with [V3 SDK that enables hot migration paths. An application can migrate to V3 SDK and migrate back to the Change Feed processor library without losing any state.

2.3.1

- Corrected a case when `FeedProcessing.ChangeFeedObserverCloseReason.Unknown` close reason was sent to `FeedProcessing.IChangeFeedObserver.CloseAsync` if the partition cannot be found or if the target replica is not up to date up with the read session. In these cases `FeedProcessing.ChangeFeedObserverCloseReason.ResourceGone` and `FeedProcessing.ChangeFeedObserverCloseReason.ReadSessionNotAvailable` close reasons are now used.
- Added a new close reason `FeedProcessing.ChangeFeedObserverCloseReason.ReadSessionNotAvailable` that is sent to close the change feed observer when the target replica is not up to date up with the read session.

2.3.0

- Added a new method `ChangeFeedProcessorBuilder.WithCheckpointPartitionProcessorFactory` and corresponding public interface `ICheckpointPartitionProcessorFactory`. This allows an implementation of the `IPartitionProcessor` interface to use built-in checkpointing mechanism. The new factory is similar to the existing `IPartitionProcessorFactory`, except that its `Create` method also takes the `ILeaseCheckpointer` parameter.
- Only one of the two methods, either `ChangeFeedProcessorBuilder.WithPartitionProcessorFactory` or `ChangeFeedProcessorBuilder.WithCheckpointPartitionProcessorFactory`, can be used for the same `ChangeFeedProcessorBuilder` instance.

2.2.8

- Stability and diagnosability improvements:
  - Added support to detect reading change feed taking long time. When it takes longer than the value specified by the `ChangeFeedProcessorOptions.ChangeFeedTimeout` property, the following steps are taken:
    - The operation to read change feed on the problematic partition is aborted.
    - The change feed processor instance drops ownership of the problematic lease. The dropped lease will be picked up during the next lease acquire step that will be done by the same or different change feed processor instance. This way, reading change feed will start over.
    - An issue is reported to the health monitor. The default heath monitor sends all reported issues to trace log.
  - Added a new public property: `ChangeFeedProcessorOptions.ChangeFeedTimeout`. The default value of this property is 10 mins.
  - Added a new public enum value: `Monitoring.MonitoredOperation.ReadChangeFeed`. When the value of `HealthMonitoringRecord.Operation` is set to `Monitoring.MonitoredOperation.ReadChangeFeed`, it indicates the health issue is related to reading change feed.

2.2.7

- Improved load-balancing strategy for scenario when getting all leases takes longer than lease expiration interval, for example, due to network issues:
  - In this scenario load-balancing algorithm used to falsely consider leases as expired, causing stealing leases from active owners. This could trigger unnecessary rebalancing many leases.
  - This issue is fixed in this release by avoiding retry on conflict while acquiring expired lease which owner hasn't changed and postponing acquiring expired lease to next load-balancing iteration.

2.2.6

- Improved handling of Observer exceptions.
- Richer information on Observer errors:
  - When an Observer is closed due to an exception thrown by Observer's ProcessChangesAsync, the CloseAsync will now receive the reason parameter set to ChangeFeedObserverCloseReason.ObserverError.
  - Added traces to identify errors within user code in an Observer.

2.2.5

- Added support for handling split in collections that use shared database throughput.
  - This release fixes an issue that may occur during split in collections using shared database throughput when split result into partition rebalancing with only one child partition key range created, rather than two. When this happens, Change Feed Processor may get stuck deleting the lease for old partition key range and not creating new leases. The issue is fixed in this release.

2.2.4

- Added new property ChangeFeedProcessorOptions.StartContinuation to support starting change feed from request continuation token. This is only used when lease collection is empty or a lease does not have ContinuationToken set. For leases in lease collection that have ContinuationToken set, the ContinuationToken is used and ChangeFeedProcessorOptions.StartContinuation is ignored.

2.2.3

- Added support for using custom store to persist continuation tokens per partition.
  - For example, a custom lease store can be Azure Cosmos DB lease collection partitioned in any custom way.
  - Custom lease stores can use new extensibility point ChangeFeedProcessorBuilder.WithLeaseStoreManager(ILeaseStoreManager) and ILeaseStoreManager

public interface.
  - Refactored the ILeaseManager interface into multiple role interfaces.
- Minor breaking change: removed extensibility point ChangeFeedProcessorBuilder.WithLeaseManager(ILeaseManager), use ChangeFeedProcessorBuilder.WithLeaseStoreManager(ILeaseStoreManager) instead.

2.2.2

- This release fixes an issue that occurs during processing a split in monitored collection and using a partitioned lease collection. When processing a lease for split partition, the lease corresponding to that partition may not be deleted. The issue is fixed in this release.

2.2.1

- Fixed Estimator calculation for accounts with multiple write regions and new Session Token format.

2.2.0

- Added support for partitioned lease collections. The partition key must be defined as /id.
- Minor breaking change: the methods of the IChangeFeedDocumentClient interface and the ChangeFeedDocumentClient class were changed to include RequestOptions and CancellationToken parameters. IChangeFeedDocumentClient is an advanced extensibility point that allows you to provide custom implementation of the Document Client to use with Change Feed Processor, for example, decorate DocumentClient and intercept all calls to it to do extra tracing, error handling, etc. With this update, the code that implement IChangeFeedDocumentClient will need to be changed to include new parameters in the implementation.
- Minor diagnostics improvements.

2.1.0

- Added new API, Task<IReadOnlyList<RemainingPartitionWork>> IRemainingWorkEstimator.GetEstimatedRemainingWorkPerPartitionAsync(). This can be used to get estimated work for each partition.
- Supports Microsoft.Azure.DocumentDB SDK 2.0. Requires Microsoft.Azure.DocumentDB 2.0 or later.

2.0.6

- Added ChangeFeedEventHost.HostName public property for compatibility with v1.

2.0.5

- Fixed a race condition that occurs during partition split. The race condition may lead to acquiring lease and immediately losing it during partition split and causing contention. The race condition issue is fixed with this release.

2.0.4

- GA SDK

2.0.3-prerelease

- Fixed the following issues:
  - When partition split happens, there could be duplicate processing of documents modified before the split.
  - The GetEstimatedRemainingWork API returned 0 when no leases were present in the lease collection.
- The following exceptions are made public. Extensions that implement IPartitionProcessor can throw these exceptions.
  - Microsoft.Azure.Documents.ChangeFeedProcessor.Exceptions.LeaseLostException.
  - Microsoft.Azure.Documents.ChangeFeedProcessor.Exceptions.PartitionException.
  - Microsoft.Azure.Documents.ChangeFeedProcessor.Exceptions.PartitionNotFoundException.

    - Microsoft.Azure.Documents.ChangeFeedProcessor.Exceptions.PartitionSplitException.

2.0.2-prerelease

- Minor API changes:
    - Removed ChangeFeedProcessorOptions.IsAutoCheckpointEnabled that was marked as obsolete.

2.0.1-prerelease

- Stability improvements:
    - Better handling of lease store initialization. When lease store is empty, only one instance of processor can initialize it, the others will wait.
    - More stable/efficient lease renewal/release. Renewing and releasing a lease one partition is independent from renewing others. In v1 that was done sequentially for all partitions.
- New v2 API:
    - Builder pattern for flexible construction of the processor: the ChangeFeedProcessorBuilder class.
        - Can take any combination of parameters.
        - Can take DocumentClient instance for monitoring and/or lease collection (not available in v1).
    - IChangeFeedObserver.ProcessChangesAsync now takes CancellationToken.
    - IRemainingWorkEstimator - the remaining work estimator can be used separately from the processor.
    - New extensibility points:
        - IPartitionLoadBalancingStrategy - for custom load-balancing of partitions between instances of the processor.
        - ILease, ILeaseManager - for custom lease management.
        - IPartitionProcessor - for custom processing changes on a partition.
- Logging - uses LibLog library.
- 100% backward compatible with v1 API.
- New code base.
- Compatible with SQL .NET SDK versions 1.21.1 and above.

v1 builds

1.3.3

- Added more logging.
- Fixed a DocumentClient leak when calling the pending work estimation multiple times.

1.3.2

- Fixes in the pending work estimation.

1.3.1

- Stability improvements.
    - Fix for handling canceled tasks issue that might lead to stopped observers on some partitions.
- Support for manual checkpointing.
- Compatible with SQL .NET SDK versions 1.21 and above.

1.2.0

- Adds support for .NET Standard 2.0. The package now supports `netstandard2.0` and `net451` framework monikers.
- Compatible with SQL .NET SDK versions 1.17.0 and above.
- Compatible with SQL .NET Core SDK versions 1.5.1 and above.

1.1.1

- Fixes an issue with the calculation of the estimate of remaining work when the Change Feed was empty or no

work was pending.

- Compatible with SQL .NET SDK versions 1.13.2 and above.

1.1.0

- Added a method to obtain an estimate of remaining work to be processed in the Change Feed.
- Compatible with SQL .NET SDK versions 1.13.2 and above.

1.0.0

- GA SDK
- Compatible with SQL .NET SDK versions 1.14.1 and below.

Release & Retirement dates

Microsoft will provide notification at least **12 months** in advance of retiring an SDK in order to smooth the transition to a newer/supported version. New features and functionality and optimizations are only added to the current SDK, as such it is recommended that you always upgrade to the latest SDK version as early as possible.

> **WARNING**
> After 31 August 2022, Azure Cosmos DB will no longer make bug fixes, add new features, and provide support to versions 1.x of the Azure Cosmos DB .NET or .NET Core SDK for SQL API. If you prefer not to upgrade, requests sent from version 1.x of the SDK will continue to be served by the Azure Cosmos DB service.

VERSION	RELEASE DATE	RETIREMENT DATE
2.3.2	August 11, 2020	---
2.3.1	July 30, 2020	---
2.3.0	April 2, 2020	---
2.2.8	October 28, 2019	---
2.2.7	May 14, 2019	---
2.2.6	January 29, 2019	---
2.2.5	December 13, 2018	---
2.2.4	November 29, 2018	---
2.2.3	November 19, 2018	---
2.2.2	October 31, 2018	---
2.2.1	October 24, 2018	---
1.3.3	May 08, 2018	---
1.3.2	April 18, 2018	---

VERSION	RELEASE DATE	RETIREMENT DATE
1.3.1	March 13, 2018	---
1.2.0	October 31, 2017	---
1.1.1	August 29, 2017	---
1.1.0	August 13, 2017	---
1.0.0	July 07, 2017	---

FAQ

How will I be notified of the retiring SDK?

Microsoft will provide 12 month's advance notice before the end of support of the retiring SDK to facilitate a smooth transition to a supported SDK. We'll notify you through various communication channels: the Azure portal, Azure updates, and direct communication to assigned service administrators.

Can I author applications by using a to-be-retired Azure Cosmos DB SDK during the 12-month period?

Yes, you'll be able to author, deploy, and modify applications by using the to-be-retired Azure Cosmos DB SDK during the 12-month notice period. We recommend that you migrate to a newer supported version of the Azure Cosmos DB SDK during the 12-month notice period, as appropriate.

After the retirement date, what happens to applications that use the unsupported Azure Cosmos DB SDK?

After the retirement date, Azure Cosmos DB will no longer make bug fixes, add new features, or provide support to the retired SDK versions. If you prefer not to upgrade, requests sent from the retired versions of the SDK will continue to be served by the Azure Cosmos DB service.

Which SDK versions will have the latest features and updates?

New features and updates will be added only to the latest minor version of the latest supported major SDK version. We recommend that you always use the latest version to take advantage of new features, performance improvements, and bug fixes. If you're using an old, non-retired version of the SDK, your requests to Azure Cosmos DB will still function, but you won't have access to any new capabilities.

What should I do if I can't update my application before a cutoff date?

We recommend that you upgrade to the latest SDK as early as possible. After an SDK is tagged for retirement, you'll have 12 months to update your application. If you're not able to update by the retirement date, requests sent from the retired versions of the SDK will continue to be served by Azure Cosmos DB, so your running applications will continue to function. But Azure Cosmos DB will no longer make bug fixes, add new features, or provide support to the retired SDK versions.

If you have a support plan and require technical support, contact us by filing a support ticket.

See also

To learn more about Cosmos DB, see Microsoft Azure Cosmos DB service page.

.NET bulk executor library: Download information

11/2/2020 • 3 minutes to read • Edit Online

APPLIES TO: SQL API

Description	The .NET bulk executor library allows client applications to perform bulk operations on Azure Cosmos DB accounts. This library provides BulkImport, BulkUpdate, and BulkDelete namespaces. The BulkImport module can bulk ingest documents in an optimized way such that the throughput provisioned for a collection is consumed to its maximum extent. The BulkUpdate module can bulk update existing data in Azure Cosmos containers as patches. The BulkDelete module can bulk delete documents in an optimized way such that the throughput provisioned for a collection is consumed to its maximum extent.
SDK download	NuGet
Bulk executor library in GitHub	GitHub
API documentation	.NET API reference documentation
Get started	Get started with the bulk executor library .NET SDK
Current supported framework	Microsoft .NET Framework 4.5.2, 4.6.1 and .NET Standard 2.0

> **NOTE**
> If you are using bulk executor, please see the latest version 3.x of the .NET SDK, which has bulk executor built into the SDK.

Release notes

2.4.1-preview

- Fixed TotalElapsedTime in the response of BulkDelete to correctly measure the total time including any retries.

2.4.0-preview

- Changed SDK dependency to >= 2.5.1

2.3.0-preview2

- Added support for graph bulk executor to accept ttl on vertices and edges

2.2.0-preview2

- Fixed an issue, which caused exceptions during elastic scaling of Azure Cosmos DB when running in Gateway mode. This fix makes it functionally equivalent to 1.4.1 release.

2.1.0-preview2

- Added BulkDelete support for SQL API accounts to accept partition key, document ID tuples to delete. This

change makes it functionally equivalent to 1.4.0 release.

2.0.0-preview2

- Including MongoBulkExecutor to support .NET Standard 2.0. This feature makes it functionally equivalent to 1.3.0 release, with the addition of supporting .NET Standard 2.0 as the target framework.

2.0.0-preview

- Added .NET Standard 2.0 as one of the supported target frameworks to make the bulk executor library work with .NET Core applications.

1.8.9

- Fixed an issue with BulkDeleteAsync when values with escaped quotes were passed as input parameters.

1.8.8

- Fixed an issue on MongoBulkExecutor that was increasing the document size unexpectedly by adding padding and in some cases, going over the maximum document size limit.

1.8.7

- Fixed an issue with BulkDeleteAsync when the Collection has nested partition key paths.

1.8.6

- MongoBulkExecutor now implements IDisposable and it's expected to be disposed after used.

1.8.5

- Removed lock on SDK version. Package is now dependent on SDK >= 2.5.1.

1.8.4

- Fixed handling of identifiers when calling BulkImport with a list of POCO objects with numeric values.

1.8.3

- Fixed TotalElapsedTime in the response of BulkDelete to correctly measure the total time including any retries.

1.8.2

- Fixed high CPU consumption on certain scenarios.
- Tracing now uses TraceSource. Users can define listeners for the `BulkExecutorTrace` source.
- Fixed a rare scenario that could cause a lock when sending documents near 2Mb of size.

1.6.0

- Updated the bulk executor to now use the latest version of the Azure Cosmos DB .NET SDK (2.4.0)

1.5.0

- Added support for graph bulk executor to accept ttl on vertices and edges

1.4.1

- Fixed an issue, which caused exceptions during elastic scaling of Azure Cosmos DB when running in Gateway mode.

1.4.0

- Added BulkDelete support for SQL API accounts to accept partition key, document ID tuples to delete.

1.3.0

- Fixed an issue, which caused a formatting issue in the user agent used by bulk executor.

1.2.0

- Made improvement to bulk executor import and update APIs to transparently adapt to elastic scaling of Cosmos container when storage exceeds current capacity without throwing exceptions.

1.1.2

- Bumped up the DocumentDB .NET SDK dependency to version 2.1.3.

1.1.1

- Fixed an issue, which caused bulk executor to throw JSRT error while importing to fixed collections.

1.1.0

- Added support for BulkDelete operation for Azure Cosmos DB SQL API accounts.
- Added support for BulkImport operation for accounts with Azure Cosmos DB's API for MongoDB.
- Bumped up the DocumentDB .NET SDK dependency to version 2.0.0.

1.0.2

- Added support for BulkImport operation for Azure Cosmos DB Gremlin API accounts.

1.0.1

- Minor bug fix to the BulkImport operation for Azure Cosmos DB SQL API accounts.

1.0.0

- Added support for BulkImport and BulkUpdate operations for Azure Cosmos DB SQL API accounts.

Next steps

To learn about the bulk executor Java library, see the following article:

Java bulk executor library SDK and release information

Azure Cosmos DB Java SDK v4 for Core (SQL) API: release notes and resources

11/2/2020 • 10 minutes to read • Edit Online

APPLIES TO: SQL API

The Azure Cosmos DB Java SDK v4 for Core (SQL) combines an Async API and a Sync API into one Maven artifact. The v4 SDK brings enhanced performance, new API features, and Async support based on Project Reactor and the Netty library. Users can expect improved performance with Azure Cosmos DB Java SDK v4 versus the Azure Cosmos DB Async Java SDK v2 and the Azure Cosmos DB Sync Java SDK v2.

> **IMPORTANT**
>
> These Release Notes are for Azure Cosmos DB Java SDK v4 only. If you are currently using an older version than v4, see the Migrate to Azure Cosmos DB Java SDK v4 guide for help upgrading to v4.
>
> Here are three steps to get going fast!
>
> 1. Install the minimum supported Java runtime, JDK 8 so you can use the SDK.
> 2. Work through the Quickstart Guide for Azure Cosmos DB Java SDK v4 which gets you access to the Maven artifact and walks through basic Azure Cosmos DB requests.
> 3. Read the Azure Cosmos DB Java SDK v4 performance tips and troubleshooting guides to optimize the SDK for your application.
>
> The Azure Cosmos DB workshops and labs are another great resource for learning how to use Azure Cosmos DB Java SDK v4!

Helpful content

CONTENT	LINK
SDK download	Maven
API documentation	Java API reference documentation
Contribute to SDK	Azure SDK for Java Central Repo on GitHub
Get started	Quickstart: Build a Java app to manage Azure Cosmos DB SQL API data GitHub repo with quickstart code
Basic code samples	Azure Cosmos DB: Java examples for the SQL API GitHub repo with sample code
Console app with Change Feed	Change feed - Java SDK v4 sample GitHub repo with sample code
Web app sample	Build a web app with Java SDK v4 GitHub repo with sample code

CONTENT	LINK
Performance tips	Performance tips for Java SDK v4
Troubleshooting	Troubleshoot Java SDK v4
Migrate to v4 from an older SDK	Migrate to Java V4 SDK
Minimum supported runtime	JDK 8
Azure Cosmos DB workshops and labs	Cosmos DB workshops home page

Release History

4.9.0-beta.1 (Unreleased)

4.8.0 (2020-10-27)

New Features

- Added `contentResponseOnWriteEnabled` feature to `CosmosItemRequestOptions`.

Key Bug Fixes

- Fixed an issue which may affect query behaviour when resuming from a continuation token.

4.7.1 (2020-10-21)

Key Bug Fixes

- Improved the 449 retry policy to force back-off on initial retry and start with shorter back-offs.

4.7.0 (2020-10-17) NOTE: WE STRONGLY RECOMMEND OUR CUSTOMERS TO USE VERSION 4.7.0 AND ABOVE

New Features

- Added Beta for transactional batches.

Key Bug Fixes

- Fixed an error parsing query metrics on locales with ',' as floating-point delimiter.
- Stopped excessive regional fail-overs when retrieving responses with invalid json from Gateway.
- Fixed an error resulting in certain queries unnecessarily being expected in the Gateway even when using Direct transport.
- Reduced logging noise level by handling SSLException on channel closure.
- Improved efficiency of retry logic for "404 - ReadSession not available" errors.

4.6.0 (2020-09-30)

New Features

- Added new API to support AAD role-based access control in Cosmos. This is a preview feature which needs to be enabled at the account settings.
- Added handler API(beta) to `CosmosPagedFlux` / `CosmosPagedIterable` to be invoked on every response.

4.5.2 (2020-09-29)

Key Bug Fixes

- Increased robustness of query execution and fetching metadata cache in case of intermittent connectivity issues.

4.5.1 (2020-09-25)

Key Bug Fixes

- Added preview implementation for ChangeFeedProcessor which allows for a more detailed view of the current state.
- Fixed Multiple partition supervisor tasks running simultaneously if leaseAcquireInterval is smaller than leaseRenewInterval.
- Improved Diagnostics for Rntbd connectivity.
- Stopped onError Dropped events from leaking into default reactor hook.

4.5.0 (2020-09-16)

New Features

- Increased robustness of the Rntbd stack in case of intermittent connectivity issues.
- Improved latency in case of intermittent connectivity issues to individual backend replicas for multi-region accounts avoiding initiation of unnecessary regional fail-overs.

4.4.0 (2020-09-12)

Key Bug Fixes

- Fixed RequestTimeoutException when enabling `netty-tcnative-boringssl` dependency.
- Fixed memory leak issue on `Delete` operations in `GATEWAY` mode.
- Fixed a leak in `CosmosClient` instantiation when endpoint uri is invalid.
- Improved `CPU History` diagnostics.

4.4.0-beta.1 (2020-08-27)

New Features

- Added new API to efficiently load many documents (via list of pk/id pairs or all documents for a set of pk values).
- Added new `deleteItem` API.
- Enabled query metrics by default.

Key Bug Fixes

- Fixed NPE in `GatewayAddressCache`.
- Fixing query metric issue for zero item response.
- Improved performance (reduced CPU usage) for address parsing and Master-Key authentication.

4.3.2-beta.2 (2020-08-17)

Key Bug Fixes

- No changes from previous version, releasing for compatibility issues with spring data modules.

4.3.2-beta.1 (2020-08-14)

Key Bug Fixes

- Fixed issue in RntbdServiceEndpoint to avoid early closure of an unused TCP connection.

4.3.1 (2020-08-13)

Key Bug Fixes

- Fixed issue with `GROUP BY` query, where it was returning only one page.
- Fixed user agent string format to comply with central SDK guidelines.
- Enhanced diagnostics information to include query plan diagnostics.

4.3.0 (2020-07-29)

New Features

- Updated reactor-core library version to `3.3.8.RELEASE`.
- Updated reactor-netty library version to `0.9.10.RELEASE`.
- Updated netty library version to `4.1.51.Final`.
- Added new overload APIs for `upsertItem` with `partitionKey`.

- Added open telemetry tracing support.

Key Bug Fixes

- Fixed issue where SSLException gets thrown in case of cancellation of requests in GATEWAY mode.
- Fixed resource throttle retry policy on stored procedures execution.
- Fixed issue where SDK hangs in log level DEBUG mode.
- Fixed periodic spikes in latency in Direct mode.
- Fixed high client initialization time issue.
- Fixed http proxy bug when customizing client with direct mode and gateway mode.
- Fixed potential NPE in users passes null options.
- Added timeUnit to `requestLatency` in diagnostics string.
- Removed duplicate uri string from diagnostics string.
- Fixed diagnostics string in proper JSON format for point operations.
- Fixed issue with `.single()` operator causing the reactor chain to blow up in case of Not Found exception.

4.2.0 (2020-07-14)

New Features

- Added script logging enabled API to `CosmosStoredProcedureRequestOptions`.
- Updated `DirectConnectionConfig` default `idleEndpointTimeout` to 1h and default `connectTimeout` to 5s.

Key Bug Fixes

- Fixed issue where `GatewayConnectionConfig` `idleConnectionTimeout` was overriding `DirectConnectionConfig` `idleConnectionTimeout`.
- Fixed `responseContinuationTokenLimitInKb` get and set APIs in `CosmosQueryRequestOptions`.
- Fixed issue in query and change feed when recreating the collection with same name.
- Fixed issue with top query throwing ClassCastException.
- Fixed issue with order by query throwing NullPointerException.
- Fixed issue in handling of cancelled requests in direct mode causing reactor `onErrorDropped` being called.

4.1.0 (2020-06-25)

New Features

- Added support for `GROUP BY` query.
- Increased the default value of maxConnectionsPerEndpoint to 130 in DirectConnectionConfig.
- Increased the default value of maxRequestsPerConnection to 30 in DirectConnectionConfig.

Key Bug Fixes

- Fixed issues with order by query returning duplicate results when resuming by using continuation token.
- Fixed issues with value query returning null values for nested object.
- Fixed null pointer exception on request manager in RntbdClientChannelPool.

4.0.1 (2020-06-10)

New Features

- Renamed `QueryRequestOptions` to `CosmosQueryRequestOptions`.
- Updated `ChangeFeedProcessorBuilder` to builder pattern.
- Updated `CosmosPermissionProperties` with new container name and child resources APIs.

Key Bug Fixes

- Fixed ConnectionPolicy `toString()` Null Pointer Exception.

4.0.1-beta.4 (2020-06-03)

New Features

- Added more samples & enriched docs to `CosmosClientBuilder`.
- Updated `CosmosDatabase` & `CosmosContainer` APIs with throughputProperties for autoscale/autopilot support.
- Renamed `CosmosClientException` to `CosmosException`.
- Replaced `AccessCondition` & `AccessConditionType` by `ifMatchETag()` & `ifNoneMatchETag()` APIs.
- Merged all `Cosmos*AsyncResponse` & `CosmosResponse` types to a single `CosmosResponse` type.
- Renamed `CosmosResponseDiagnostics` to `CosmosDiagnostics`.
- Wrapped `FeedResponseDiagnostics` in `CosmosDiagnostics`.
- Removed `jackson` dependency from azure-cosmos & relying on azure-core.
- Replaced `CosmosKeyCredential` with `AzureKeyCredential` type.
- Added `ProxyOptions` APIs to `GatewayConnectionConfig`.
- Updated SDK to use `Instant` type instead of `OffsetDateTime`.
- Added new enum type `OperationKind`.
- Renamed `FeedOptions` to `QueryRequestOptions`.
- Added `getETag()` & `getTimestamp()` APIs to `Cosmos*Properties` types.
- Added `userAgent` information in `CosmosException` & `CosmosDiagnostics`.
- Updated new line character in `Diagnostics` to System new line character.
- Removed `readAll*` APIs, use query select all APIs instead.
- Added `ChangeFeedProcessor` estimate lag API.

Key Bug Fixes

- Fixed issue with parsing of query results in case of Value order by queries.

4.0.1-beta.3 (2020-05-15)

New Features

- Added autoscale/autopilot throughput provisioning support to SDK.
- Replaced `ConnectionPolicy` with new connection configs. Exposed `DirectConnectionConfig` & `GatewayConnectionConfig` APIs through `CosmosClientBuilder` for Direct & Gateway mode connection configurations.
- Moved `JsonSerializable` & `Resource` to implementation package.
- Added `contentResponseOnWriteEnabled` API to CosmosClientBuilder which disables full response content on write operations.
- Exposed `getETag()` APIs on response types.
- Moved `CosmosAuthorizationTokenResolver` to implementation.
- Renamed `preferredLocations` & `multipleWriteLocations` API to `preferredRegions` & `multipleWriteRegions`.
- Updated `reactor-core` to 3.3.5.RELEASE, `reactor-netty` to 0.9.7.RELEASE & `netty` to 4.1.49.Final versions.
- Added support for `analyticalStoreTimeToLive` in SDK.

Key Bug Fixes

- Fixed socket leak issues with Direct TCP client.
- Fixed `orderByQuery` with continuation token bug.

4.0.1-beta.2 (2020-04-21)

New Features

- `CosmosClientException` extends `AzureException`.
- Removed `maxItemCount` & `requestContinuationToken` APIs from `FeedOptions` instead using `byPage()` APIs from `CosmosPagedFlux` & `CosmosPagedIterable`.

- Introduced `CosmosPermissionProperties` on public surface for `Permission` APIs.
- Removed `SqlParameterList` type & replaced with `List`
- Fixed multiple memory leaks in Direct TCP client.
- Added support for `DISTINCT` queries.
- Removed external dependencies on `fasterxml.uuid, guava, commons-io, commons-collection4, commons-text`.
- Moved `CosmosPagedFlux` & `CosmosPagedIterable` to `utils` package.
- Updated netty to 4.1.45.Final & project reactor to 3.3.3 version.
- Updated public rest contracts to `Final` classes.
- Added support for advanced Diagnostics for point operations.

Key Bug Fixes

- `ChangeFeedProcessor` bug fix for handling partition splits & when partition not found.
- `ChangeFeedProcessor` bug fix when synchronizing lease updates across different threads.

4.0.1-beta.1 (2020-03-10)

New Features

- Updated package to `com.azure.cosmos`
- Added `models` package for model / rest contracts
- Added `utils` package for `CosmosPagedFlux` & `CosmosPagedIterable` types.
- Updated public APIs to use `Duration` across the SDK.
- Added all rest contracts to `models` package.
- `RetryOptions` renamed to `ThrottlingRetryOptions`.
- Added `CosmosPagedFlux` & `CosmosPagedIterable` pagination types for query APIs.
- Added support for sharing TransportClient across multiple instances of CosmosClients using a new API in the `CosmosClientBuilder#connectionSharingAcrossClientsEnabled(true)`
- Query Optimizations by removing double serialization / deserialization.
- Response Headers optimizations by removing unnecessary copying back and forth.
- Optimized `ByteBuffer` serialization / deserialization by removing intermediate String instantiations.

Key Bug Fixes

- Fixed race condition causing `ArrayIndexOutOfBound` exception in StoreReader

FAQ

How will I be notified of the retiring SDK?

Microsoft will provide 12 month's advance notice before the end of support of the retiring SDK to facilitate a smooth transition to a supported SDK. We'll notify you through various communication channels: the Azure portal, Azure updates, and direct communication to assigned service administrators.

Can I author applications by using a to-be-retired Azure Cosmos DB SDK during the 12-month period?

Yes, you'll be able to author, deploy, and modify applications by using the to-be-retired Azure Cosmos DB SDK during the 12-month notice period. We recommend that you migrate to a newer supported version of the Azure Cosmos DB SDK during the 12-month notice period, as appropriate.

After the retirement date, what happens to applications that use the unsupported Azure Cosmos DB SDK?

After the retirement date, Azure Cosmos DB will no longer make bug fixes, add new features, or provide support to the retired SDK versions. If you prefer not to upgrade, requests sent from the retired versions of

the SDK will continue to be served by the Azure Cosmos DB service.

Which SDK versions will have the latest features and updates?

New features and updates will be added only to the latest minor version of the latest supported major SDK version. We recommend that you always use the latest version to take advantage of new features, performance improvements, and bug fixes. If you're using an old, non-retired version of the SDK, your requests to Azure Cosmos DB will still function, but you won't have access to any new capabilities.

What should I do if I can't update my application before a cutoff date?

We recommend that you upgrade to the latest SDK as early as possible. After an SDK is tagged for retirement, you'll have 12 months to update your application. If you're not able to update by the retirement date, requests sent from the retired versions of the SDK will continue to be served by Azure Cosmos DB, so your running applications will continue to function. But Azure Cosmos DB will no longer make bug fixes, add new features, or provide support to the retired SDK versions.

If you have a support plan and require technical support, contact us by filing a support ticket.

Next steps

To learn more about Cosmos DB, see Microsoft Azure Cosmos DB service page.

Azure Cosmos DB Async Java SDK for SQL API: Release notes and resources

12/4/2020 • 8 minutes to read • Edit Online

APPLIES TO: SQL API

The SQL API Async Java SDK differs from the SQL API Java SDK by providing asynchronous operations with support of the Netty library. The pre-existing SQL API Java SDK does not support asynchronous operations.

> **IMPORTANT**
>
> This is *not* the latest Java SDK for Azure Cosmos DB! Consider using Azure Cosmos DB Java SDK v4 for your project. To upgrade, follow the instructions in the Migrate to Azure Cosmos DB Java SDK v4 guide and the Reactor vs RxJava guide.

SDK Download	Maven
API documentation	Java API reference documentation
Contribute to SDK	GitHub
Get started	Get started with the Async Java SDK
Code sample	GitHub
Performance tips	GitHub readme
Minimum supported runtime	JDK 8

2.6.11 - 2020-09-15

- Added http request time out cap for avoiding region failover for high latency request.(#360)

2.6.10 - 2020-08-31

- Fixes region fail over issue for direct tcp calls, direct tcp calls network failure will be retried in other available region but it wont mark the current region unavailable, and subsequent tcp requests will still go to current region.(#351)

2.6.9 - 2020-06-15

- Fixed duplicate results on order by when resuming from continuation token (#341)

2.6.8 - 2020-05-11

- Fixes an issue due to which the order by results when resuming from continuation token might contain duplicates/missing documents (#332)

2.6.7 - 2020-04-21

- Fixed multiple memory leak in direct TCP transport client (#326)

2.6.6 - 2020-03-12

- Made DocumentClientException thread safe (#323)

2.6.5 - 2020-01-31

- Fixed an issue where continuation token is not being set properly when trying to resume cross partition queries(#312)
- Moving deserialization out of netty threads (#315)
- Fixed dont retry writes on forbidden (#307)

2.6.4 - 2020-01-02

- Fixed a bug where SDK was retrying on writes on network failure(#301)
- Add support for specifying default Direct TCP options using System properties (#299)
- Improved diagnostics in Direct TCP transport client (#287)
- Bumped netty.version to 4.1.42.Final and netty-tcnative.version to 2.0.26.Final to address a Direct TCP SSL issue (#274)

2.6.3 - 2019-10-23

- Addressed status code reporting errors that undermined retry policy:
  - `RequestRateTooLargeException.getStatusCode` now correctly returns `TOO_MANY_REQUESTS`.
  - `ServiceUnavailableException.getStatusCode` now correctly returns `SERVICE_UNAVAILABLE`.
  - `DocumentClientExceptionTest` in commons and direct-impl now verify that the correct status code is returned for all `DocumentClientException` subtypes.

2.6.2 - 2019-10-05

- Fixed query failure when setting MaxItemCount to -1 (#261).
- Fixed a NPE bug on Partitoin split (#267).
- Improved logging in Direct mode.

2.6.1 - 2019-09-04

- Multimaster regional failover fixes for query logic (#245)
- jackson-databind 2.9.9.3 update (#244)
- Direct TCP fixes (memory, perf) & metrics interface (#242)

2.6.0 - 2019-08-01

- Retrieving query info using new query plan retriever (#111)
- Offset limit query support (#234)

2.5.1 - 2019-07-31

- Fixing executeStoredProcedureInternal to use client retry policy (#210)

2.4.6 - 2019-07-19

- memory improvements

2.5.0 - 2019-06-25

- TCP transport enabled by default
- TCP transport improvements
- Lots of testing improvements

2.4.5 - 2019-05-06

- Added support for Hash v2 (#96)
- Open source direct connectivity implementation (#94)
- Added Support for Diagnostics String

2.4.4 - 2019-06-25

- misc fixes

2.4.3 - 2019-03-05

- Fixed resource leak issue on closing client

2.4.2 - 2019-03-01

- Fixed bugs in continuation token support for cross partition queries

2.4.1 - 2019-02-20

- Fixed some bugs in Direct mode.
- Improved logging in Direct mode.
- Improved connection management.

2.4.0 - 2019-02-08

- Direct GA.
- Added support for QueryMetrics.
- Changed the APIs accepting java.util.Collection for which order is important to accept java.util.List instead. Now ConnectionPolicy#getPreferredLocations(), JsonSerialization, and PartitionKey(.) accept List.

2.4.0-beta1 - 2019-02-04

- Added support for Direct Https.
- Changed the APIs accepting java.util.Collection for which order is important to accept java.util.List instead. Now ConnectionPolicy#getPreferredLocations(), JsonSerialization, and PartitionKey(.) accept List.
- Fixed a Session bug for Document query in Gateway mode.
- Upgraded dependencies (netty 0.4.20 github #79, RxJava 1.3.8).

2.3.1 - 2019-01-15

- Fix handling very large query responses.

- Fix resource token handling when instantiating client (github #78).
- Upgraded vulnerable dependency jackson-databind (github #77).

2.3.0 - 2018-11-29

- Fixed a resource leak bug.
- Added support for MultiPolygon
- Added support for custom headers in RequestOptions.

2.2.2 - 2018-11-26

- Fixed a packaging bug.

2.2.1 - 2018-11-05

- Fixed a NPE bug in write retry path.
- Fixed a NPE bug in endpoint management.
- Upgraded vulnerable dependencies (github #68).
- Added support for Netty network logging for troubleshooting.

2.2.0 - 2018-10-04

- Added support for Multi-region write.

2.1.0 - 2018-09-06

- Added support for Proxy.
- Added support for resource token authorization.
- Fixed a bug in handling large partition keys (github #63).
- Documentation improved.
- SDK restructured into more granular modules.

2.0.1 - 2018-08-16

- Fixed a bug for non-english locales (github #51).
- Added helper methods for Conflict resource.

2.0.0 - 2018-06-20

- Replaced org.json dependency by jackson due to performance reasons and licensing (github #29).
- Removed deprecated OfferV2 class.
- Added accessor method to Offer class for throughput content.
- Any method in Document/Resource returning org.json types changed to return a jackson object type.
- getObject(.) method of classes extending JsonSerializable changed to return a jackson ObjectNode type.
- getCollection(.) method changed to return Collection of ObjectNode.
- Removed JsonSerializable subclasses' constructors with org.json.JSONObject arg.
- JsonSerializable.toJson (SerializationFormattingPolicy.Indented) now uses two spaces for indentation.

1.0.2 - 2018-05-18

- Added support for Unique Index Policy.

- Added support for limiting response continuation token size in feed options.
- Added support for Partition Split in Cross Partition Query.
- Fixed a bug in Json timestamp serialization (github #32).
- Fixed a bug in Json enum serialization.
- Fixed a bug in managing documents of 2MB size (github #33).
- Dependency com.fasterxml.jackson.core:jackson-databind upgraded to 2.9.5 due to a bug (jackson-databind: github #1599)
- Dependency on rxjava-extras upgraded to 0.8.0.17 due to a bug (rxjava-extras: github #30).
- The metadata description in pom file updated to be inline with the rest of documentation.
- Syntax improvement (github #41), (github #40).

1.0.1 - 2018-04-20

- Added back-pressure support in query.
- Added support for partition key range id in query.
- Changed to allow larger continuation token in request header (bugfix github #24).
- netty dependency upgraded to 4.1.22.Final to ensure JVM shuts down after main thread finishes.
- Changed to avoid passing session token when reading master resources.
- Added more examples.
- Added more benchmarking scenarios.
- Fixed java header files for proper javadoc generation.

1.0.0 - 2018-02-26

- Release 1.0.0 has fully end to end support for non-blocking IO using netty library in Gateway mode.
- Dependency on `azure-documentdb` SDK removed.
- Artifact id changed to `azure-cosmosdb` from `azure-documentdb-rx` in 0.9.0-rc2.
- Java package name changed to `com.microsoft.azure.cosmosdb` from `com.microsoft.azure.documentdb` in 0.9.0-rc2.

0.9.0-rc2 - 2018-02-26

- `FeedResponsePage` renamed to `FeedReponse`
- Some minor modifications to `ConnectionPolicy` configuration. All time fields and methods in ConnectionPolicy suffixed with "InMillis" to be more precise of the time unit.
- `ConnectionPolicy#setProxy()` removed.
- `FeedOptions#pageSize` renamed to `FeedOptions#maxItemCount`
- Release 1.0.0 deprecates 0.9.x releases.

0.9.0-rc1

- First release of `azure-documentdb-rx` SDK.
- CRUD Document API fully non-blocking using netty. Query async API implemented as a wrapper using blocking SDK `azure-documentdb`.

Release and retirement dates

Microsoft will provide notification at least **12 months** in advance of retiring an SDK in order to smooth the transition to a newer/supported version.

New features and functionality and optimizations are only added to the current SDK. So it's recommended that you always upgrade to the latest SDK version as early as possible.

Any request to Cosmos DB using a retired SDK will be rejected by the service.

> **WARNING**
> All versions **1.x** of the Async Java SDK for SQL API will be retired on **August 30, 2020**.

VERSION	RELEASE DATE	RETIREMENT DATE
2.6.4	Jan 2, 2020	---
2.6.3	Oct 23, 2019	---
2.6.2	Oct 5, 2019	---
2.6.1	Sept 4, 2019	---
2.6.0	Aug 1, 2019	---
2.5.1	Jul 31, 2019	---
2.4.6	Jul 19, 2019	---
2.5.0	Jun 25, 2019	---
2.4.5	May 6, 2019	---
2.4.3	Mar 5, 2019	---
2.4.2	Mar 1, 2019	---
2.4.1	Feb 20, 2019	---
2.4.0	Feb 8, 2019	---
2.4.0-beta-1	Feb 4, 2019	---
2.3.1	Jan 15, 2019	---
2.3.0	Nov 29, 2018	---
2.2.2	Nov 8, 2018	---
2.2.1	Nov 2, 2018	---
2.2.0	September 22, 2018	---
2.1.0	September 5, 2018	---

VERSION	RELEASE DATE	RETIREMENT DATE
2.0.1	August 16, 2018	---
2.0.0	June 20, 2018	---
1.0.2	May 18, 2018	August 30, 2020
1.0.1	April 20, 2018	August 30, 2020
1.0.0	February 27, 2018	August 30, 2020

FAQ

How will I be notified of the retiring SDK?

Microsoft will provide 12 month's advance notice before the end of support of the retiring SDK to facilitate a smooth transition to a supported SDK. We'll notify you through various communication channels: the Azure portal, Azure updates, and direct communication to assigned service administrators.

Can I author applications by using a to-be-retired Azure Cosmos DB SDK during the 12-month period?

Yes, you'll be able to author, deploy, and modify applications by using the to-be-retired Azure Cosmos DB SDK during the 12-month notice period. We recommend that you migrate to a newer supported version of the Azure Cosmos DB SDK during the 12-month notice period, as appropriate.

After the retirement date, what happens to applications that use the unsupported Azure Cosmos DB SDK?

After the retirement date, Azure Cosmos DB will no longer make bug fixes, add new features, or provide support to the retired SDK versions. If you prefer not to upgrade, requests sent from the retired versions of the SDK will continue to be served by the Azure Cosmos DB service.

Which SDK versions will have the latest features and updates?

New features and updates will be added only to the latest minor version of the latest supported major SDK version. We recommend that you always use the latest version to take advantage of new features, performance improvements, and bug fixes. If you're using an old, non-retired version of the SDK, your requests to Azure Cosmos DB will still function, but you won't have access to any new capabilities.

What should I do if I can't update my application before a cutoff date?

We recommend that you upgrade to the latest SDK as early as possible. After an SDK is tagged for retirement, you'll have 12 months to update your application. If you're not able to update by the retirement date, requests sent from the retired versions of the SDK will continue to be served by Azure Cosmos DB, so your running applications will continue to function. But Azure Cosmos DB will no longer make bug fixes, add new features, or provide support to the retired SDK versions.

If you have a support plan and require technical support, contact us by filing a support ticket.

See also

To learn more about Cosmos DB, see Microsoft Azure Cosmos DB service page.

Azure Cosmos DB Java SDK for SQL API: Release notes and resources

12/4/2020 • 10 minutes to read • Edit Online

APPLIES TO: SQL API

This is the original Azure Cosmos DB Sync Java SDK v2 for SQL API which supports synchronous operations.

> **IMPORTANT**
>
> This is *not* the latest Java SDK for Azure Cosmos DB! Consider using Azure Cosmos DB Java SDK v4 for your project. To upgrade, follow the instructions in the Migrate to Azure Cosmos DB Java SDK v4 guide and the Reactor vs RxJava guide.

SDK Download	Maven
API documentation	Java API reference documentation
Contribute to SDK	GitHub
Get started	Get started with the Java SDK
Web app tutorial	Web application development with Azure Cosmos DB
Minimum supported runtime	Java Development Kit (JDK) 7+

Release notes

2.5.1

- Fixes primary partition cache issue on documentCollection query.

2.5.0

- Added support for 449 retry custom configuration.

2.4.7

- Fixes connection pool timeout issue.
- Fixes auth token refresh on internal retries.

2.4.6

- Updated correct client side replica policy tag on databaseAccount and made databaseAccount configuration reads from cache.

2.4.5

- Avoiding retry on invalid partition key range error, if user provides pkRangeId.

2.4.4

- Optimized partition key range cache refreshes.

- Fixes the scenario where the SDK doesn't entertain partition split hint from server and results in incorrect client side routing caches refresh.

2.4.2

- Optimized collection cache refreshes.

2.4.1

- Added support to retrieve inner exception message from request diagnostic string.

2.4.0

- Introduced version api on PartitionKeyDefinition.

2.3.0

- Added separate timeout support for direct mode.

2.2.3

- Consuming null error message from service and producing document client exception.

2.2.2

- Socket connection improvement, adding SoKeepAlive default true.

2.2.0

- Added request diagnostics string support.

2.1.3

- Fixed bug in PartitionKey for Hash V2.

2.1.2

- Added support for composite indexes.
- Fixed bug in global endpoint manager to force refresh.
- Fixed bug for upserts with pre-conditions in direct mode.

2.1.1

- Fixed bug in gateway address cache.

2.1.0

- Multi-region write support added for direct mode.
- Added support for handling IOExceptions thrown as ServiceUnavailable exceptions, from a proxy.
- Fixed a bug in endpoint discovery retry policy.
- Fixed a bug to ensure null pointer exceptions are not thrown in BaseDatabaseAccountConfigurationProvider.
- Fixed a bug to ensure QueryIterator does not return nulls.
- Fixed a bug to ensure large PartitionKey is allowed

2.0.0

- Multi-region write support added for gateway mode.

1.16.4

- Fixed a bug in Read partition Key ranges for a query.

1.16.3

- Fixed a bug in setting continuation token header size in DirectHttps mode.

1.16.2

- Added streaming fail over support.

- Added support for custom metadata.
- Improved session handling logic.
- Fixed a bug in partition key range cache.
- Fixed a NPE bug in direct mode.

1.16.1

- Added support for Unique Index.
- Added support for limiting continuation token size in feed-options.
- Fixed a bug in Json Serialization (timestamp).
- Fixed a bug in Json Serialization (enum).
- Dependency on com.fasterxml.jackson.core:jackson-databind upgraded to 2.9.5.

1.16.0

- Improved Connection Pooling for Direct Mode.
- Improved Prefetch improvement for non-orderby cross partition query.
- Improved UUID generation.
- Improved Session consistency logic.
- Added support for multipolygon.
- Added support for Partition Key Range Statistics for Collection.
- Fixed a bug in Multi-region support.

1.15.0

- Improved Json Serialization performance.
- This SDK version requires the latest version of Azure Cosmos DB Emulator.

1.14.0

- Internal changes for Microsoft friends libraries.

1.13.0

- Fixed an issue in reading single partition key ranges.
- Fixed an issue in ResourceID parsing that affects database with short names.
- Fixed an issue cause by partition key encoding.

1.12.0

- Critical bug fixes to request processing during partition splits.
- Fixed an issue with the Strong and BoundedStaleness consistency levels.

1.11.0

- Added support for a new consistency level called ConsistentPrefix.
- Fixed a bug in reading collection in session mode.

1.10.0

- Enabled support for partitioned collection with as low as 2,500 RU/sec and scale in increments of 100 RU/sec.
- Fixed a bug in the native assembly which can cause NullRef exception in some queries.

1.9.6

- Fixed a bug in the query engine configuration that may cause exceptions for queries in Gateway mode.
- Fixed a few bugs in the session container that may cause an "Owner resource not found" exception for requests immediately after collection creation.

1.9.5

- Added support for aggregation queries (COUNT, MIN, MAX, SUM, and AVG). See Aggregation support.
- Added support for change feed.
- Added support for collection quota information through RequestOptions.setPopulateQuotaInfo.
- Added support for stored procedure script logging through RequestOptions.setScriptLoggingEnabled.
- Fixed a bug where query in DirectHttps mode may stop responding when encountering throttle failures.
- Fixed a bug in session consistency mode.
- Fixed a bug which may cause NullReferenceException in HttpContext when request rate is high.
- Improved performance of DirectHttps mode.

1.9.4

- Added simple client instance-based proxy support with ConnectionPolicy.setProxy() API.
- Added DocumentClient.close() API to properly shutdown DocumentClient instance.
- Improved query performance in direct connectivity mode by deriving the query plan from the native assembly instead of the Gateway.
- Set FAIL_ON_UNKNOWN_PROPERTIES = false so users don't need to define JsonIgnoreProperties in their POJO.
- Refactored logging to use SLF4J.
- Fixed a few other bugs in consistency reader.

1.9.3

- Fixed a bug in the connection management to prevent connection leaks in direct connectivity mode.
- Fixed a bug in the TOP query where it may throw NullReference exception.
- Improved performance by reducing the number of network call for the internal caches.
- Added status code, ActivityID and Request URI in DocumentClientException for better troubleshooting.

1.9.2

- Fixed an issue in the connection management for stability.

1.9.1

- Added support for BoundedStaleness consistency level.
- Added support for direct connectivity for CRUD operations for partitioned collections.
- Fixed a bug in querying a database with SQL.
- Fixed a bug in the session cache where session token may be set incorrectly.

1.9.0

- Added support for cross partition parallel queries.
- Added support for TOP/ORDER BY queries for partitioned collections.
- Added support for strong consistency.
- Added support for name based requests when using direct connectivity.
- Fixed to make ActivityId stay consistent across all request retries.
- Fixed a bug related to the session cache when recreating a collection with the same name.
- Added Polygon and LineString DataTypes while specifying collection indexing policy for geo-fencing spatial queries.
- Fixed issues with Java Doc for Java 1.8.

1.8.1

- Fixed a bug in PartitionKeyDefinitionMap to cache single partition collections and not make extra fetch partition key requests.
- Fixed a bug to not retry when an incorrect partition key value is provided.

1.8.0

- Added the support for multi-region database accounts.
- Added support for automatic retry on throttled requests with options to customize the max retry attempts and max retry wait time. See RetryOptions and ConnectionPolicy.getRetryOptions().
- Deprecated IPartitionResolver based custom partitioning code. Please use partitioned collections for higher storage and throughput.

1.7.1

- Added retry policy support for rate limiting.

1.7.0

- Added time to live (TTL) support for documents.

1.6.0

- Implemented partitioned collections and user-defined performance levels.

1.5.1

- Fixed a bug in HashPartitionResolver to generate hash values in little-endian to be consistent with other SDKs.

1.5.0

- Add Hash & Range partition resolvers to assist with sharding applications across multiple partitions.

1.4.0

- Implement Upsert. New upsertXXX methods added to support Upsert feature.
- Implement ID Based Routing. No public API changes, all changes internal.

1.3.0

- Release skipped to bring version number in alignment with other SDKs

1.2.0

- Supports GeoSpatial Index
- Validates ID property for all resources. Ids for resources cannot contain ?, /, #, , characters or end with a space.
- Adds new header "index transformation progress" to ResourceResponse.

1.1.0

- Implements V2 indexing policy

1.0.0

- GA SDK

Release and retirement dates

Microsoft will provide notification at least **12 months** in advance of retiring an SDK in order to smooth the transition to a newer/supported version. New features and functionality and optimizations are only added to the current SDK, as such it is recommend that you always upgrade to the latest SDK version as early as possible.

WARNING

After 30 May 2020, Azure Cosmos DB will no longer make bug fixes, add new features, and provide support to versions 1.x of the Azure Cosmos DB Java SDK for SQL API. If you prefer not to upgrade, requests sent from version 1.x of the SDK will continue to be served by the Azure Cosmos DB service.

After 29 February 2016, Azure Cosmos DB will no longer make bug fixes, add new features, and provide support to versions 0.x of the Azure Cosmos DB Java SDK for SQL API. If you prefer not to upgrade, requests sent from version 0.x of the SDK will continue to be served by the Azure Cosmos DB service.

VERSION	RELEASE DATE	RETIREMENT DATE
2.5.1	June 03, 2020	---
2.5.0	May 12, 2020	---
2.4.7	Feb 20, 2020	---
2.4.6	Jan 24, 2020	---
2.4.5	Nov 10, 2019	---
2.4.4	Oct 24, 2019	---
2.4.2	Sep 26, 2019	---
2.4.1	Jul 18, 2019	---
2.4.0	May 04, 2019	---
2.3.0	Apr 24, 2019	---
2.2.3	Apr 16, 2019	---
2.2.2	Apr 05, 2019	---
2.2.0	Mar 27, 2019	---
2.1.3	Mar 13, 2019	---
2.1.2	Mar 09, 2019	---
2.1.1	Dec 13, 2018	---
2.1.0	Nov 20, 2018	---
2.0.0	Sept 21, 2018	---
1.16.4	Sept 10, 2018	May 30, 2020
1.16.3	Sept 09, 2018	May 30, 2020

VERSION	RELEASE DATE	RETIREMENT DATE
1.16.2	June 29, 2018	May 30, 2020
1.16.1	May 16, 2018	May 30, 2020
1.16.0	March 15, 2018	May 30, 2020
1.15.0	Nov 14, 2017	May 30, 2020
1.14.0	Oct 28, 2017	May 30, 2020
1.13.0	August 25, 2017	May 30, 2020
1.12.0	July 11, 2017	May 30, 2020
1.11.0	May 10, 2017	May 30, 2020
1.10.0	March 11, 2017	May 30, 2020
1.9.6	February 21, 2017	May 30, 2020
1.9.5	January 31, 2017	May 30, 2020
1.9.4	November 24, 2016	May 30, 2020
1.9.3	October 30, 2016	May 30, 2020
1.9.2	October 28, 2016	May 30, 2020
1.9.1	October 26, 2016	May 30, 2020
1.9.0	October 03, 2016	May 30, 2020
1.8.1	June 30, 2016	May 30, 2020
1.8.0	June 14, 2016	May 30, 2020
1.7.1	April 30, 2016	May 30, 2020
1.7.0	April 27, 2016	May 30, 2020
1.6.0	March 29, 2016	May 30, 2020
1.5.1	December 31, 2015	May 30, 2020
1.5.0	December 04, 2015	May 30, 2020
1.4.0	October 05, 2015	May 30, 2020
1.3.0	October 05, 2015	May 30, 2020

VERSION	RELEASE DATE	RETIREMENT DATE
1.2.0	August 05, 2015	May 30, 2020
1.1.0	July 09, 2015	May 30, 2020
1.0.1	May 12, 2015	May 30, 2020
1.0.0	April 07, 2015	May 30, 2020
0.9.5-prelease	Mar 09, 2015	February 29, 2016
0.9.4-prelease	February 17, 2015	February 29, 2016
0.9.3-prelease	January 13, 2015	February 29, 2016
0.9.2-prelease	December 19, 2014	February 29, 2016
0.9.1-prelease	December 19, 2014	February 29, 2016
0.9.0-prelease	December 10, 2014	February 29, 2016

FAQ

How will I be notified of the retiring SDK?

Microsoft will provide 12 month's advance notice before the end of support of the retiring SDK to facilitate a smooth transition to a supported SDK. We'll notify you through various communication channels: the Azure portal, Azure updates, and direct communication to assigned service administrators.

Can I author applications by using a to-be-retired Azure Cosmos DB SDK during the 12-month period?

Yes, you'll be able to author, deploy, and modify applications by using the to-be-retired Azure Cosmos DB SDK during the 12-month notice period. We recommend that you migrate to a newer supported version of the Azure Cosmos DB SDK during the 12-month notice period, as appropriate.

After the retirement date, what happens to applications that use the unsupported Azure Cosmos DB SDK?

After the retirement date, Azure Cosmos DB will no longer make bug fixes, add new features, or provide support to the retired SDK versions. If you prefer not to upgrade, requests sent from the retired versions of the SDK will continue to be served by the Azure Cosmos DB service.

Which SDK versions will have the latest features and updates?

New features and updates will be added only to the latest minor version of the latest supported major SDK version. We recommend that you always use the latest version to take advantage of new features, performance improvements, and bug fixes. If you're using an old, non-retired version of the SDK, your requests to Azure Cosmos DB will still function, but you won't have access to any new capabilities.

What should I do if I can't update my application before a cutoff date?

We recommend that you upgrade to the latest SDK as early as possible. After an SDK is tagged for retirement, you'll have 12 months to update your application. If you're not able to update by the retirement date, requests sent from the retired versions of the SDK will continue to be served by Azure Cosmos DB, so

your running applications will continue to function. But Azure Cosmos DB will no longer make bug fixes, add new features, or provide support to the retired SDK versions.

If you have a support plan and require technical support, contact us by filing a support ticket.

See also

To learn more about Cosmos DB, see Microsoft Azure Cosmos DB service page.

Spring Data Azure Cosmos DB v2 for Core (SQL) API: Release notes and resources

11/2/2020 • 7 minutes to read • Edit Online

APPLIES TO: SQL API

Spring Data Azure Cosmos DB version 2 for Core (SQL) allows developers to use Azure Cosmos DB in Spring applications. Spring Data Azure Cosmos DB exposes the Spring Data interface for manipulating databases and collections, working with documents, and issuing queries. Both Sync and Async (Reactive) APIs are supported in the same Maven artifact.

The Spring Framework is a programming and configuration model that streamlines Java application development. Spring streamlines the "plumbing" of applications by using dependency injection. Many developers like Spring because it makes building and testing applications more straightforward. Spring Boot extends this handling of the plumbing with an eye toward web application and microservices development. Spring Data is a programming model for accessing datastores like Azure Cosmos DB from the context of a Spring or Spring Boot application.

You can use Spring Data Azure Cosmos DB in your Azure Spring Cloud applications.

IMPORTANT

These release notes are for version 2 of Spring Data Azure Cosmos DB. You can find release notes for version 3 here.

Spring Data Azure Cosmos DB supports only the SQL API.

See the following articles for information about Spring Data on other Azure Cosmos DB APIs:

- Spring Data for Apache Cassandra with Azure Cosmos DB
- Spring Data MongoDB with Azure Cosmos DB
- Spring Data Gremlin with Azure Cosmos DB

Want to get going fast?

1. Install the minimum supported Java runtime, JDK 8, so you can use the SDK.
2. Create a Spring Data Azure Cosmos DB app by using the starter. It's easy!
3. Work through the Spring Data Azure Cosmos DB developer's guide, which walks through basic Azure Cosmos DB requests.

You can spin up Spring Boot Starter apps fast by using Spring Initializr!

Resources

RESOURCE	LINK
SDK download	Maven
API documentation	Spring Data Azure Cosmos DB reference documentation
Contribute to the SDK	Spring Data Azure Cosmos DB repo on GitHub

RESOURCE	LINK
Spring Boot Starter	Azure Cosmos DB Spring Boot Starter client library for Java
Spring TODO app sample with Azure Cosmos DB	End-to-end Java Experience in App Service Linux (Part 2)
Developer's guide	Spring Data Azure Cosmos DB developer's guide
Using Starter	How to use Spring Boot Starter with the Azure Cosmos DB SQL API GitHub repo for Azure Cosmos DB Spring Boot Starter
Sample with Azure App Service	How to use Spring and Cosmos DB with App Service on Linux TODO app sample

Release history

2.3.0 (May 21, 2020)

New features

- Updates Spring Boot version to 2.3.0.

2.2.5 (May 19, 2020)

New features

- Updates Azure Cosmos DB version to 3.7.3.

Key bug fixes

- Contains memory leak fixes and Netty version upgrades from Azure Cosmos DB SDK 3.7.3.

2.2.4 (April 6, 2020)

Key bug fixes

- Fixes `allowTelemetry` flag to take into account from `CosmosDbConfig`.
- Fixes `TTL` property on container.

2.2.3 (February 25, 2020)

New features

- Adds new `findAll` by partition key API.
- Updates Azure Cosmos DB version to 3.7.0.

Key bug fixes

- Fixes `collectionName` -> `containerName`.
- Fixes `entityClass` and `domainClass` -> `domainType`.
- Fixes "Return entity collection saved by repository instead of input entity."

2.1.10 (February 25, 2020)

Key bug fixes

- Backports fix for "Return entity collection saved by repository instead of input entity."

2.2.2 (January 15, 2020)

New features

- Updates Azure Cosmos DB version to 3.6.0.

Key bug fixes

2.2.1 (December 31, 2019)

New features

- Updates Azure Cosmos DB SDK version to 3.5.0.
- Adds annotation field to enable or disable automatic collection creation.
- Improves exception handling. Exposes `CosmosClientException` through `CosmosDBAccessException`.
- Exposes `requestCharge` and `activityId` through `ResponseDiagnostics`.

Key bug fixes

- SDK 3.5.0 update fixes "Exception when Cosmos DB HTTP response header is larger than 8192 bytes," "ConsistencyPolicy.defaultConsistencyLevel() fails on Bounded Staleness and Consistent Prefix."
- Fixes `findById` method's behavior. Previously, this method returned empty if the entity wasn't found instead of throwing an exception.
- Fixes a bug in which sorting wasn't applied on the next page when `CosmosPageRequest` was used.

2.1.9 (December 26, 2019)

New features

- Adds annotation field to enable or disable automatic collection creation.

Key bug fixes

- Fixes `findById` method's behavior. Previously, this method returned empty if the entity wasn't found instead of throwing an exception.

2.2.0 (October 21, 2019)

New features

- Complete Reactive Cosmos Repository support.
- Azure Cosmos DB Request Diagnostics String and Query Metrics support.
- Azure Cosmos DB SDK version update to 3.3.1.
- Spring Framework version upgrade to 5.2.0.RELEASE.
- Spring Data Commons version upgrade to 2.2.0.RELEASE.
- Adds `findByIdAndPartitionKey` and `deleteByIdAndPartitionKey` APIs.
- Removes dependency from azure-documentdb.
- Rebrands DocumentDB to Azure Cosmos DB.

Key bug fixes

- Fixes "Sorting throws exception when pageSize is less than total items in repository."

2.1.8 (October 18, 2019)

New features

- Deprecates DocumentDB APIs.
- Adds `findByIdAndPartitionKey` and `deleteByIdAndPartitionKey` APIs.
- Adds optimistic locking based on `_etag`.
- Enables SpEL expression for document collection name.
- Adds `ObjectMapper` improvements.

2.1.7 (October 18, 2019)

New features

- Adds Azure Cosmos DB SDK version 3 dependency.
- Adds Reactive Cosmos Repository.
- Updates implementation of `DocumentDbTemplate` to use Azure Cosmos DB SDK version 3.
- Adds other configuration changes for Reactive Cosmos Repository support.

2.1.2 (March 19, 2019)

Key bug fixes

- Removes `applicationInsights` dependency for:
  - Potential risk of dependencies polluting.
  - Java 11 incompatibility.
  - Avoiding potential performance impact to CPU and/or memory.

2.0.7 (March 20, 2019)

Key bug fixes

- Backport removes `applicationInsights` dependency for:
  - Potential risk of dependencies polluting.
  - Java 11 incompatibility.
  - Avoiding potential performance impact to CPU and/or memory.

2.1.1 (March 7, 2019)

New features

- Updates main version to 2.1.1.

2.0.6 (March 7, 2019)

New features

- Ignore all exceptions from telemetry.

2.1.0 (December 17, 2018)

New features

- Updates version to 2.1.0 to address problem.

2.0.5 (September 13, 2018)

New features

- Adds keywords `exists` and `startsWith`.
- Updates Readme.

Key bug fixes

- Fixes "Can't call self href directly for Entity."
- Fixes "findAll will fail if collection is not created."

2.0.4 (Prerelease) (August 23, 2018)

New features

- Renames package from documentdb to cosmosdb.
- Adds new feature of query method keyword. 16 keywords from SQL API are now supported.
- Adds new feature of query with paging and sorting.
- Simplifies the configuration of spring-data-cosmosdb.
- Adds `deleteCollection` and `deleteAll` APIs.

Key bug fixes

- Bug fix and defect mitigation.

FAQ

How will I be notified of the retiring SDK?

Microsoft will provide 12 month's advance notice before the end of support of the retiring SDK to facilitate a smooth transition to a supported SDK. We'll notify you through various communication channels: the Azure portal, Azure updates, and direct communication to assigned service administrators.

Can I author applications by using a to-be-retired Azure Cosmos DB SDK during the 12-month period?

Yes, you'll be able to author, deploy, and modify applications by using the to-be-retired Azure Cosmos DB SDK during the 12-month notice period. We recommend that you migrate to a newer supported version of the Azure Cosmos DB SDK during the 12-month notice period, as appropriate.

After the retirement date, what happens to applications that use the unsupported Azure Cosmos DB SDK?

After the retirement date, Azure Cosmos DB will no longer make bug fixes, add new features, or provide support to the retired SDK versions. If you prefer not to upgrade, requests sent from the retired versions of the SDK will continue to be served by the Azure Cosmos DB service.

Which SDK versions will have the latest features and updates?

New features and updates will be added only to the latest minor version of the latest supported major SDK version. We recommend that you always use the latest version to take advantage of new features, performance improvements, and bug fixes. If you're using an old, non-retired version of the SDK, your requests to Azure Cosmos DB will still function, but you won't have access to any new capabilities.

What should I do if I can't update my application before a cutoff date?

We recommend that you upgrade to the latest SDK as early as possible. After an SDK is tagged for retirement, you'll have 12 months to update your application. If you're not able to update by the retirement date, requests sent from the retired versions of the SDK will continue to be served by Azure Cosmos DB, so your running applications will continue to function. But Azure Cosmos DB will no longer make bug fixes, add new features, or provide support to the retired SDK versions.

If you have a support plan and require technical support, contact us by filing a support ticket.

Next steps

Learn more about Azure Cosmos DB.

Learn more about the Spring Framework.

Learn more about Spring Boot.

Learn more about Spring Data.

Spring Data Azure Cosmos DB v3 for Core (SQL) API: Release notes and resources

11/2/2020 • 5 minutes to read • Edit Online

APPLIES TO: SQL API

The Spring Data Azure Cosmos DB version 3 for Core (SQL) allows developers to use Azure Cosmos DB in Spring applications. Spring Data Azure Cosmos DB exposes the Spring Data interface for manipulating databases and collections, working with documents, and issuing queries. Both Sync and Async (Reactive) APIs are supported in the same Maven artifact.

Spring Data Azure Cosmos DB has a dependency on the Spring Data framework. The Azure Cosmos DB SDK team releases Maven artifacts for Spring Data versions 2.2 and 2.3.

The Spring Framework is a programming and configuration model that streamlines Java application development. Spring streamlines the "plumbing" of applications by using dependency injection. Many developers like Spring because it makes building and testing applications more straightforward. Spring Boot extends this handling of the plumbing with an eye toward web application and microservices development. Spring Data is a programming model and framework for accessing datastores like Azure Cosmos DB from the context of a Spring or Spring Boot application.

You can use Spring Data Azure Cosmos DB in your Azure Spring Cloud applications.

> **IMPORTANT**
>
> These release notes are for version 3 of Spring Data Azure Cosmos DB. You can find release notes for version 2 here.
>
> Spring Data Azure Cosmos DB supports only the SQL API.
>
> See these articles for information about Spring Data on other Azure Cosmos DB APIs:
>
> - Spring Data for Apache Cassandra with Azure Cosmos DB
> - Spring Data MongoDB with Azure Cosmos DB
> - Spring Data Gremlin with Azure Cosmos DB

Start here

- Explore
- pom.xml
- Connect
- Doc ops

These tabs contain basic Spring Data Azure Cosmos DB samples.

Resources

- **Contribute to the SDK:** Spring Data Azure Cosmos DB repo on GitHub

- **Tutorial**: Spring Data Azure Cosmos DB tutorial on GitHub

Release History

3.2.0-beta.1 (Unreleased)

3.1.0 (2020-10-21)

New Features

- Added support for `ARRAY_CONTAINS` `CriteriaType`.
- Updated azure-cosmos version to 4.7.1.

Key Bug Fixes

- Fixed an issue where annotated queries do not pick the annotated container name.

3.0.0 (2020-09-30)

New Features

- Updated azure-cosmos dependency to `4.6.0`

3.0.0-beta.2 (2020-09-17)

New Features

- Updated artifact id to `azure-spring-data-cosmos`.
- Updated azure-cosmos dependency to `4.5.0`.
- `Query Annotation` support for native queries.
- Support for Java 11.
- Added support for Nested Partition Key by exposing `partitionKeyPath` field in `@Container` annotation.
- Added support for `limit` query type allowing `top` and `first` to be used when defining repository APIs.

Key Bug Fixes

- Fixed nested partition key bug when used with `@GeneratedValue` annotation.

3.0.0-beta.1 (2020-08-17)

New Features

- Updated group id to `com.azure`.
- Updated artifact id to `azure-spring-data-cosmos-core`.
- Updated azure-cosmos SDK dependency to `4.3.2-beta.2`.
- Support for auditing entities - automatic management of createdBy, createdDate, lastModifiedBy and lastModifiedDate annotated fields.
- `@GeneratedValue` annotation support for automatic id generation for id fields of `String` type.
- Multi-database configuration support for single cosmos account with multiple databases and multiple cosmos accounts with multiple databases.
- Support for `@Version` annotation on any string field.
- Updated sync APIs return types to `Iterable` types instead of `List`.
- Exposed `CosmosClientBuilder` from Cosmos SDK as spring bean to `@Configuration` class.
- Updated `CosmosConfig` to contain query metrics and response diagnostics processor implementation.
- Support for returning `Optional` data type for single result queries.

Renames

- `CosmosDbFactory` to `CosmosFactory`.
- `CosmosDBConfig` to `CosmosConfig`.
- `CosmosDBAccessException` to `CosmosAccessException`.

- `Document` annotation to `Container` annotation.
- `DocumentIndexingPolicy` annotation to `CosmosIndexingPolicy` annotation.
- `DocumentQuery` to `CosmosQuery`.
- application.properties flag `populateQueryMetrics` to `queryMetricsEnabled`.

Key Bug Fixes

- Scheduling diagnostics logging task to `Parallel` threads to avoid blocking Netty I/O threads.
- Fixed optimistic locking on delete operation.
- Fixed issue with escaping queries for `IN` clause.
- Fixed issue by allowing `long` data type for `@Id`.
- Fixed issue by allowing `boolean`, `long`, `int`, `double` as data types for `@PartitionKey` annotation.
- Fixed `IgnoreCase` & `AllIgnoreCase` keywords for ignore case queries.
- Removed default request unit value of 4000 when creating containers automatically.

FAQ

How will I be notified of the retiring SDK?

Microsoft will provide 12 month's advance notice before the end of support of the retiring SDK to facilitate a smooth transition to a supported SDK. We'll notify you through various communication channels: the Azure portal, Azure updates, and direct communication to assigned service administrators.

Can I author applications by using a to-be-retired Azure Cosmos DB SDK during the 12-month period?

Yes, you'll be able to author, deploy, and modify applications by using the to-be-retired Azure Cosmos DB SDK during the 12-month notice period. We recommend that you migrate to a newer supported version of the Azure Cosmos DB SDK during the 12-month notice period, as appropriate.

After the retirement date, what happens to applications that use the unsupported Azure Cosmos DB SDK?

After the retirement date, Azure Cosmos DB will no longer make bug fixes, add new features, or provide support to the retired SDK versions. If you prefer not to upgrade, requests sent from the retired versions of the SDK will continue to be served by the Azure Cosmos DB service.

Which SDK versions will have the latest features and updates?

New features and updates will be added only to the latest minor version of the latest supported major SDK version. We recommend that you always use the latest version to take advantage of new features, performance improvements, and bug fixes. If you're using an old, non-retired version of the SDK, your requests to Azure Cosmos DB will still function, but you won't have access to any new capabilities.

What should I do if I can't update my application before a cutoff date?

We recommend that you upgrade to the latest SDK as early as possible. After an SDK is tagged for retirement, you'll have 12 months to update your application. If you're not able to update by the retirement date, requests sent from the retired versions of the SDK will continue to be served by Azure Cosmos DB, so your running applications will continue to function. But Azure Cosmos DB will no longer make bug fixes, add new features, or provide support to the retired SDK versions.

If you have a support plan and require technical support, contact us by filing a support ticket.

Next steps

Learn more about Azure Cosmos DB.

Learn more about the Spring Framework.

Learn more about Spring Boot.

Learn more about Spring Data.

Azure Cosmos DB Apache Spark Connector for Core (SQL) API: Release notes and resources

11/2/2020 • 4 minutes to read • Edit Online

APPLIES TO: SQL API

You can accelerate big data analytics by using the Azure Cosmos DB Apache Spark Connector for Core (SQL). The Spark Connector allows you to run Spark jobs on data stored in Azure Cosmos DB. Batch and stream processing are supported.

You can use the connector with Azure Databricks or Azure HDInsight, which provide managed Spark clusters on Azure. The following table shows supported versions:

COMPONENT	VERSION
Apache Spark	2.4.*x*, 2.3.*x*, 2.2.*x*, and 2.1.*x*
Scala	2.11
Azure Databricks (runtime version)	Later than 3.4

WARNING

This connector supports the core (SQL) API of Azure Cosmos DB. For the Cosmos DB API for MongoDB, use the MongoDB Connector for Spark. For the Cosmos DB Cassandra API, use the Cassandra Spark connector.

Resources

RESOURCE	LINK
SDK download	Download latest .jar, Maven
API documentation	Spark Connector reference
Contribute to the SDK	Azure Cosmos DB Connector for Apache Spark on GitHub
Get started	Accelerate big data analytics by using the Apache Spark to Azure Cosmos DB connector Use Apache Spark Structured Streaming with Apache Kafka and Azure Cosmos DB

Release history

3.3.0

New features

- Adds a new config option, `changefeedstartfromdatetime`, which can be used to specify the start time for when the changefeed should be processed. For more information, see Config options.

3.2.0

Key bug fixes

- Fixes a regression that caused excessive memory consumption on the executors for large result sets (for example, with millions of rows), ultimately resulting in the error `java.lang.OutOfMemoryError: GC overhead limit exceeded`.

3.1.1

Key bug fixes

- Fixes a streaming checkpoint edge case in which the `ID` contains the pipe character (|) with the `ChangeFeedMaxPagesPerBatch` config applied.

3.1.0

New features

- Adds support for bulk updates when nested partition keys are used.
- Adds support for Decimal and Float data types during writes to Azure Cosmos DB.
- Adds support for Timestamp types when they're using Long (Unix epoch) as a value.

3.0.8

Key bug fixes

- Fixes typecast exception that occurs when the `WriteThroughputBudget` config is used.

3.0.7

New features

- Adds error information for bulk failures to exception and log.

3.0.6

Key bug fixes

- Fixes streaming checkpoint issues.

3.0.5

Key bug fixes

- To reduce noise, fixes log level of a message left unintentionally with level ERROR.

3.0.4

Key bug fixes

- Fixes a bug in structured streaming during partition splits. The bug could result in some missing change feed records or Null exceptions for checkpoint writes.

3.0.3

Key bug fixes

- Fixes a bug that causes a custom schema provided for readStream to be ignored.

3.0.2

Key bug fixes

- Fixes a regression (unshaded JAR includes all shaded dependencies) that increases build time by 50 percent.

3.0.1

Key bug fixes

- Fixes a dependency problem that causes Direct Transport over TCP to fail with RequestTimeoutException.

3.0.0

New features

- Improves connection management and connection pooling to reduce the number of metadata calls.

FAQ

How will I be notified of the retiring SDK?

Microsoft will provide 12 month's advance notice before the end of support of the retiring SDK to facilitate a smooth transition to a supported SDK. We'll notify you through various communication channels: the Azure portal, Azure updates, and direct communication to assigned service administrators.

Can I author applications by using a to-be-retired Azure Cosmos DB SDK during the 12-month period?

Yes, you'll be able to author, deploy, and modify applications by using the to-be-retired Azure Cosmos DB SDK during the 12-month notice period. We recommend that you migrate to a newer supported version of the Azure Cosmos DB SDK during the 12-month notice period, as appropriate.

After the retirement date, what happens to applications that use the unsupported Azure Cosmos DB SDK?

After the retirement date, Azure Cosmos DB will no longer make bug fixes, add new features, or provide support to the retired SDK versions. If you prefer not to upgrade, requests sent from the retired versions of the SDK will continue to be served by the Azure Cosmos DB service.

Which SDK versions will have the latest features and updates?

New features and updates will be added only to the latest minor version of the latest supported major SDK version. We recommend that you always use the latest version to take advantage of new features, performance improvements, and bug fixes. If you're using an old, non-retired version of the SDK, your requests to Azure Cosmos DB will still function, but you won't have access to any new capabilities.

What should I do if I can't update my application before a cutoff date?

We recommend that you upgrade to the latest SDK as early as possible. After an SDK is tagged for retirement, you'll have 12 months to update your application. If you're not able to update by the retirement date, requests sent from the retired versions of the SDK will continue to be served by Azure Cosmos DB, so your running applications will continue to function. But Azure Cosmos DB will no longer make bug fixes, add new features, or provide support to the retired SDK versions.

If you have a support plan and require technical support, contact us by filing a support ticket.

Next steps

Learn more about Azure Cosmos DB.

Learn more about Apache Spark.

Java bulk executor library: Download information

11/2/2020 • 2 minutes to read • Edit Online

APPLIES TO: SQL API

Description	The bulk executor library allows client applications to perform bulk operations in Azure Cosmos DB accounts. bulk executor library provides BulkImport, and BulkUpdate namespaces. The BulkImport module can bulk ingest documents in an optimized way such that the throughput provisioned for a collection is consumed to its maximum extent. The BulkUpdate module can bulk update existing data in Azure Cosmos containers as patches.
SDK download	Maven
Bulk executor library in GitHub	GitHub
API documentation	Java API reference documentation
Get started	Get started with the bulk executor library Java SDK
Minimum supported runtime	Java Development Kit (JDK) 7+

Release notes

2.10.0

- Fix for DocumentAnalyzer.java to correctly extract nested partition key values from json.

2.9.4

- Add functionality in BulkDelete operations to retry on specific failures and also return a list of failures to the user that could be retried.

2.9.3

- Update for Cosmos SDK version 2.4.7.

2.9.2

- Fix for 'mergeAll' to continue on 'id' and partition key value so that any patched document properties which are placed after 'id' and partition key value get added to the updated item list.

2.9.1

- Update start degree of concurrency to 1 and added debug logs for minibatch.

Azure Cosmos DB Node.js SDK for SQL API: Release notes and resources

11/2/2020 • 14 minutes to read • Edit Online

APPLIES TO: SQL API

RESOURCE	LINK
Download SDK	NPM
API Documentation	JavaScript SDK reference documentation
SDK installation instructions	Installation instructions
Contribute to SDK	GitHub
Samples	Node.js code samples
Getting started tutorial	Get started with the JavaScript SDK
Web app tutorial	Build a Node.js web application using Azure Cosmos DB
Current supported platform	Node.js v12.x - SDK Version 3.x.x Node.js v10.x - SDK Version 3.x.x Node.js v8.x - SDK Version 3.x.x Node.js v6.x - SDK Version 2.x.x Node.js v4.2.0- SDK Version 1.x.x Node.js v0.12- SDK Version 1.x.x Node.js v0.10- SDK Version 1.x.x

Release notes

3.1.0

- Set default ResponseContinuationTokenLimitInKB to 1kb. By default, we are capping this to 1kb to avoid long headers (Node.js has a global header size limit). A user may set this field to allow for longer headers, which can help the backend optimize query execution.
- Remove disableSSLVerification. This option has new alternatives described in #388

3.0.4

- Allow initialHeaders to explicitly set partition key header
- Use package.json#files to prevent extraneous files from being published
- Fix routing map sort error on older version of node+v8
- Fixes bug when user supplies partial retry options

3.0.3

- Prevent webpack from resolving modules called with require

3.0.2

- Fixes a long outstanding bug where RUs were always being reported as 0 for aggregate queries

3.0.0

🎉 v3 release! 🎉 Many new features, bug fixes, and a few breaking changes. Primary goals of this release:

- Implement major new features
  - DISTINCT queries
  - LIMIT/OFFSET queries
  - User cancelable requests
- Update to the latest Cosmos REST API version where all containers have unlimited scale
- Make it easier to use Cosmos from the browser
- Better align with the new Azure JS SDK guidelines

Migration guide for breaking changes

Improved client Constructor options

Constructor options have been simplified:

- masterKey was renamed key and moved to the top-level
- Properties previously under options.auth have moved to the top-level

```
// v2
const client = new CosmosClient({
    endpoint: "https://your-database.cosmos.azure.com",
    auth: {
        masterKey: "your-primary-key"
    }
})

// v3
const client = new CosmosClient({
    endpoint: "https://your-database.cosmos.azure.com",
    key: "your-primary-key"
})
```

Simplified QueryIterator API

In v2 there were many different ways to iterate or retrieve results from a query. We have attempted to simplify the v3 API and remove similar or duplicate APIs:

- Remove iterator.next() and iterator.current(). Use fetchNext() to get pages of results.
- Remove iterator.forEach(). Use async iterators instead.
- iterator.executeNext() renamed to iterator.fetchNext()
- iterator.toArray() renamed to iterator.fetchAll()
- Pages are now proper Response objects instead of plain JS objects
- const container = client.database(dbId).container(containerId)

```
// v2
container.items.query('SELECT * from c').toArray()
container.items.query('SELECT * from c').executeNext()
container.items.query('SELECT * from c').forEach(({ body: item }) => { console.log(item.id) })

// v3
container.items.query('SELECT * from c').fetchAll()
container.items.query('SELECT * from c').fetchNext()
for await(const { result: item } in client.databases.readAll().getAsyncIterator()) {
    console.log(item.id)
}
```

Fixed containers are now partitioned

The Cosmos service now supports partition keys on all containers, including those that were previously created as fixed containers. The v3 SDK updates to the latest API version that implements this change, but it is

not breaking. If you do not supply a partition key for operations, we will default to a system key that works with all your existing containers and documents.

Upsert removed for stored procedures

Previously upsert was allowed for non-partitioned collections, but with the API version update, all collections are partitioned so we removed it entirely.

Item reads will not throw on 404

const container = client.database(dbId).container(containerId)

```
// v2
try {
    container.items.read(id, undefined)
} catch (e) {
    if (e.code === 404) { console.log('item not found') }
}

// v3
const { result: item }  = container.items.read(id, undefined)
if (item === undefined) { console.log('item not found') }
```

Default multi-region write

The SDK will now write to multiple regions by default if your Cosmos configuration supports it. This was previously opt-in behavior.

Proper error objects

Failed requests now throw proper Error or subclasses of Error. Previously they threw plain JS objects.

New features

User-cancelable requests

The move to fetch internally allows us to use the browser AbortController API to support user-cancelable operations. In the case of operations where multiple requests are potentially in progress (like cross partition queries), all requests for the operation will be canceled. Modern browser users will already have AbortController. Node.js users will need to use a polyfill library

```
 const controller = new AbortController()
 const {result: item} = await items.query('SELECT * from c', { abortSignal: controller.signal});
 controller.abort()
```

Set throughput as part of db/container create operation

```
const { database }  = client.databases.create({ id: 'my-database', throughput: 10000 })
database.containers.create({ id: 'my-container', throughput: 10000 })
```

@azure/cosmos-sign

Header token generation was split out into a new library, @azure/cosmos-sign. Anyone calling the Cosmos REST API directly can use this to sign headers using the same code we call inside @azure/cosmos.

UUID for generated IDs

v2 had custom code to generate item IDs. We have switched to the well known and maintained community library uuid.

Connection strings

It is now possible to pass a connection string copied from the Azure portal:

```
const client = new CosmosClient("AccountEndpoint=https://test-
account.documents.azure.com:443/;AccountKey=c213asdasdefgdfgrtweaYPpgoeCsHbpRTHhxuMsTaw==;")
Add DISTINCT and LIMIT/OFFSET queries (#306)
 const { results } = await items.query('SELECT DISTINCT VALUE r.name FROM ROOT').fetchAll()
 const { results } = await items.query('SELECT * FROM root r OFFSET 1 LIMIT 2').fetchAll()
```

Improved browser experience

While it was possible to use the v2 SDK in the browser it was not an ideal experience. You needed to polyfill several node.js built-in libraries and use a bundler like webpack or Parcel. The v3 SDK makes the out of the box experience much better for browser users.

- Replace request internals with fetch (#245)
- Remove usage of Buffer (#330)
- Remove node builtin usage in favor of universal packages/APIs (#328)
- Switch to node-abort-controller (#294)

Bug fixes

- Fix offer read and bring back offer tests (#224)
- Fix EnableEndpointDiscovery (#207)
- Fix missing RUs on paginated results (#360)
- Expand SQL query parameter type (#346)
- Add ttl to ItemDefinition (#341)
- Fix CP query metrics (#311)
- Add activityId to FeedResponse (#293)
- Switch _ts type from string to number (#252)(#295)
- Fix Request Charge Aggregation (#289)
- Allow blank string partition keys (#277)
- Add string to conflict query type (#237)
- Add uniqueKeyPolicy to container (#234)

Engineering systems

Not always the most visible changes, but they help our team ship better code, faster.

- Use rollup for production builds (#104)
- Update to Typescript 3.5 (#327)
- Convert to TS project references. Extract test folder (#270)
- Enable noUnusedLocals and noUnusedParameters (#275)
- Azure Pipelines YAML for CI builds (#298)

2.1.5

- No code changes. Fixes an issue where some extra files were included in 2.1.4 package.

2.1.4

- Fix regional failover within retry policy
- Fix ChangeFeed hasMoreResults property
- Dev dependency updates
- Add PolicheckExclusions.txt

2.1.3

- Switch _ts type from string to number
- Fix default indexing tests
- Backport uniqueKeyPolicy to v2
- Demo and demo debugging fixes

2.1.2

- Backport offer fixes from v3 branch
- Fix bug in executeNext() type signature
- Typo fixes

2.1.1

- Build restructuring. Allows pulling the SDK version at build time.

2.1.0

New Features

- Added ChangeFeed support (#196)
- Added MultiPolygon datatype for indexing (#191)
- Add "key" property to constructor as alias for masterKey (#202)

Fixes

- Fix bug where next() was returning incorrect value on iterator

Engineering Improvements

- Add integration test for typescript consumption (#199)
- Enable installing directly from GitHub (#194)

2.0.5

- Adds interface for node Agent type. Typescript users no longer have to install @types/node as a dependency
- Preferred locations are now properly honored
- Improvements to contributing developer documentation
- Various typo fixes

2.0.4

- Fixes type definition issue introduced in 2.0.3

2.0.3

- Remove `big-integer` dependency
- Switch to reference directives for AsyncIterable type. Typescript users no longer have to customize their "lib" setting.
- Typo Fixes

2.0.2

- Fix readme links

2.0.1

- Fix retry interface implementation

2.0.0

- GA of Version 2.0.0 of the JavaScript SDK
- Added support for multi-region writes.

2.0.0-3

- RC1 of Version 2.0.0 of the JavaScript SDK for public preview.
- New object model, with top-level CosmosClient and methods split across relevant Database, Container, and Item classes.
- Support for promises.
- SDK converted to TypeScript.

1.14.4

- npm documentation fixed.

1.14.3

- Added support for default retries on connection issues.

- Added support to read collection change feed.
- Fixed session consistency bug that intermittently caused "read session not available".
- Added support for query metrics.
- Modified http Agent's maximum number of connections.

1.14.2

- Updated documentation to reference Azure Cosmos DB instead of Azure DocumentDB.
- Added Support for proxyUrl setting in ConnectionPolicy.

1.14.1

- Minor fix for case sensitive file systems.

1.14.0

- Adds support for Session Consistency.
- This SDK version requires the latest version of Azure Cosmos DB Emulator.

1.13.0

- Split proofed cross partition queries.
- Adds supports for resource link with leading and trailing slashes (and corresponding tests).

1.12.2

- npm documentation fixed.

1.12.1

- Fixed a bug in executeStoredProcedure where documents involved had special Unicode characters (LS, PS).
- Fixed a bug in handling documents with Unicode characters in the partition key.
- Fixed support for creating collections with the name media. GitHub issue #114.
- Fixed support for permission authorization token. GitHub issue #178.

1.12.0

- Added support for a new consistency level called ConsistentPrefix.
- Added support for UriFactory.
- Fixed a Unicode support bug. GitHub issue #171.

1.11.0

- Added the support for aggregation queries (COUNT, MIN, MAX, SUM, and AVG).
- Added the option for controlling degree of parallelism for cross partition queries.
- Added the option for disabling TLS verification when running against Azure Cosmos DB Emulator.
- Lowered minimum throughput on partitioned collections from 10,100 RU/s to 2500 RU/s.
- Fixed the continuation token bug for single partition collection. GitHub issue #107.
- Fixed the executeStoredProcedure bug in handling 0 as single param. GitHub issue #155.

1.10.2

- Fixed user-agent header to include the SDK version.
- Minor code cleanup.

1.10.1

- Disabling TLS verification when using the SDK to target the emulator(hostname=localhost).
- Added support for enabling script logging during stored procedure execution.

1.10.0

- Added support for cross partition parallel queries.

- Added support for TOP/ORDER BY queries for partitioned collections.

1.9.0

- Added retry policy support for throttled requests. (Throttled requests receive a request rate too large exception, error code 429.) By default, Azure Cosmos DB retries nine times for each request when error code 429 is encountered, honoring the retryAfter time in the response header. A fixed retry interval time can now be set as part of the RetryOptions property on the ConnectionPolicy object if you want to ignore the retryAfter time returned by server between the retries. Azure Cosmos DB now waits for a maximum of 30 seconds for each request that is being throttled (irrespective of retry count) and returns the response with error code 429. This time can also be overridden in the RetryOptions property on ConnectionPolicy object.
- Cosmos DB now returns x-ms-throttle-retry-count and x-ms-throttle-retry-wait-time-ms as the response headers in every request to denote the throttle retry count and the cumulative time the request waited between the retries.
- The RetryOptions class was added, exposing the RetryOptions property on the ConnectionPolicy class that can be used to override some of the default retry options.

1.8.0

- Added the support for multi-region database accounts.

1.7.0

- Added the support for Time To Live(TTL) feature for documents.

1.6.0

- Implemented partitioned collections and user-defined performance levels.

1.5.6

- Fixed RangePartitionResolver.resolveForRead bug where it was not returning links due to a bad concat of results.

1.5.5

- Fixed hashPartitionResolver resolveForRead(): When no partition key supplied was throwing exception, instead of returning a list of all registered links.

1.5.4

- Fixes issue #100 - Dedicated HTTPS Agent: Avoid modifying the global agent for Azure Cosmos DB purposes. Use a dedicated agent for all of the lib's requests.

1.5.3

- Fixes issue #81 - Properly handle dashes in media IDs.

1.5.2

- Fixes issue #95 - EventEmitter listener leak warning.

1.5.1

- Fixes issue #92 - rename folder Hash to hash for case-sensitive systems.

1.5.0

- Implement sharding support by adding hash & range partition resolvers.

1.4.0

- Implement Upsert. New upsertXXX methods on documentClient.

1.3.0

- Skipped to bring version numbers in alignment with other SDKs.

1.2.2

- Split Q promises wrapper to new repository.
- Update to package file for npm registry.

1.2.1

- Implements ID Based Routing.
- Fixes Issue #49 - current property conflicts with method current().

1.2.0

- Added support for GeoSpatial index.
- Validates ID property for all resources. IDs for resources cannot contain ?, /, #, //, characters or end with a space.
- Adds new header "index transformation progress" to ResourceResponse.

1.1.0

- Implements V2 indexing policy.

1.0.3

- Issue #40 - Implemented eslint and grunt configurations in the core and promise SDK.

1.0.2

- Issue #45 - Promises wrapper does not include header with error.

1.0.1

- Implemented ability to query for conflicts by adding readConflicts, readConflictAsync, and queryConflicts.
- Updated API documentation.
- Issue #41 - client.createDocumentAsync error.

1.0.0

- GA SDK.

Release & Retirement Dates

Microsoft provides notification at least **12 months** in advance of retiring an SDK in order to smooth the transition to a newer/supported version. New features and functionality and optimizations are only added to the current SDK, as such it is recommended that you always upgrade to the latest SDK version as early as possible.

VERSION	RELEASE DATE	RETIREMENT DATE
3.1.0	July 26, 2019	---
3.0.4	July 22, 2019	---
3.0.3	July 17, 2019	---
3.0.2	July 9, 2019	---
3.0.0	June 28, 2019	---
2.1.5	March 20, 2019	---
2.1.4	March 15, 2019	---

VERSION	RELEASE DATE	RETIREMENT DATE
2.1.3	March 8, 2019	---
2.1.2	January 28, 2019	---
2.1.1	December 5, 2018	---
2.1.0	December 4, 2018	---
2.0.5	November 7, 2018	---
2.0.4	October 30, 2018	---
2.0.3	October 30, 2018	---
2.0.2	October 10, 2018	---
2.0.1	September 25, 2018	---
2.0.0	September 24, 2018	---
2.0.0-3 (RC)	August 2, 2018	---
1.14.4	May 03, 2018	August 30, 2020
1.14.3	May 03, 2018	August 30, 2020
1.14.2	December 21, 2017	August 30, 2020
1.14.1	November 10, 2017	August 30, 2020
1.14.0	November 9, 2017	August 30, 2020
1.13.0	October 11, 2017	August 30, 2020
1.12.2	August 10, 2017	August 30, 2020
1.12.1	August 10, 2017	August 30, 2020
1.12.0	May 10, 2017	August 30, 2020
1.11.0	March 16, 2017	August 30, 2020
1.10.2	January 27, 2017	August 30, 2020
1.10.1	December 22, 2016	August 30, 2020
1.10.0	October 03, 2016	August 30, 2020
1.9.0	July 07, 2016	August 30, 2020

VERSION	RELEASE DATE	RETIREMENT DATE
1.8.0	June 14, 2016	August 30, 2020
1.7.0	April 26, 2016	August 30, 2020
1.6.0	March 29, 2016	August 30, 2020
1.5.6	March 08, 2016	August 30, 2020
1.5.5	February 02, 2016	August 30, 2020
1.5.4	February 01, 2016	August 30, 2020
1.5.2	January 26, 2016	August 30, 2020
1.5.2	January 22, 2016	August 30, 2020
1.5.1	January 4, 2016	August 30, 2020
1.5.0	December 31, 2015	August 30, 2020
1.4.0	October 06, 2015	August 30, 2020
1.3.0	October 06, 2015	August 30, 2020
1.2.2	September 10, 2015	August 30, 2020
1.2.1	August 15, 2015	August 30, 2020
1.2.0	August 05, 2015	August 30, 2020
1.1.0	July 09, 2015	August 30, 2020
1.0.3	June 04, 2015	August 30, 2020
1.0.2	May 23, 2015	August 30, 2020
1.0.1	May 15, 2015	August 30, 2020
1.0.0	April 08, 2015	August 30, 2020

FAQ

How will I be notified of the retiring SDK?

Microsoft will provide 12 month's advance notice before the end of support of the retiring SDK to facilitate a smooth transition to a supported SDK. We'll notify you through various communication channels: the Azure portal, Azure updates, and direct communication to assigned service administrators.

Can I author applications by using a to-be-retired Azure Cosmos DB SDK during the 12-month period?

Yes, you'll be able to author, deploy, and modify applications by using the to-be-retired Azure Cosmos DB SDK during the 12-month notice period. We recommend that you migrate to a newer supported version of the Azure Cosmos DB SDK during the 12-month notice period, as appropriate.

After the retirement date, what happens to applications that use the unsupported Azure Cosmos DB SDK?

After the retirement date, Azure Cosmos DB will no longer make bug fixes, add new features, or provide support to the retired SDK versions. If you prefer not to upgrade, requests sent from the retired versions of the SDK will continue to be served by the Azure Cosmos DB service.

Which SDK versions will have the latest features and updates?

New features and updates will be added only to the latest minor version of the latest supported major SDK version. We recommend that you always use the latest version to take advantage of new features, performance improvements, and bug fixes. If you're using an old, non-retired version of the SDK, your requests to Azure Cosmos DB will still function, but you won't have access to any new capabilities.

What should I do if I can't update my application before a cutoff date?

We recommend that you upgrade to the latest SDK as early as possible. After an SDK is tagged for retirement, you'll have 12 months to update your application. If you're not able to update by the retirement date, requests sent from the retired versions of the SDK will continue to be served by Azure Cosmos DB, so your running applications will continue to function. But Azure Cosmos DB will no longer make bug fixes, add new features, or provide support to the retired SDK versions.

If you have a support plan and require technical support, contact us by filing a support ticket.

See also

To learn more about Cosmos DB, see Microsoft Azure Cosmos DB service page.

Azure Cosmos DB Python SDK for SQL API: Release notes and resources

11/2/2020 • 11 minutes to read • Edit Online

APPLIES TO: SQL API

Download SDK	PyPI
API documentation	Python API reference documentation
SDK installation instructions	Python SDK installation instructions
Get started	Get started with the Python SDK
Current supported platform	Python 2.7 and Python 3.5.3+

Release history

4.1.0 (2020-08-10)

- Added deprecation warning for "lazy" indexing mode. The backend no longer allows creating containers with this mode and will set them to consistent instead.

New features

- Added the ability to set the analytical storage TTL when creating a new container.

Bug fixes

- Fixed support for dicts as inputs for get_client APIs.
- Fixed Python 2/3 compatibility in query iterators.
- Fixed type hint error (Issue #12570).
- Fixed bug where options headers were not added to upsert_item function. Issue #11791 - thank you @aalapatirvbd.
- Fixed error raised when a non string ID is used in an item. It now raises TypeError rather than AttributeError (Issue #11793).

4.0.0

- Stable release.
- Added HttpLoggingPolicy to pipeline to enable passing in a custom logger for request and response headers.

4.0.0b6

- Fixed bug in synchronized_request for media APIs.
- Removed MediaReadMode and MediaRequestTimeout from ConnectionPolicy as media requests are not supported.

4.0.0b5

- azure.cosmos.errors module deprecated and replaced by azure.cosmos.exceptions

- The access condition parameters (`access_condition`, `if_match`, `if_none_match`) have been deprecated in favor of separate `match_condition` and `etag` parameters.
- Fixed bug in routing map provider.
- Added query Distinct, Offset, and Limit support.
- Default document query execution context now used for
  - ChangeFeed queries
  - single partition queries (partitionkey, partitionKeyRangeId is present in options)
  - Non-document queries
- Errors out for aggregates on multiple partitions, with enable cross partition query set to true, but no "value" keyword present
- Hits query plan endpoint for other scenarios to fetch query plan
- Added `__repr__` support for Cosmos entity objects.
- Updated documentation.

4.0.0b4

- Added support for a `timeout` keyword argument to all operations to specify an absolute timeout in seconds within which the operation must be completed. If the timeout value is exceeded, a `azure.cosmos.errors.CosmosClientTimeoutError` will be raised.
- Added a new `ConnectionRetryPolicy` to manage retry behavior during HTTP connection errors.
- Added new constructor and per-operation configuration keyword arguments:
  - `retry_total` - Maximum retry attempts.
  - `retry_backoff_max` - Maximum retry wait time in seconds.
  - `retry_fixed_interval` - Fixed retry interval in milliseconds.
  - `retry_read` - Maximum number of sockets read retry attempts.
  - `retry_connect` - Maximum number of connection error retry attempts.
  - `retry_status` - Maximum number of retry attempts on error status codes.
  - `retry_on_status_codes` - A list of specific status codes to retry on.
  - `retry_backoff_factor` - Factor to calculate wait time between retry attempts.

4.0.0b3

- Added `create_database_if_not_exists()` and `create_container_if_not_exists` functionalities to CosmosClient and Database respectively.

4.0.0b2

- Version 4.0.0b2 is the second iteration in our efforts to build a client library that suits the Python language best practices.

Breaking changes

- The client connection has been adapted to consume the HTTP pipeline defined in `azure.core.pipeline`.
- Interactive objects have now been renamed as proxies. This includes:
  - `Database` -> `DatabaseProxy`
  - `User` -> `UserProxy`
  - `Container` -> `ContainerProxy`

  - `Scripts` -> `ScriptsProxy`
- The constructor of `CosmosClient` has been updated:
  - The `auth` parameter has been renamed to `credential` and will now take an authentication type directly. This means the primary key value, a dictionary of resource tokens, or a list of permissions can be passed in. However the old dictionary format is still supported.
  - The `connection_policy` parameter has been made a keyword only parameter, and while it is still supported, each of the individual attributes of the policy can now be passed in as explicit keyword arguments:
    - `request_timeout`
    - `media_request_timeout`
    - `connection_mode`
    - `media_read_mode`
    - `proxy_config`
    - `enable_endpoint_discovery`
    - `preferred_locations`
    - `multiple_write_locations`
- A new constructor has been added to `CosmosClient` to enable creation via a connection string retrieved from the Azure portal.
- Some `read_all` operations have been renamed to `list` operations:
  - `CosmosClient.read_all_databases` -> `CosmosClient.list_databases`
  - `Container.read_all_conflicts` -> `ContainerProxy.list_conflicts`
  - `Database.read_all_containers` -> `DatabaseProxy.list_containers`
  - `Database.read_all_users` -> `DatabaseProxy.list_users`
  - `User.read_all_permissions` -> `UserProxy.list_permissions`
- All operations that take `request_options` or `feed_options` parameters, these have been moved to keyword only parameters. In addition, while these options dictionaries are still supported, each of the individual options within the dictionary are now supported as explicit keyword arguments.
- The error hierarchy is now inherited from `azure.core.AzureError`:
  - `HTTPFailure` has been renamed to `CosmosHttpResponseError`
  - `JSONParseFailure` has been removed and replaced by `azure.core.DecodeError`
  - Added additional errors for specific response codes:
    - `CosmosResourceNotFoundError` for status 404
    - `CosmosResourceExistsError` for status 409
    - `CosmosAccessConditionFailedError` for status 412
- `CosmosClient` can now be run in a context manager to handle closing the client connection.
- Iterable responses (for example, query responses and list responses) are now of type `azure.core.paging.ItemPaged`. The method `fetch_next_block` has been replaced by a secondary iterator, accessed by the `by_page` method.

4.0.0b1

Version 4.0.0b1 is the first preview of our efforts to create a user-friendly client library that suits the Python language best practices. For more information about this, and preview releases of other Azure SDK libraries, please visit https://aka.ms/azure-sdk-preview1-python.

Breaking changes: New API design

- Operations are now scoped to a particular client:
  - `CosmosClient`: This client handles account-level operations. This includes managing service properties and listing the databases within an account.
  - `Database`: This client handles database-level operations. This includes creating and deleting containers, users, and stored procedures. It can be accessed from a cosmosClient` instance by name.
  - `Container`: This client handles operations for a particular container. This includes querying and inserting items and managing properties.
  - `User`: This client handles operations for a particular user. This includes adding and deleting permissions and managing user properties.

    These clients can be accessed by navigating down the client hierarchy using the `get_<child>_client` method. For full details on the new API, please see the reference documentation.
- Clients are accessed by name rather than by ID. No need to concatenate strings to create links.
- No more need to import types and methods from individual modules. The public API surface area is available directly in the `azure.cosmos` package.
- Individual request properties can be provided as keyword arguments rather than constructing a separate `RequestOptions` instance.

3.0.2

- Added Support for MultiPolygon Datatype
- Bug Fix in Session Read Retry Policy
- Bug Fix for Incorrect padding issues while decoding base 64 strings

3.0.1

- Bug fix in LocationCache
- Bug fix endpoint retry logic
- Fixed documentation

3.0.0

- Multi-regions write support added
- Naming changes
  - DocumentClient to CosmosClient
  - Collection to Container
  - Document to Item
  - Package name updated to "azure-cosmos"
  - Namespace updated to "azure.cosmos"

2.3.3

- Added support for proxy
- Added support for reading change feed
- Added support for collection quota headers
- Bugfix for large session tokens issue
- Bugfix for ReadMedia API
- Bugfix in partition key range cache

2.3.2

- Added support for default retries on connection issues.

2.3.1

- Updated documentation to reference Azure Cosmos DB instead of Azure DocumentDB.

2.3.0

- This SDK version requires the latest version of Azure Cosmos DB Emulator available for download from https://aka.ms/cosmosdb-emulator.

2.2.1

- bugfix for aggregate dict
- bugfix for trimming slashes in the resource link
- tests for unicode encoding

2.2.0

- Added support for Request Unit per Minute (RU/m) feature.
- Added support for a new consistency level called ConsistentPrefix.

2.1.0

- Added support for aggregation queries (COUNT, MIN, MAX, SUM, and AVG).
- Added an option for disabling SSL verification when running against DocumentDB Emulator.
- Removed the restriction of dependent requests module to be exactly 2.10.0.
- Lowered minimum throughput on partitioned collections from 10,100 RU/s to 2500 RU/s.
- Added support for enabling script logging during stored procedure execution.
- REST API version bumped to '2017-01-19' with this release.

2.0.1

- Made editorial changes to documentation comments.

2.0.0

- Added support for Python 3.5.
- Added support for connection pooling using the requests module.
- Added support for session consistency.
- Added support for TOP/ORDERBY queries for partitioned collections.

1.9.0

- Added retry policy support for throttled requests. (Throttled requests receive a request rate too large exception, error code 429.) By default, DocumentDB retries nine times for each request when error code 429 is encountered, honoring the retryAfter time in the response header. A fixed retry interval time can now be set as part of the RetryOptions property on the ConnectionPolicy object if you want to ignore the retryAfter time returned by server between the retries. DocumentDB now waits for a maximum of 30 seconds for each request that is being throttled (irrespective of retry count) and returns the response with error code 429. This time can also be overridden in the RetryOptions property on ConnectionPolicy object.

- DocumentDB now returns x-ms-throttle-retry-count and x-ms-throttle-retry-wait-time-ms as the response headers in every request to denote the throttle retry count and the cumulative time the request waited between the retries.

- Removed the RetryPolicy class and the corresponding property (retry_policy) exposed on the document_client class and instead introduced a RetryOptions class exposing the RetryOptions property on ConnectionPolicy class that can be used to override some of the default retry options.

1.8.0

- Added the support for geo-replicated database accounts.
- Test fixes to move the global host and masterKey into the individual test classes.

1.7.0

- Added the support for Time To Live(TTL) feature for documents.

1.6.1

- Bug fixes related to server-side partitioning to allow special characters in partition key path.

1.6.0

- Added the support for server-side partitioned collections feature.

1.5.0

- Added Client-side sharding framework to the SDK. Implemented HashPartionResolver and RangePartitionResolver classes.

1.4.2

- Implement Upsert. New UpsertXXX methods added to support Upsert feature.
- Implement ID-Based Routing. No public API changes, all changes internal.

1.3.0

- Release skipped to bring version number in alignment with other SDKs

1.2.0

- Supports GeoSpatial index.
- Validates ID property for all resources. Ids for resources cannot contain `?, /, #, \\` characters or end with a space.
- Adds new header "index transformation progress" to ResourceResponse.

1.1.0

- Implements V2 indexing policy

1.0.1

- Supports proxy connection

Release & retirement dates

Microsoft provides notification at least **12 months** in advance of retiring an SDK in order to smooth the transition to a newer/supported version. New features and functionality and optimizations are only added to the current SDK, as such it is recommended that you always upgrade to the latest SDK version as early as possible.

> **WARNING**
>
> After 31 August 2022, Azure Cosmos DB will no longer make bug fixes, add new features, and provide support to versions 1.x or 2.x of the Azure Cosmos DB Python SDK for SQL API. If you prefer not to upgrade, requests sent from version 1.x and 2.x of the SDK will continue to be served by the Azure Cosmos DB service.

VERSION	RELEASE DATE	RETIREMENT DATE
4.0.0	May 20, 2020	---

VERSION	RELEASE DATE	RETIREMENT DATE
3.0.2	Nov 15, 2018	---
3.0.1	Oct 04, 2018	---
2.3.3	Sept 08, 2018	August 30, 2020
2.3.2	May 08, 2018	August 30, 2020
2.3.1	December 21, 2017	August 30, 2020
2.3.0	November 10, 2017	August 30, 2020
2.2.1	Sep 29, 2017	August 30, 2020
2.2.0	May 10, 2017	August 30, 2020
2.1.0	May 01, 2017	August 30, 2020
2.0.1	October 30, 2016	August 30, 2020
2.0.0	September 29, 2016	August 30, 2020
1.9.0	July 07, 2016	August 30, 2020
1.8.0	June 14, 2016	August 30, 2020
1.7.0	April 26, 2016	August 30, 2020
1.6.1	April 08, 2016	August 30, 2020
1.6.0	March 29, 2016	August 30, 2020
1.5.0	January 03, 2016	August 30, 2020
1.4.2	October 06, 2015	August 30, 2020
1.4.1	October 06, 2015	August 30, 2020
1.2.0	August 06, 2015	August 30, 2020
1.1.0	July 09, 2015	August 30, 2020
1.0.1	May 25, 2015	August 30, 2020
1.0.0	April 07, 2015	August 30, 2020
0.9.4-prelease	January 14, 2015	February 29, 2016
0.9.3-prelease	December 09, 2014	February 29, 2016

VERSION	RELEASE DATE	RETIREMENT DATE
0.9.2-prelease	November 25, 2014	February 29, 2016
0.9.1-prelease	September 23, 2014	February 29, 2016
0.9.0-prelease	August 21, 2014	February 29, 2016

FAQ

How will I be notified of the retiring SDK?

Microsoft will provide 12 month's advance notice before the end of support of the retiring SDK to facilitate a smooth transition to a supported SDK. We'll notify you through various communication channels: the Azure portal, Azure updates, and direct communication to assigned service administrators.

Can I author applications by using a to-be-retired Azure Cosmos DB SDK during the 12-month period?

Yes, you'll be able to author, deploy, and modify applications by using the to-be-retired Azure Cosmos DB SDK during the 12-month notice period. We recommend that you migrate to a newer supported version of the Azure Cosmos DB SDK during the 12-month notice period, as appropriate.

After the retirement date, what happens to applications that use the unsupported Azure Cosmos DB SDK?

After the retirement date, Azure Cosmos DB will no longer make bug fixes, add new features, or provide support to the retired SDK versions. If you prefer not to upgrade, requests sent from the retired versions of the SDK will continue to be served by the Azure Cosmos DB service.

Which SDK versions will have the latest features and updates?

New features and updates will be added only to the latest minor version of the latest supported major SDK version. We recommend that you always use the latest version to take advantage of new features, performance improvements, and bug fixes. If you're using an old, non-retired version of the SDK, your requests to Azure Cosmos DB will still function, but you won't have access to any new capabilities.

What should I do if I can't update my application before a cutoff date?

We recommend that you upgrade to the latest SDK as early as possible. After an SDK is tagged for retirement, you'll have 12 months to update your application. If you're not able to update by the retirement date, requests sent from the retired versions of the SDK will continue to be served by Azure Cosmos DB, so your running applications will continue to function. But Azure Cosmos DB will no longer make bug fixes, add new features, or provide support to the retired SDK versions.

If you have a support plan and require technical support, contact us by filing a support ticket.

Next steps

To learn more about Cosmos DB, see Microsoft Azure Cosmos DB service page.

Azure Policy built-in definitions for Azure Cosmos DB

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: SQL API Cassandra API Gremlin API Table API Azure Cosmos DB API for MongoDB

This page is an index of Azure Policy built-in policy definitions for Azure Cosmos DB. For additional Azure Policy built-ins for other services, see Azure Policy built-in definitions.

The name of each built-in policy definition links to the policy definition in the Azure portal. Use the link in the **Version** column to view the source on the Azure Policy GitHub repo.

Azure Cosmos DB

NAME (AZURE PORTAL)	DESCRIPTION	EFFECT(S)	VERSION (GITHUB)
Azure Cosmos DB account should use customer-managed keys to encrypt data at rest	Use customer-managed keys to control the encryption at rest of the data stored in Azure Cosmos DB when this is a regulatory or compliance requirement. Customer-managed keys also deliver double encryption by adding a second layer of encryption on top of the default one done with service-managed keys. See https://aka.ms/cosmosdb-cmk	audit, deny, disabled	1.0.0
Azure Cosmos DB accounts should have firewall rules	Audit or deny resources that do not have any IP rules configured and allow all networks by default. Accounts that have at least one IP rule defined with the virtual network filter enabled are deemed compliant. Accounts disabling public access are also deemed compliant.	Audit, Deny, Disabled	1.0.0
Azure Cosmos DB allowed locations	This policy enables you to restrict the locations your organization can specify when deploying Azure Cosmos DB resources. Use to enforce your geo-compliance requirements.	[parameters('policyEffect')]	1.0.0
Azure Cosmos DB key based metadata write access should be disabled	This policy enables you to ensure all Azure Cosmos DB accounts disable key based metadata write access.	append	1.0.0

NAME	DESCRIPTION	EFFECT(S)	VERSION
Azure Cosmos DB throughput should be limited	This policy enables you to restrict the maximum throughput your organization can specify when creating Azure Cosmos DB databases and containers through the resource provider. It blocks the creation of autoscale resources.	audit, deny, disabled	1.0.0
Cosmos DB should use a virtual network service endpoint	This policy audits any Cosmos DB not configured to use a virtual network service endpoint.	Audit, Disabled	1.0.0
Deploy Advanced Threat Protection for Cosmos DB Accounts	This policy enables Advanced Threat Protection across Cosmos DB accounts.	DeployIfNotExists, Disabled	1.0.0

Next steps

- See the built-ins on the Azure Policy GitHub repo.
- Review the Azure Policy definition structure.
- Review Understanding policy effects.

Introduction to the Azure Cosmos DB Cassandra API

12/4/2020 • 3 minutes to read • Edit Online

APPLIES TO: Cassandra API

Azure Cosmos DB Cassandra API can be used as the data store for apps written for Apache Cassandra. This means that by using existing Apache drivers compliant with CQLv4, your existing Cassandra application can now communicate with the Azure Cosmos DB Cassandra API. In many cases, you can switch from using Apache Cassandra to using Azure Cosmos DB's Cassandra API, by just changing a connection string.

The Cassandra API enables you to interact with data stored in Azure Cosmos DB using the Cassandra Query Language (CQL) , Cassandra-based tools (like cqlsh) and Cassandra client drivers that you're already familiar with.

> **NOTE**
>
> The serverless capacity mode is now available on Azure Cosmos DB's Cassandra API.

What is the benefit of using Apache Cassandra API for Azure Cosmos DB?

No operations management: As a fully managed cloud service, Azure Cosmos DB Cassandra API removes the overhead of managing and monitoring a myriad of settings across OS, JVM, and yaml files and their interactions. Azure Cosmos DB provides monitoring of throughput, latency, storage, availability, and configurable alerts.

Open source standard: Despite being a fully managed service, Cassandra API still supports a large surface area of the native Apache Cassandra wire protocol, allowing you to build applications on a widely used and cloud agnostic open source standard.

Performance management: Azure Cosmos DB provides guaranteed low latency reads and writes at the 99th percentile, backed up by the SLAs. Users do not have to worry about operational overhead to ensure high performance and low latency reads and writes. This means that users do not need to deal with scheduling compaction, managing tombstones, setting up bloom filters and replicas manually. Azure Cosmos DB removes the overhead to manage these issues and lets you focus on the application logic.

Ability to use existing code and tools: Azure Cosmos DB provides wire protocol level compatibility with existing Cassandra SDKs and tools. This compatibility ensures you can use your existing codebase with Azure Cosmos DB Cassandra API with trivial changes.

Throughput and storage elasticity: Azure Cosmos DB provides throughput across all regions and can scale the provisioned throughput with Azure portal, PowerShell, or CLI operations. You can elastically scale storage and throughput for your tables as needed with predictable performance.

Global distribution and availability: Azure Cosmos DB provides the ability to globally distribute data across all Azure regions and serve the data locally while ensuring low latency data access and high availability. Azure Cosmos DB provides 99.99% high availability within a region and 99.999% read and write availability across multiple regions with no operations overhead. Learn more in Distribute data globally article.

Choice of consistency: Azure Cosmos DB provides the choice of five well-defined consistency levels to achieve optimal tradeoffs between consistency and performance. These consistency levels are strong, bounded-staleness, session, consistent prefix and eventual. These well-defined, practical, and intuitive consistency levels allow developers to make precise trade-offs between consistency, availability, and latency. Learn more in consistency

levels article.

Enterprise grade: Azure cosmos DB provides compliance certifications to ensure users can use the platform securely. Azure Cosmos DB also provides encryption at rest and in motion, IP firewall, and audit logs for control plane activities.

Event Sourcing: Cassandra API provides access to a persistent change log, the Change Feed, which can facilitate event sourcing directly from the database. In Apache Cassandra, the only equivalent is change data capture (CDC), which is merely a mechanism to flag specific tables for archival as well as rejecting writes to those tables once a configurable size-on-disk for the CDC log is reached (these capabilities are redundant in Cosmos DB as the relevant aspects are automatically governed).

Next steps

- You can quickly get started with building the following language-specific apps to create and manage Cassandra API data:
  - Node.js app
  - .NET app
  - Python app
- Get started with creating a Cassandra API account, database, and a table by using a Java application.
- Load sample data to the Cassandra API table by using a Java application.
- Query data from the Cassandra API account by using a Java application.
- To learn about Apache Cassandra features supported by Azure Cosmos DB Cassandra API, see Cassandra support article.
- Read the Frequently Asked Questions.

Apache Cassandra features supported by Azure Cosmos DB Cassandra API

12/4/2020 • 7 minutes to read • Edit Online

APPLIES TO: Cassandra API

Azure Cosmos DB is Microsoft's globally distributed multi-model database service. You can communicate with the Azure Cosmos DB Cassandra API through the CQL Binary Protocol v4 wire protocol compliant open-source Cassandra client drivers.

By using the Azure Cosmos DB Cassandra API, you can enjoy the benefits of the Apache Cassandra APIs as well as the enterprise capabilities that Azure Cosmos DB provides. The enterprise capabilities include global distribution, automatic scale out partitioning, availability and latency guarantees, encryption at rest, backups, and much more.

Cassandra protocol

The Azure Cosmos DB Cassandra API is compatible with Cassandra Query Language (CQL) v3.11 API (backward-compatible with version 2.x). The supported CQL commands, tools, limitations, and exceptions are listed below. Any client driver that understands these protocols should be able to connect to Azure Cosmos DB Cassandra API.

Cassandra driver

The following versions of Cassandra drivers are supported by Azure Cosmos DB Cassandra API:

- Java 3.5+
- C# 3.5+
- Nodejs 3.5+
- Python 3.15+
- C++ 2.9
- PHP 1.3
- Gocql

CQL data types

Azure Cosmos DB Cassandra API supports the following CQL data types:

COMMAND	SUPPORTED
ascii	Yes
bigint	Yes
blob	Yes
boolean	Yes
counter	Yes
date	Yes

COMMAND	SUPPORTED
decimal	Yes
double	Yes
float	Yes
frozen	Yes
inet	Yes
int	Yes
list	Yes
set	Yes
smallint	Yes
text	Yes
time	Yes
timestamp	Yes
timeuuid	Yes
tinyint	Yes
tuple	Yes
uuid	Yes
varchar	Yes
varint	Yes
tuples	Yes
udts	Yes
map	Yes

Static is supported for data type declaration.

CQL functions

Azure Cosmos DB Cassandra API supports the following CQL functions:

COMMAND	SUPPORTED
Token *	Yes
ttl	Yes
writetime	Yes
cast	No

> **NOTE**
> * Cassandra API supports token as a projection/selector, and only allows token(pk) on the left-hand side of a where clause. For example, `WHERE token(pk) > 1024` is supported, but `WHERE token(pk) > token(100)` is **not** supported.

Aggregate functions:

COMMAND	SUPPORTED
avg	Yes
count	Yes
min	Yes
max	Yes
sum	Yes

> **NOTE**
> Aggregate functions work on regular columns, but aggregates on clustering columns are **not** supported.

Blob conversion functions:

COMMAND	SUPPORTED
typeAsBlob(value)	Yes
blobAsType(value)	Yes

UUID and timeuuid functions:

COMMAND	SUPPORTED
dateOf()	Yes
now()	Yes
minTimeuuid()	Yes

COMMAND	SUPPORTED
unixTimestampOf()	Yes
toDate(timeuuid)	Yes
toTimestamp(timeuuid)	Yes
toUnixTimestamp(timeuuid)	Yes
toDate(timestamp)	Yes
toUnixTimestamp(timestamp)	Yes
toTimestamp(date)	Yes
toUnixTimestamp(date)	Yes

CQL commands

Azure Cosmos DB supports the following database commands on Cassandra API accounts.

COMMAND	SUPPORTED
ALLOW FILTERING	Yes
ALTER KEYSPACE	N/A (PaaS service, replication managed internally)
ALTER MATERIALIZED VIEW	No
ALTER ROLE	No
ALTER TABLE	Yes
ALTER TYPE	No
ALTER USER	No
BATCH	Yes (unlogged batch only)
COMPACT STORAGE	N/A (PaaS service)
CREATE AGGREGATE	No
CREATE CUSTOM INDEX (SASI)	No
CREATE INDEX	Yes (without specifying index name, and indexes on clustering keys or full FROZEN collection not supported)
CREATE FUNCTION	No
CREATE KEYSPACE (replication settings ignored)	Yes

COMMAND	SUPPORTED
CREATE MATERIALIZED VIEW	No
CREATE TABLE	Yes
CREATE TRIGGER	No
CREATE TYPE	Yes
CREATE ROLE	No
CREATE USER (Deprecated in native Apache Cassandra)	No
DELETE	Yes
DELETE (lightweight transactions with IF CONDITION)	Yes
DISTINCT	No
DROP AGGREGATE	No
DROP FUNCTION	No
DROP INDEX	Yes
DROP KEYSPACE	Yes
DROP MATERIALIZED VIEW	No
DROP ROLE	No
DROP TABLE	Yes
DROP TRIGGER	No
DROP TYPE	Yes
DROP USER (Deprecated in native Apache Cassandra)	No
GRANT	No
INSERT	Yes
INSERT (lightweight transactions with IF CONDITION)	Yes
LIST PERMISSIONS	No
LIST ROLES	No
LIST USERS (Deprecated in native Apache Cassandra)	No

COMMAND	SUPPORTED
REVOKE	No
SELECT	Yes
SELECT (lightweight transactions with IF CONDITION)	No
UPDATE	Yes
UPDATE (lightweight transactions with IF CONDITION)	No
TRUNCATE	No
USE	Yes

JSON Support

COMMAND	SUPPORTED
SELECT JSON	Yes
INSERT JSON	Yes
fromJson()	No
toJson()	No

Cassandra API limits

Azure Cosmos DB Cassandra API does not have any limits on the size of data stored in a table. Hundreds of terabytes or Petabytes of data can be stored while ensuring partition key limits are honored. Similarly, every entity or row equivalent does not have any limits on the number of columns. However, the total size of the entity should not exceed 2 MB. The data per partition key cannot exceed 20 GB as in all other APIs.

Tools

Azure Cosmos DB Cassandra API is a managed service platform. It does not require any management overhead or utilities such as Garbage Collector, Java Virtual Machine(JVM), and nodetool to manage the cluster. It supports tools such as cqlsh that utilizes Binary CQLv4 compatibility.

- Azure portal's data explorer, metrics, log diagnostics, PowerShell, and CLI are other supported mechanisms to manage the account.

Hosted CQL shell (preview)

You can open a hosted native Cassandra shell (CQLSH v5.0.1) directly from the Data Explorer in the Azure portal or the Azure Cosmos DB Explorer. Before enabling the CQL shell, you must enable the Notebooks feature in your account (if not already enabled, you will be prompted when clicking on `Open Cassandra Shell`). Check the highlighted note in Enable notebooks for Azure Cosmos DB accounts for supported Azure Regions.

You can also connect to the Cassandra API in Azure Cosmos DB by using the CQLSH installed on a local machine. It comes with Apache Cassandra 3.1.1 and works out of the box by setting the environment variables. The following sections include the instructions to install, configure, and connect to Cassandra API in Azure Cosmos DB, on Windows or Linux using CQLSH.

> **NOTE**
> Connections to Azure Cosmos DB Cassandra API will not work with DataStax Enterprise (DSE) versions of CQLSH. Please ensure you use only the open source Apache Cassandra versions of CQLSH when connecting to Cassandra API.

Windows:

If using windows, we recommend you enable the Windows filesystem for Linux. You can then follow the linux commands below.

Unix/Linux/Mac:

```
# Install default-jre and default-jdk
sudo apt install default-jre
sudo apt-get update
sudo apt install default-jdk

# Import the Baltimore CyberTrust root certificate:
curl https://cacert.omniroot.com/bc2025.crt > bc2025.crt
keytool -importcert -alias bc2025ca -file bc2025.crt

# Install the Cassandra libraries in order to get CQLSH:
echo "deb http://www.apache.org/dist/cassandra/debian 311x main" | sudo tee -a
/etc/apt/sources.list.d/cassandra.sources.list
curl https://downloads.apache.org/cassandra/KEYS | sudo apt-key add -
sudo apt-get update
sudo apt-get install cassandra

# Export the SSL variables:
export SSL_VERSION=TLSv1_2
export SSL_VALIDATE=false

# Connect to Azure Cosmos DB API for Cassandra:
cqlsh <YOUR_ACCOUNT_NAME>.cassandra.cosmosdb.azure.com 10350 -u <YOUR_ACCOUNT_NAME> -p <YOUR_ACCOUNT_PASSWORD>
--ssl
```

All CRUD operations that are executed through a CQL v4 compatible SDK will return extra information about error and request units consumed. The DELETE and UPDATE commands should be handled with resource governance taken into consideration, to ensure the most efficient use of the provisioned throughput.

- Note gc_grace_seconds value must be zero if specified.

```
var tableInsertStatement = table.Insert(sampleEntity);
var insertResult = await tableInsertStatement.ExecuteAsync();

foreach (string key in insertResult.Info.IncomingPayload)
        {
            byte[] valueInBytes = customPayload[key];
            double value = Encoding.UTF8.GetString(valueInBytes);
            Console.WriteLine($"CustomPayload:  {key}: {value}");
        }
```

Consistency mapping

Azure Cosmos DB Cassandra API provides choice of consistency for read operations. The consistency mapping is detailed here.

Permission and role management

Azure Cosmos DB supports Azure role-based access control (Azure RBAC) for provisioning, rotating keys, viewing metrics and read-write and read-only passwords/keys that can be obtained through the Azure portal. Azure Cosmos DB does not support roles for CRUD activities.

Keyspace and Table options

The options for region name, class, replication_factor, and datacenter in the "Create Keyspace" command are ignored currently. The system uses the underlying Azure Cosmos DB's global distribution replication method to add the regions. If you need the cross-region presence of data, you can enable it at the account level with PowerShell, CLI, or portal, to learn more, see the how to add regions article. Durable_writes can't be disabled because Azure Cosmos DB ensures every write is durable. In every region, Azure Cosmos DB replicates the data across the replica set that is made up of four replicas and this replica set configuration can't be modified.

All the options are ignored when creating the table, except gc_grace_seconds, which should be set to zero. The Keyspace and table have an extra option named "cosmosdb_provisioned_throughput" with a minimum value of 400 RU/s. The Keyspace throughput allows sharing throughput across multiple tables and it is useful for scenarios when all tables are not utilizing the provisioned throughput. Alter Table command allows changing the provisioned throughput across the regions.

```
CREATE  KEYSPACE  sampleks WITH REPLICATION = {  'class' : 'SimpleStrategy'}   AND
cosmosdb_provisioned_throughput=2000;

CREATE TABLE sampleks.t1(user_id int PRIMARY KEY, lastname text) WITH cosmosdb_provisioned_throughput=2000;

ALTER TABLE gks1.t1 WITH cosmosdb_provisioned_throughput=10000 ;
```

Secondary Index

Cassandra API supports secondary indexes on all data types except frozen collection types, decimal and variant types.

Usage of Cassandra retry connection policy

Azure Cosmos DB is a resource governed system. This means you can do a certain number of operations in a given second based on the request units consumed by the operations. If an application exceeds that limit in a given second, requests are rate-limited and exceptions will be thrown. The Cassandra API in Azure Cosmos DB translates these exceptions to overloaded errors on the Cassandra native protocol. To ensure that your application can intercept and retry requests in case of rate limitation, the spark and the Java extensions are provided. See also Java code samples for version 3 and version 4 Datastax drivers, when connecting to Cassandra API in Azure Cosmos DB. If you use other SDKs to access Cassandra API in Azure Cosmos DB, create a connection policy to retry on these exceptions.

Next steps

- Get started with creating a Cassandra API account, database, and a table by using a Java application

Quickstart: Build a Cassandra app with .NET SDK and Azure Cosmos DB

11/2/2020 • 6 minutes to read • Edit Online

APPLIES TO: Cassandra API

This quickstart shows how to use .NET and the Azure Cosmos DB Cassandra API to build a profile app by cloning an example from GitHub. This quickstart also shows you how to use the web-based Azure portal to create an Azure Cosmos DB account.

Azure Cosmos DB is Microsoft's globally distributed multi-model database service. You can quickly create and query document, table, key-value, and graph databases, all of which benefit from the global distribution and horizontal scale capabilities at the core of Azure Cosmos DB.

Prerequisites

If you don't have an Azure subscription, create a free account before you begin. Alternatively, you can Try Azure Cosmos DB for free without an Azure subscription, free of charge and commitments.

In addition, you need:

- If you don't already have Visual Studio 2019 installed, you can download and use the **free** Visual Studio 2019 Community Edition. Make sure that you enable **Azure development** during the Visual Studio setup.
- Install Git to clone the example.

Create a database account

1. In a new browser window, sign in to the Azure portal.
2. In the left menu, select **Create a resource**.

3. On the **New** page, select **Databases** > **Azure Cosmos DB**.

4. On the **Create Azure Cosmos DB Account** page, enter the settings for the new Azure Cosmos DB account.

SETTING	VALUE	DESCRIPTION
Subscription	Your subscription	Select the Azure subscription that you want to use for this Azure Cosmos DB account.
Resource Group	Create new Then enter the same name as Account Name	Select **Create new**. Then enter a new resource group name for your account. For simplicity, use the same name as your Azure Cosmos account name.
Account Name	Enter a unique name	Enter a unique name to identify your Azure Cosmos DB account. Your account URI will be *cassandra.cosmos.azure.com* appended to your unique account name. The account name can use only lowercase letters, numbers, and hyphens (-), and must be between 3 and 31 characters long.

SETTING	VALUE	DESCRIPTION
API	Cassandra	The API determines the type of account to create. Azure Cosmos DB provides five APIs: Core (SQL) for document databases, Gremlin for graph databases, MongoDB for document databases, Azure Table, and Cassandra. You must create a separate account for each API. Select **Cassandra**, because in this quickstart you are creating a table that works with the Cassandra API. Learn more about the Cassandra API.
Location	Select the region closest to your users	Select a geographic location to host your Azure Cosmos DB account. Use the location that's closest to your users to give them the fastest access to the data.
Capacity mode	Provisioned throughput or Serverless	Select **Provisioned throughput** to create an account in provisioned throughput mode. Select **Serverless** to create an account in serverless mode.

Select **Review+Create**. You can skip the **Networking**, **Backup**, **Encryption** and **Tags** section.

5. The account creation takes a few minutes. Wait for the portal to display the page saying **Congratulations! Your Azure Cosmos DB account was created**.

Clone the sample application

Now let's switch to working with code. Let's clone a Cassandra API app from GitHub, set the connection string, and run it. You'll see how easy it is to work with data programmatically.

1. Open a command prompt. Create a new folder named `git-samples`. Then, close the command prompt.

```
md "C:\git-samples"
```

2. Open a git terminal window, such as git bash, and use the `cd` command to change to the new folder to install the sample app.

```
cd "C:\git-samples"
```

3. Run the following command to clone the sample repository. This command creates a copy of the sample app on your computer.

```
git clone https://github.com/Azure-Samples/azure-cosmos-db-cassandra-dotnet-getting-started.git
```

4. Next, open the CassandraQuickStartSample solution file in Visual Studio.

Review the code

This step is optional. If you're interested to learn how the code creates the database resources, you can review the following snippets. The snippets are all taken from the `Program.cs` file installed in the `C:\git-samples\azure-cosmos-db-cassandra-dotnet-getting-started\CassandraQuickStartSample` folder. Otherwise, you can skip ahead to Update your connection string.

- Initialize the session by connecting to a Cassandra cluster endpoint. The Cassandra API on Azure Cosmos DB supports only TLSv1.2.

```
var options = new Cassandra.SSLOptions(SslProtocols.Tls12, true, ValidateServerCertificate);
options.SetHostNameResolver((ipAddress) => CassandraContactPoint);
Cluster cluster = Cluster.Builder().WithCredentials(UserName,
Password).WithPort(CassandraPort).AddContactPoint(CassandraContactPoint).WithSSL(options).Build();
ISession session = cluster.Connect();
```

- Create a new keyspace.

```
session.Execute("CREATE KEYSPACE uprofile WITH REPLICATION = { 'class' : 'NetworkTopologyStrategy',
'datacenter1' : 1 };");
```

- Create a new table.

```
session.Execute("CREATE TABLE IF NOT EXISTS uprofile.user (user_id int PRIMARY KEY, user_name text,
user_bcity text)");
```

- Insert user entities by using the IMapper object with a new session that connects to the uprofile keyspace.

```
mapper.Insert<User>(new User(1, "LyubovK", "Dubai"));
```

- Query to get all user's information.

```
foreach (User user in mapper.Fetch<User>("Select * from user"))
{
  Console.WriteLine(user);
}
```

- Query to get a single user's information.

```
mapper.FirstOrDefault<User>("Select * from user where user_id = ?", 3);
```

Update your connection string

Now go back to the Azure portal to get your connection string information and copy it into the app. The connection string information enables your app to communicate with your hosted database.

1. In the Azure portal, select **Connection String**.

2. Use the button on the right side of the screen to copy the USERNAME value.

3. In Visual Studio, open the Program.cs file.

4. Paste the USERNAME value from the portal over `<FILLME>` on line 13.

   Line 13 of Program.cs should now look similar to

   ```
   private const string UserName = "cosmos-db-quickstart";
   ```

5. Go back to portal and copy the PASSWORD value. Paste the PASSWORD value from the portal over `<FILLME>` on line 14.

   Line 14 of Program.cs should now look similar to

   ```
   private const string Password = "2Ggkr662ifxz2Mg...==";
   ```

6. Go back to portal and copy the CONTACT POINT value. Paste the CONTACT POINT value from the portal over `<FILLME>` on line 15.

   Line 15 of Program.cs should now look similar to

   ```
   private const string CassandraContactPoint = "cosmos-db-quickstarts.cassandra.cosmosdb.azure.com"; //
   DnsName
   ```

7. Save the Program.cs file.

Run the .NET app

1. In Visual Studio, select **Tools** > **NuGet Package Manager** > **Package Manager Console**.

2. At the command prompt, use the following command to install the .NET Driver's NuGet package.

   ```
   Install-Package CassandraCSharpDriver
   ```

3. Press CTRL + F5 to run the application. Your app displays in your console window.

```
created keyspace uprofile
created table user
Inserted data into user table
Select ALL
------------------------------
 1 | LyubovK | Dubai
 2 | JiriK | Toronto
 3 | IvanH | Mumbai
 4 | LiliyaB | Seattle
 5 | JindrichH | Buenos Aires
Getting by id 3
------------------------------
 3 | IvanH | Mumbai
```

Press CTRL + C to stop execution of the program and close the console window.

4. In the Azure portal, open **Data Explorer** to query, modify, and work with this new data.

Review SLAs in the Azure portal

The Azure portal monitors your Cosmos DB account throughput, storage, availability, latency, and consistency. Charts for metrics associated with an Azure Cosmos DB Service Level Agreement (SLA) show the SLA value compared to actual performance. This suite of metrics makes monitoring your SLAs transparent.

To review metrics and SLAs:

1. Select **Metrics** in your Cosmos DB account's navigation menu.

2. Select a tab such as **Latency**, and select a timeframe on the right. Compare the **Actual** and **SLA** lines on the charts.

3. Review the metrics on the other tabs.

Clean up resources

When you're done with your app and Azure Cosmos DB account, you can delete the Azure resources you created so you don't incur more charges. To delete the resources:

1. In the Azure portal Search bar, search for and select **Resource groups**.

2. From the list, select the resource group you created for this quickstart.

3. On the resource group **Overview** page, select **Delete resource group**.

4. In the next window, enter the name of the resource group to delete, and then select **Delete**.

Next steps

In this quickstart, you've learned how to create an Azure Cosmos DB account, create a container using the Data Explorer, and run a web app. You can now import additional data to your Cosmos DB account.

Import Cassandra data into Azure Cosmos DB

Quickstart: Build a Cassandra app with .NET Core and Azure Cosmos DB

11/2/2020 • 6 minutes to read • Edit Online

APPLIES TO: Cassandra API

This quickstart shows how to use .NET Core and the Azure Cosmos DB Cassandra API to build a profile app by cloning an example from GitHub. This quickstart also shows you how to use the web-based Azure portal to create an Azure Cosmos DB account.

Azure Cosmos DB is Microsoft's globally distributed multi-model database service. You can quickly create and query document, table, key-value, and graph databases, all of which benefit from the global distribution and horizontal scale capabilities at the core of Azure Cosmos DB.

Prerequisites

If you don't have an Azure subscription, create a free account before you begin. Alternatively, you can Try Azure Cosmos DB for free without an Azure subscription, free of charge and commitments.

In addition, you need:

- If you don't already have Visual Studio 2019 installed, you can download and use the **free** Visual Studio 2019 Community Edition. Make sure that you enable **Azure development** during the Visual Studio setup.
- Install Git to clone the example.

Create a database account

1. In a new browser window, sign in to the Azure portal.
2. In the left menu, select **Create a resource**.

3. On the **New** page, select **Databases** > **Azure Cosmos DB**.

4. On the **Create Azure Cosmos DB Account** page, enter the settings for the new Azure Cosmos DB account.

SETTING	VALUE	DESCRIPTION
Subscription	Your subscription	Select the Azure subscription that you want to use for this Azure Cosmos DB account.
Resource Group	Create new Then enter the same name as Account Name	Select **Create new**. Then enter a new resource group name for your account. For simplicity, use the same name as your Azure Cosmos account name.
Account Name	Enter a unique name	Enter a unique name to identify your Azure Cosmos DB account. Your account URI will be *cassandra.cosmos.azure.com* appended to your unique account name. The account name can use only lowercase letters, numbers, and hyphens (-), and must be between 3 and 31 characters long.

SETTING	VALUE	DESCRIPTION
API	Cassandra	The API determines the type of account to create. Azure Cosmos DB provides five APIs: Core (SQL) for document databases, Gremlin for graph databases, MongoDB for document databases, Azure Table, and Cassandra. You must create a separate account for each API. Select **Cassandra**, because in this quickstart you are creating a table that works with the Cassandra API. Learn more about the Cassandra API.
Location	Select the region closest to your users	Select a geographic location to host your Azure Cosmos DB account. Use the location that's closest to your users to give them the fastest access to the data.
Capacity mode	Provisioned throughput or Serverless	Select **Provisioned throughput** to create an account in provisioned throughput mode. Select **Serverless** to create an account in serverless mode.

Select **Review+Create**. You can skip the **Networking**, **Backup**, **Encryption** and **Tags** section.

5. The account creation takes a few minutes. Wait for the portal to display the page saying **Congratulations! Your Azure Cosmos DB account was created**.

Clone the sample application

Now let's switch to working with code. Let's clone a Cassandra API app from GitHub, set the connection string, and run it. You'll see how easy it is to work with data programmatically.

1. Open a command prompt. Create a new folder named `git-samples`. Then, close the command prompt.

```
md "C:\git-samples"
```

2. Open a git terminal window, such as git bash, and use the `cd` command to change to the new folder to install the sample app.

```
cd "C:\git-samples"
```

3. Run the following command to clone the sample repository. This command creates a copy of the sample app on your computer.

```
git clone https://github.com/Azure-Samples/azure-cosmos-db-cassandra-dotnet-core-getting-started.git
```

4. Next, open the CassandraQuickStartSample solution file in Visual Studio.

Review the code

This step is optional. If you're interested to learn how the code creates the database resources, you can review the following snippets. The snippets are all taken from the `Program.cs` file within `async Task ProcessAsync()` method, installed in the `C:\git-samples\azure-cosmos-db-cassandra-dotnet-core-getting-started\CassandraQuickStart` folder. Otherwise, you can skip ahead to Update your connection string.

- Initialize the session by connecting to a Cassandra cluster endpoint. The Cassandra API on Azure Cosmos DB supports only TLSv1.2.

```
    var options = new Cassandra.SSLOptions(SslProtocols.Tls12, true, ValidateServerCertificate);
    options.SetHostNameResolver((ipAddress) => CASSANDRACONTACTPOINT);
    Cluster cluster = Cluster
        .Builder()
        .WithCredentials(USERNAME, PASSWORD)
        .WithPort(CASSANDRAPORT)
        .AddContactPoint(CASSANDRACONTACTPOINT)
        .WithSSL(options)
        .Build()
    ;
    ISession session = await cluster.ConnectAsync();
```

- Drop existing keyspace if it already exists.

```
await session.ExecuteAsync(new SimpleStatement("DROP KEYSPACE IF EXISTS uprofile"));
```

- Create a new keyspace.

```
await session.ExecuteAsync(new SimpleStatement("CREATE KEYSPACE uprofile WITH REPLICATION = { 'class' :
'NetworkTopologyStrategy', 'datacenter1' : 1 };"));
```

- Create a new table.

```
await session.ExecuteAsync(new SimpleStatement("CREATE TABLE IF NOT EXISTS uprofile.user (user_id int
PRIMARY KEY, user_name text, user_bcity text)"));
```

- Insert user entities by using the IMapper object with a new session that connects to the uprofile keyspace.

```
await mapper.InsertAsync<User>(new User(1, "LyubovK", "Dubai"));
```

- Query to get all user's information.

```
foreach (User user in await mapper.FetchAsync<User>("Select * from user"))
{
    Console.WriteLine(user);
}
```

- Query to get a single user's information.

```
mapper.FirstOrDefault<User>("Select * from user where user_id = ?", 3);
```

Update your connection string

Now go back to the Azure portal to get your connection string information and copy it into the app. The connection string information enables your app to communicate with your hosted database.

1. In the Azure portal, select **Connection String**.

2. Use the button on the right side of the screen to copy the USERNAME value.

3. In Visual Studio, open the Program.cs file.

4. Paste the USERNAME value from the portal over `<PROVIDE>` on line 13.

   Line 13 of Program.cs should now look similar to

   ```
   private const string UserName = "cosmos-db-quickstart";
   ```

   You can also paste the same value over `<PROVIDE>` on line 15 for the CONTACT POINT value:

   ```
   private const string CassandraContactPoint = "cosmos-db-quickstarts.cassandra.cosmosdb.azure.com"; //
   DnsName
   ```

5. Go back to portal and copy the PASSWORD value. Paste the PASSWORD value from the portal over `<PROVIDE>` on line 14.

   Line 14 of Program.cs should now look similar to

   ```
   private const string Password = "2Ggkr662ifxz2Mg...==";
   ```

6. Go back to portal and copy the CONTACT POINT value. Paste the CONTACT POINT value from the portal over `<PROVIDE>` on line 16.

   Line 16 of Program.cs should now look similar to

   ```
   private const string CASSANDRACONTACTPOINT = "quickstart-cassandra-api.cassandra.cosmos.azure.com";
   ```

7. Save the Program.cs file.

Run the .NET Core app

1. In Visual Studio, select **Tools** > **NuGet Package Manager** > **Package Manager Console**.

2. At the command prompt, use the following command to install the .NET Driver's NuGet package.

   ```
   Install-Package CassandraCSharpDriver
   ```

3. Press CTRL + F5 to run the application. Your app displays in your console window.

   ```
   created keyspace uprofile
   created table user
   Inserted data into user table
   Select ALL
   -------------------------------
    1 | LyubovK | Dubai
    2 | JiriK | Toronto
    3 | IvanH | Mumbai
    4 | LiliyaB | Seattle
    5 | JindrichH | Buenos Aires
   Getting by id 3
   -------------------------------
    3 | IvanH | Mumbai
   ```

   Press CTRL + C to stop execution of the program and close the console window.

4. In the Azure portal, open **Data Explorer** to query, modify, and work with this new data.

Review SLAs in the Azure portal

The Azure portal monitors your Cosmos DB account throughput, storage, availability, latency, and consistency. Charts for metrics associated with an Azure Cosmos DB Service Level Agreement (SLA) show the SLA value compared to actual performance. This suite of metrics makes monitoring your SLAs transparent.

To review metrics and SLAs:

1. Select **Metrics** in your Cosmos DB account's navigation menu.
2. Select a tab such as **Latency**, and select a timeframe on the right. Compare the **Actual** and **SLA** lines on the charts.

3. Review the metrics on the other tabs.

Clean up resources

When you're done with your app and Azure Cosmos DB account, you can delete the Azure resources you created so you don't incur more charges. To delete the resources:

1. In the Azure portal Search bar, search for and select **Resource groups**.
2. From the list, select the resource group you created for this quickstart.

3. On the resource group **Overview** page, select **Delete resource group**.

4. In the next window, enter the name of the resource group to delete, and then select **Delete**.

Next steps

In this quickstart, you've learned how to create an Azure Cosmos DB account, create a container using the Data Explorer, and run a web app. You can now import additional data to your Cosmos DB account.

Import Cassandra data into Azure Cosmos DB

Quickstart: Build a Java app to manage Azure Cosmos DB Cassandra API data (v3 Driver)

11/2/2020 • 7 minutes to read • Edit Online

APPLIES TO: Cassandra API

In this quickstart, you create an Azure Cosmos DB Cassandra API account, and use a Cassandra Java app cloned from GitHub to create a Cassandra database and container using the v3.x Apache Cassandra drivers for Java. Azure Cosmos DB is a multi-model database service that lets you quickly create and query document, table, key-value, and graph databases with global distribution and horizontal scale capabilities.

Prerequisites

- An Azure account with an active subscription. Create one for free. Or try Azure Cosmos DB for free without an Azure subscription.
- Java Development Kit (JDK) 8. Point your `JAVA_HOME` environment variable to the folder where the JDK is installed.
- A Maven binary archive. On Ubuntu, run `apt-get install maven` to install Maven.
- Git. On Ubuntu, run `sudo apt-get install git` to install Git.

Create a database account

Before you can create a document database, you need to create a Cassandra account with Azure Cosmos DB.

1. In a new browser window, sign in to the Azure portal.

2. In the left menu, select **Create a resource**.

3. On the **New** page, select **Databases** > **Azure Cosmos DB**.

4. On the **Create Azure Cosmos DB Account** page, enter the settings for the new Azure Cosmos DB account.

SETTING	VALUE	DESCRIPTION
Subscription	Your subscription	Select the Azure subscription that you want to use for this Azure Cosmos DB account.
Resource Group	Create new Then enter the same name as Account Name	Select **Create new**. Then enter a new resource group name for your account. For simplicity, use the same name as your Azure Cosmos account name.
Account Name	Enter a unique name	Enter a unique name to identify your Azure Cosmos DB account. Your account URI will be *cassandra.cosmos.azure.com* appended to your unique account name. The account name can use only lowercase letters, numbers, and hyphens (-), and must be between 3 and 31 characters long.

SETTING	VALUE	DESCRIPTION
API	Cassandra	The API determines the type of account to create. Azure Cosmos DB provides five APIs: Core (SQL) for document databases, Gremlin for graph databases, MongoDB for document databases, Azure Table, and Cassandra. You must create a separate account for each API. Select **Cassandra**, because in this quickstart you are creating a table that works with the Cassandra API. Learn more about the Cassandra API.
Location	Select the region closest to your users	Select a geographic location to host your Azure Cosmos DB account. Use the location that's closest to your users to give them the fastest access to the data.
Capacity mode	Provisioned throughput or Serverless	Select **Provisioned throughput** to create an account in provisioned throughput mode. Select **Serverless** to create an account in serverless mode.

Select **Review+Create**. You can skip the **Networking**, **Backup**, **Encryption** and **Tags** section.

5. The account creation takes a few minutes. Wait for the portal to display the page saying **Congratulations! Your Azure Cosmos DB account was created**.

Clone the sample application

Now let's switch to working with code. Let's clone a Cassandra app from GitHub, set the connection string, and run it. You'll see how easy it is to work with data programmatically.

1. Open a command prompt. Create a new folder named `git-samples`. Then, close the command prompt.

```
md "C:\git-samples"
```

2. Open a git terminal window, such as git bash, and use the `cd` command to change to the new folder to install the sample app.

```
cd "C:\git-samples"
```

3. Run the following command to clone the sample repository. This command creates a copy of the sample app on your computer.

```
git clone https://github.com/Azure-Samples/azure-cosmos-db-cassandra-java-getting-started.git
```

Review the code

This step is optional. If you're interested to learn how the code creates the database resources, you can review the following snippets. Otherwise, you can skip ahead to Update your connection string. These snippets are all taken from the *src/main/java/com/azure/cosmosdb/cassandra/util/CassandraUtils.java* file.

- The Cassandra host, port, user name, password, and TLS/SSL options are set. The connection string information comes from the connection string page in the Azure portal.

```
cluster =
Cluster.builder().addContactPoint(cassandraHost).withPort(cassandraPort).withCredentials(cassandraUsern
ame, cassandraPassword).withSSL(sslOptions).build();
```

- The `cluster` connects to the Azure Cosmos DB Cassandra API and returns a session to access.

```
return cluster.connect();
```

The following snippets are from the *src/main/java/com/azure/cosmosdb/cassandra/repository/UserRepository.java* file.

- Create a new keyspace.

```
public void createKeyspace() {
    final String query = "CREATE KEYSPACE IF NOT EXISTS uprofile WITH replication = {'class':
'SimpleStrategy', 'replication_factor': '3' } ";
    session.execute(query);
    LOGGER.info("Created keyspace 'uprofile'");
}
```

- Create a new table.

```
public void createTable() {
      final String query = "CREATE TABLE IF NOT EXISTS uprofile.user (user_id int PRIMARY KEY,
user_name text, user_bcity text)";
      session.execute(query);
      LOGGER.info("Created table 'user'");
}
```

- Insert user entities using a prepared statement object.

```
public PreparedStatement prepareInsertStatement() {
    final String insertStatement = "INSERT INTO  uprofile.user (user_id, user_name , user_bcity) VALUES
(?,?,?)";
    return session.prepare(insertStatement);
}

  public void insertUser(PreparedStatement statement, int id, String name, String city) {
    BoundStatement boundStatement = new BoundStatement(statement);
    session.execute(boundStatement.bind(id, name, city));
}
```

- Query to get all user information.

```
public void selectAllUsers() {
    final String query = "SELECT * FROM uprofile.user";
    List<Row> rows = session.execute(query).all();

    for (Row row : rows) {
        LOGGER.info("Obtained row: {} | {} | {} ", row.getInt("user_id"), row.getString("user_name"),
row.getString("user_bcity"));
    }
}
```

- Query to get a single user's information.

```
public void selectUser(int id) {
    final String query = "SELECT * FROM uprofile.user where user_id = 3";
    Row row = session.execute(query).one();

    LOGGER.info("Obtained row: {} | {} | {} ", row.getInt("user_id"), row.getString("user_name"),
row.getString("user_bcity"));
}
```

Update your connection string

Now go back to the Azure portal to get your connection string information and copy it into the app. The connection string details enable your app to communicate with your hosted database.

1. In your Azure Cosmos DB account in the Azure portal, select **Connection String**.

2. Use the button on the right side of the screen to copy the CONTACT POINT value.

3. Open the *config.properties* file from the *C:\git-samples\azure-cosmosdb-cassandra-java-getting-started\java-examples\src\main\resources* folder.

4. Paste the CONTACT POINT value from the portal over `<Cassandra endpoint host>` on line 2.

   Line 2 of *config.properties* should now look similar to

   ```
   cassandra_host=cosmos-db-quickstart.cassandra.cosmosdb.azure.com
   ```

5. Go back to the portal and copy the USERNAME value. Past the USERNAME value from the portal over `<cassandra endpoint username>` on line 4.

   Line 4 of *config.properties* should now look similar to

   ```
   cassandra_username=cosmos-db-quickstart
   ```

6. Go back to the portal and copy the PASSWORD value. Paste the PASSWORD value from the portal over `<cassandra endpoint password>` on line 5.

   Line 5 of *config.properties* should now look similar to

   ```
   cassandra_password=2Ggkr662ifxz2Mg...==
   ```

7. On line 6, if you want to use a specific TLS/SSL certificate, then replace `<SSL key store file location>` with the location of the TLS/SSL certificate. If a value is not provided, the JDK certificate installed at <JAVA_HOME>/jre/lib/security/cacerts is used.

8. If you changed line 6 to use a specific TLS/SSL certificate, update line 7 to use the password for that certificate.

9. Save the *config.properties* file.

Run the Java app

1. In the git terminal window, `cd` to the `azure-cosmosdb-cassandra-java-getting-started` folder.

```
cd "C:\git-samples\azure-cosmosdb-cassandra-java-getting-started"
```

2. In the git terminal window, use the following command to generate the `cosmosdb-cassandra-examples.jar` file.

```
mvn clean install
```

3. In the git terminal window, run the following command to start the Java application.

```
java -cp target/cosmosdb-cassandra-examples.jar com.azure.cosmosdb.cassandra.examples.UserProfile
```

   The terminal window displays notifications that the keyspace and table are created. It then selects and returns all users in the table and displays the output, and then selects a row by ID and displays the value.

   Press Ctrl+C to stop execution of the program and close the console window.

4. In the Azure portal, open **Data Explorer** to query, modify, and work with this new data.

Review SLAs in the Azure portal

The Azure portal monitors your Cosmos DB account throughput, storage, availability, latency, and consistency. Charts for metrics associated with an Azure Cosmos DB Service Level Agreement (SLA) show the SLA value compared to actual performance. This suite of metrics makes monitoring your SLAs transparent.

To review metrics and SLAs:

1. Select **Metrics** in your Cosmos DB account's navigation menu.

2. Select a tab such as **Latency**, and select a timeframe on the right. Compare the **Actual** and **SLA** lines on the charts.

3. Review the metrics on the other tabs.

Clean up resources

When you're done with your app and Azure Cosmos DB account, you can delete the Azure resources you created so you don't incur more charges. To delete the resources:

1. In the Azure portal Search bar, search for and select **Resource groups**.

2. From the list, select the resource group you created for this quickstart.

3. On the resource group **Overview** page, select **Delete resource group**.

4. In the next window, enter the name of the resource group to delete, and then select **Delete**.

Next steps

In this quickstart, you learned how to create an Azure Cosmos DB account with Cassandra API, and run a Cassandra Java app that creates a Cassandra database and container. You can now import additional data into your Azure Cosmos DB account.

Import Cassandra data into Azure Cosmos DB

Quickstart: Build a Java app to manage Azure Cosmos DB Cassandra API data (v4 Driver)

11/2/2020 • 7 minutes to read • Edit Online

APPLIES TO: Cassandra API

In this quickstart, you create an Azure Cosmos DB Cassandra API account, and use a Cassandra Java app cloned from GitHub to create a Cassandra database and container using the v4.x Apache Cassandra drivers for Java. Azure Cosmos DB is a multi-model database service that lets you quickly create and query document, table, key-value, and graph databases with global distribution and horizontal scale capabilities.

Prerequisites

- An Azure account with an active subscription. Create one for free. Or try Azure Cosmos DB for free without an Azure subscription.
- Java Development Kit (JDK) 8. Point your `JAVA_HOME` environment variable to the folder where the JDK is installed.
- A Maven binary archive. On Ubuntu, run `apt-get install maven` to install Maven.
- Git. On Ubuntu, run `sudo apt-get install git` to install Git.

Create a database account

Before you can create a document database, you need to create a Cassandra account with Azure Cosmos DB.

1. In a new browser window, sign in to the Azure portal.

2. In the left menu, select **Create a resource**.

3. On the **New** page, select **Databases** > **Azure Cosmos DB**.

4. On the **Create Azure Cosmos DB Account** page, enter the settings for the new Azure Cosmos DB account.

SETTING	VALUE	DESCRIPTION
Subscription	Your subscription	Select the Azure subscription that you want to use for this Azure Cosmos DB account.
Resource Group	Create new Then enter the same name as Account Name	Select **Create new**. Then enter a new resource group name for your account. For simplicity, use the same name as your Azure Cosmos account name.
Account Name	Enter a unique name	Enter a unique name to identify your Azure Cosmos DB account. Your account URI will be *cassandra.cosmos.azure.com* appended to your unique account name. The account name can use only lowercase letters, numbers, and hyphens (-), and must be between 3 and 31 characters long.

SETTING	VALUE	DESCRIPTION
API	Cassandra	The API determines the type of account to create. Azure Cosmos DB provides five APIs: Core (SQL) for document databases, Gremlin for graph databases, MongoDB for document databases, Azure Table, and Cassandra. You must create a separate account for each API. Select **Cassandra**, because in this quickstart you are creating a table that works with the Cassandra API. Learn more about the Cassandra API.
Location	Select the region closest to your users	Select a geographic location to host your Azure Cosmos DB account. Use the location that's closest to your users to give them the fastest access to the data.
Capacity mode	Provisioned throughput or Serverless	Select **Provisioned throughput** to create an account in provisioned throughput mode. Select **Serverless** to create an account in serverless mode.

Select **Review+Create**. You can skip the **Networking**, **Backup**, **Encryption** and **Tags** section.

5. The account creation takes a few minutes. Wait for the portal to display the page saying **Congratulations! Your Azure Cosmos DB account was created**.

Clone the sample application

Now let's switch to working with code. Let's clone a Cassandra app from GitHub, set the connection string, and run it. You'll see how easy it is to work with data programmatically.

1. Open a command prompt. Create a new folder named `git-samples`. Then, close the command prompt.

```
md "C:\git-samples"
```

2. Open a git terminal window, such as git bash, and use the `cd` command to change to the new folder to install the sample app.

```
cd "C:\git-samples"
```

3. Run the following command to clone the sample repository. This command creates a copy of the sample app on your computer.

```
git clone https://github.com/Azure-Samples/azure-cosmos-db-cassandra-java-getting-started-v4.git
```

Review the code

This step is optional. If you're interested to learn how the code creates the database resources, you can review the following snippets. Otherwise, you can skip ahead to Update your connection string. These snippets are all taken from the *src/main/java/com/azure/cosmosdb/cassandra/util/CassandraUtils.java* file.

- The `CqlSession` connects to the Azure Cosmos DB Cassandra API and returns a session to access (`Cluster` object from v3 driver is now obsolete). Cassandra Host, Port, User name and password is set using the connection string page in the Azure portal.

```
    this.session = CqlSession.builder().withSslContext(sc)
            .addContactPoint(new InetSocketAddress(cassandraHost,
cassandraPort)).withLocalDatacenter(region)
            .withAuthCredentials(cassandraUsername, cassandraPassword).build();
```

The following snippets are from the *src/main/java/com/azure/cosmosdb/cassandra/repository/UserRepository.java* file.

- Drop the keyspace if it already exists from a previous run.

```
public void dropKeyspace() {
    String query = "DROP KEYSPACE IF EXISTS "+keyspace+"";
    session.execute(query);
    LOGGER.info("dropped keyspace '"+keyspace+"'");
}
```

- A new keyspace is created.

```
public void createKeyspace() {
    String query = "CREATE KEYSPACE "+keyspace+" WITH REPLICATION = { 'class' :
'NetworkTopologyStrategy', 'datacenter1' : 1 }";
    session.execute(query);
    LOGGER.info("Created keyspace '"+keyspace+"'");
}
```

- A new table is created.

```
  public void createTable() {
      String query = "CREATE TABLE "+keyspace+"."+table+" (user_id int PRIMARY KEY, user_name text,
user_bcity text)";
      session.execute(query);
      LOGGER.info("Created table '"+table+"'");
  }
```

- User entities are inserted using a prepared statement object.

```
public String prepareInsertStatement() {
    final String insertStatement = "INSERT INTO  "+keyspace+"."+table+" (user_id, user_name ,
user_bcity) VALUES (?,?,?)";
    return insertStatement;
}

public void insertUser(String preparedStatement, int id, String name, String city) {
    PreparedStatement prepared = session.prepare(preparedStatement);
    BoundStatement bound = prepared.bind(id, city, name).setIdempotent(true);
    session.execute(bound);
}
```

- Query to get get all User information.

```
public void selectAllUsers() {
    final String query = "SELECT * FROM "+keyspace+"."+table+"";
    List<Row> rows = session.execute(query).all();

    for (Row row : rows) {
        LOGGER.info("Obtained row: {} | {} | {} ", row.getInt("user_id"), row.getString("user_name"),
row.getString("user_bcity"));
    }
}
```

- Query to get a single User information.

```
public void selectUser(int id) {
    final String query = "SELECT * FROM "+keyspace+"."+table+" where user_id = 3";
    Row row = session.execute(query).one();

    LOGGER.info("Obtained row: {} | {} | {} ", row.getInt("user_id"), row.getString("user_name"),
row.getString("user_bcity"));
}
```

Update your connection string

Now go back to the Azure portal to get your connection string information and copy it into the app. The connection string details enable your app to communicate with your hosted database.

1. In your Azure Cosmos DB account in the Azure portal, select **Connection String**.

2. Use the button on the right side of the screen to copy the CONTACT POINT value.

3. Open the *config.properties* file from the *C:\git-samples\azure-cosmosdb-cassandra-java-getting-started\java-examples\src\main\resources* folder.

4. Paste the CONTACT POINT value from the portal over `<Cassandra endpoint host>` on line 2.

   Line 2 of *config.properties* should now look similar to

   ```
   cassandra_host=cosmos-db-quickstart.cassandra.cosmosdb.azure.com
   ```

5. Go back to the portal and copy the USERNAME value. Past the USERNAME value from the portal over `<cassandra endpoint username>` on line 4.

   Line 4 of *config.properties* should now look similar to

   ```
   cassandra_username=cosmos-db-quickstart
   ```

6. Go back to the portal and copy the PASSWORD value. Paste the PASSWORD value from the portal over `<cassandra endpoint password>` on line 5.

   Line 5 of *config.properties* should now look similar to

   ```
   cassandra_password=2Ggkr662ifxz2Mg...==
   ```

7. On line 6, if you want to use a specific TLS/SSL certificate, then replace `<SSL key store file location>` with the location of the TLS/SSL certificate. If a value is not provided, the JDK certificate installed at <JAVA_HOME>/jre/lib/security/cacerts is used.

8. If you changed line 6 to use a specific TLS/SSL certificate, update line 7 to use the password for that certificate.

9. Note that you will need to add the default region (e.g. `West US`) for the contact point, e.g.

   ```
   region=West US
   ```

   This is because the v.4x driver only allows one local DC to be paired with the contact point. If you want to add a region other than the default (which is the region that was given when the Cosmos DB account was

first created), you will need to use regional suffix when adding contact point, e.g. `host-westus.cassandra.cosmos.azure.com`.

10. Save the *config.properties* file.

Run the Java app

1. In the git terminal window, `cd` to the `azure-cosmosdb-cassandra-java-getting-started-v4` folder.

```
cd "C:\git-samples\azure-cosmosdb-cassandra-java-getting-started-v4"
```

2. In the git terminal window, use the following command to generate the `cosmosdb-cassandra-examples.jar` file.

```
mvn clean install
```

3. In the git terminal window, run the following command to start the Java application.

```
java -cp target/cosmosdb-cassandra-examples.jar com.azure.cosmosdb.cassandra.examples.UserProfile
```

   The terminal window displays notifications that the keyspace and table are created. It then selects and returns all users in the table and displays the output, and then selects a row by ID and displays the value.

   Press Ctrl+C to stop execution of the program and close the console window.

4. In the Azure portal, open **Data Explorer** to query, modify, and work with this new data.

Review SLAs in the Azure portal

The Azure portal monitors your Cosmos DB account throughput, storage, availability, latency, and consistency. Charts for metrics associated with an Azure Cosmos DB Service Level Agreement (SLA) show the SLA value compared to actual performance. This suite of metrics makes monitoring your SLAs transparent.

To review metrics and SLAs:

1. Select **Metrics** in your Cosmos DB account's navigation menu.

2. Select a tab such as **Latency**, and select a timeframe on the right. Compare the **Actual** and **SLA** lines on

the charts.

3. Review the metrics on the other tabs.

Clean up resources

When you're done with your app and Azure Cosmos DB account, you can delete the Azure resources you created so you don't incur more charges. To delete the resources:

1. In the Azure portal Search bar, search for and select **Resource groups**.

2. From the list, select the resource group you created for this quickstart.

3. On the resource group **Overview** page, select **Delete resource group**.

4. In the next window, enter the name of the resource group to delete, and then select **Delete**.

Next steps

In this quickstart, you learned how to create an Azure Cosmos DB account with Cassandra API, and run a Cassandra Java app that creates a Cassandra database and container. You can now import additional data into your Azure Cosmos DB account.

Import Cassandra data into Azure Cosmos DB

Quickstart: Build a Cassandra app with Node.js SDK and Azure Cosmos DB

11/2/2020 • 7 minutes to read • Edit Online

APPLIES TO: Cassandra API

In this quickstart, you create an Azure Cosmos DB Cassandra API account, and use a Cassandra Node.js app cloned from GitHub to create a Cassandra database and container. Azure Cosmos DB is a multi-model database service that lets you quickly create and query document, table, key-value, and graph databases with global distribution and horizontal scale capabilities.

Prerequisites

If you don't have an Azure subscription, create a free account before you begin. Alternatively, you can Try Azure Cosmos DB for free without an Azure subscription, free of charge and commitments.

In addition, you need:

- Node.js version v0.10.29 or higher
- Git

Create a database account

Before you can create a document database, you need to create a Cassandra account with Azure Cosmos DB.

1. In a new browser window, sign in to the Azure portal.

2. In the left menu, select **Create a resource**.

3. On the **New** page, select **Databases** > **Azure Cosmos DB**.

4. On the **Create Azure Cosmos DB Account** page, enter the settings for the new Azure Cosmos DB account.

SETTING	VALUE	DESCRIPTION
Subscription	Your subscription	Select the Azure subscription that you want to use for this Azure Cosmos DB account.
Resource Group	Create new Then enter the same name as Account Name	Select **Create new**. Then enter a new resource group name for your account. For simplicity, use the same name as your Azure Cosmos account name.
Account Name	Enter a unique name	Enter a unique name to identify your Azure Cosmos DB account. Your account URI will be *cassandra.cosmos.azure.com* appended to your unique account name. The account name can use only lowercase letters, numbers, and hyphens (-), and must be between 3 and 31 characters long.

SETTING	VALUE	DESCRIPTION
API	Cassandra	The API determines the type of account to create. Azure Cosmos DB provides five APIs: Core (SQL) for document databases, Gremlin for graph databases, MongoDB for document databases, Azure Table, and Cassandra. You must create a separate account for each API. Select **Cassandra**, because in this quickstart you are creating a table that works with the Cassandra API. Learn more about the Cassandra API.
Location	Select the region closest to your users	Select a geographic location to host your Azure Cosmos DB account. Use the location that's closest to your users to give them the fastest access to the data.
Capacity mode	Provisioned throughput or Serverless	Select **Provisioned throughput** to create an account in provisioned throughput mode. Select **Serverless** to create an account in serverless mode.

Select **Review+Create**. You can skip the **Networking**, **Backup**, **Encryption** and **Tags** section.

5. The account creation takes a few minutes. Wait for the portal to display the page saying **Congratulations! Your Azure Cosmos DB account was created**.

Clone the sample application

Now let's clone a Cassandra API app from GitHub, set the connection string, and run it. You see how easy it is to work with data programmatically.

1. Open a command prompt. Create a new folder named `git-samples`. Then, close the command prompt.

```
md "C:\git-samples"
```

2. Open a git terminal window, such as git bash. Use the `cd` command to change to the new folder to install the sample app.

```
cd "C:\git-samples"
```

3. Run the following command to clone the sample repository. This command creates a copy of the sample app on your computer.

```
git clone https://github.com/Azure-Samples/azure-cosmos-db-cassandra-nodejs-getting-started.git
```

Review the code

This step is optional. If you're interested to learn how the code creates the database resources, you can review the following snippets. The snippets are all taken from the `uprofile.js` file in the `C:\git-samples\azure-cosmos-db-cassandra-nodejs-getting-started` folder. Otherwise, you can skip ahead to Update your connection string.

- The username and password values were set using the connection string page in the Azure portal. The `path\to\cert` provides a path to an X509 certificate.

```
var ssl_option = {
      cert : fs.readFileSync("path\to\cert"),
      rejectUnauthorized : true,
      secureProtocol: 'TLSv1_2_method'
      };
const authProviderLocalCassandra = new cassandra.auth.PlainTextAuthProvider(config.username,
config.password);
```

- The `client` is initialized with contactPoint information. The contactPoint is retrieved from the Azure portal.

```
const client = new cassandra.Client({contactPoints: [config.contactPoint], authProvider:
authProviderLocalCassandra, sslOptions:ssl_option});
```

- The `client` connects to the Azure Cosmos DB Cassandra API.

```
client.connect(next);
```

- A new keyspace is created.

```
function createKeyspace(next) {
 var query = "CREATE KEYSPACE IF NOT EXISTS uprofile WITH replication = {\'class\':
\'NetworkTopologyStrategy\', \'datacenter1\' : \'1\' }";
 client.execute(query, next);
 console.log("created keyspace");
}
```

- A new table is created.

```
function createTable(next) {
  var query = "CREATE TABLE IF NOT EXISTS uprofile.user (user_id int PRIMARY KEY, user_name text,
user_bcity text)";
   client.execute(query, next);
   console.log("created table");
},
```

- Key/value entities are inserted.

```
    function insert(next) {
        console.log("\insert");
        const arr = ['INSERT INTO  uprofile.user (user_id, user_name , user_bcity) VALUES (1,
\'AdrianaS\', \'Seattle\')',
                    'INSERT INTO  uprofile.user (user_id, user_name , user_bcity) VALUES (2,
\'JiriK\', \'Toronto\')',
                    'INSERT INTO  uprofile.user (user_id, user_name , user_bcity) VALUES (3,
\'IvanH\', \'Mumbai\')',
                    'INSERT INTO  uprofile.user (user_id, user_name , user_bcity) VALUES (4,
\'IvanH\', \'Seattle\')',
                    'INSERT INTO  uprofile.user (user_id, user_name , user_bcity) VALUES (5,
\'IvanaV\', \'Belgaum\')',
                    'INSERT INTO  uprofile.user (user_id, user_name , user_bcity) VALUES (6,
\'LiliyaB\', \'Seattle\')',
                    'INSERT INTO  uprofile.user (user_id, user_name , user_bcity) VALUES (7,
\'JindrichH\', \'Buenos Aires\')',
                    'INSERT INTO  uprofile.user (user_id, user_name , user_bcity) VALUES (8,
\'AdrianaS\', \'Seattle\')',
                    'INSERT INTO  uprofile.user (user_id, user_name , user_bcity) VALUES (9,
\'JozefM\', \'Seattle\')'];
        arr.forEach(element => {
        client.execute(element);
        });
        next();
    },
```

- Query to get all key values.

```
    function selectAll(next) {
        console.log("\Select ALL");
        var query = 'SELECT * FROM uprofile.user';
        client.execute(query, function (err, result) {
        if (err) return next(err);
        result.rows.forEach(function(row) {
            console.log('Obtained row: %d | %s | %s ',row.user_id, row.user_name, row.user_bcity);
        }, this);
        next();
        });
    },
```

- Query to get a key-value.

```
    function selectById(next) {
        console.log("\Getting by id");
        var query = 'SELECT * FROM uprofile.user where user_id=1';
        client.execute(query, function (err, result) {
        if (err) return next(err);
        result.rows.forEach(function(row) {
            console.log('Obtained row: %d | %s | %s ',row.user_id, row.user_name, row.user_bcity);
        }, this);
        next();
        });
    }
```

Update your connection string

Now go back to the Azure portal to get your connection string information and copy it into the app. The connection string enables your app to communicate with your hosted database.

1. In your Azure Cosmos DB account in the Azure portal, select **Connection String**.

2. Use the button on the right side of the screen to copy the top value, the CONTACT POINT.

3. Open the `config.js` file.

4. Paste the CONTACT POINT value from the portal over `<FillMEIN>` on line 4.

   Line 4 should now look similar to

   ```
   config.contactPoint = "cosmos-db-quickstarts.cassandra.cosmosdb.azure.com:10350"
   ```

5. Copy the USERNAME value from the portal and paste it over `<FillMEIN>` on line 2.

   Line 2 should now look similar to

   ```
   config.username = 'cosmos-db-quickstart';
   ```

6. Copy the PASSWORD value from the portal and paste it over `<FillMEIN>` on line 3.

   Line 3 should now look similar to

   ```
   config.password = '2Ggkr662ifxz2Mg==';
   ```

7. Save the `config.js` file.

Use the X509 certificate

1. Download the Baltimore CyberTrust Root certificate locally from https://cacert.omniroot.com/bc2025.crt. Rename the file using the file extension `.cer`.

   The certificate has serial number `02:00:00:b9` and SHA1 fingerprint `d4□□20:d0:5e:66:fc:53:fe:1a:50:88:2c:78:db:28:52:ca:e4:74`.

2. Open `uprofile.js` and change the `path\to\cert` to point to your new certificate.

3. Save `uprofile.js`.

NOTE

If you experience a certificate related error in the later steps and are running on a Windows machine, ensure that you have followed the process for properly converting a .crt file into the Microsoft .cer format below.

Double-click on the .crt file to open it into the certificate display.

Press Next on the Certificate Wizard. Select Base-64 encoded X.509 (.CER), then Next.

Select Browse (to locate a destination) and type in a filename. Select Next then Finished.

You should now have a properly formatted .cer file. Ensure that the path in `uprofile.js` points to this file.

Run the Node.js app

1. In the git terminal window, ensure you are in the sample directory you cloned earlier:

```
cd azure-cosmos-db-cassandra-nodejs-getting-started
```

2. Run `npm install` to install the required npm modules.

3. Run `node uprofile.js` to start your node application.

4. Verify the results as expected from the command line.

```
D:\Nodejs-codeSample\Samples\Samples>node qs.js
created keyspace
created table
Select ALL
Obtained row: 1 | LyubovK | Dubai
Obtained row: 2 | JiriK | Toronto
Obtained row: 3 | IvanH | Mumbai
Obtained row: 4 | IvanH | Seattle
Obtained row: 5 | IvanaV | Belgaum
Obtained row: 6 | LiliyaB | Seattle
Obtained row: 7 | JindrichH | Buenos Aires
Obtained row: 8 | AdrianaS | Seattle
Obtained row: 9 | JozefM | Seattle
Obtained row: 10 | EmmaH | Seattle
Obtained row: 11 | GrzegorzM | Seattle
Obtained row: 12 | FryderykK | Seattle
Obtained row: 13 | DesislavaL | Seattle
Getting by id
Obtained row: 1 | LyubovK | Dubai
Please delete your table after verifying the presence of data in portal or from CQL
```

Press CTRL+C to stop execution of the program and close the console window.

5. In the Azure portal, open **Data Explorer** to query, modify, and work with this new data.

Review SLAs in the Azure portal

The Azure portal monitors your Cosmos DB account throughput, storage, availability, latency, and consistency. Charts for metrics associated with an Azure Cosmos DB Service Level Agreement (SLA) show the SLA value compared to actual performance. This suite of metrics makes monitoring your SLAs transparent.

To review metrics and SLAs:

1. Select **Metrics** in your Cosmos DB account's navigation menu.

2. Select a tab such as **Latency**, and select a timeframe on the right. Compare the **Actual** and **SLA** lines on the charts.

3. Review the metrics on the other tabs.

Clean up resources

When you're done with your app and Azure Cosmos DB account, you can delete the Azure resources you created so you don't incur more charges. To delete the resources:

1. In the Azure portal Search bar, search for and select **Resource groups**.

2. From the list, select the resource group you created for this quickstart.

3. On the resource group **Overview** page, select **Delete resource group**.

4. In the next window, enter the name of the resource group to delete, and then select **Delete**.

Next steps

In this quickstart, you learned how to create an Azure Cosmos DB account with Cassandra API, and run a Cassandra Node.js app that creates a Cassandra database and container. You can now import additional data into your Azure Cosmos DB account.

Import Cassandra data into Azure Cosmos DB

Quickstart: Build a Cassandra app with Python SDK and Azure Cosmos DB

11/2/2020 • 6 minutes to read • Edit Online

APPLIES TO: Cassandra API

In this quickstart, you create an Azure Cosmos DB Cassandra API account, and use a Cassandra Python app cloned from GitHub to create a Cassandra database and container. Azure Cosmos DB is a multi-model database service that lets you quickly create and query document, table, key-value, and graph databases with global distribution and horizontal scale capabilities.

Prerequisites

- An Azure account with an active subscription. Create one for free. Or try Azure Cosmos DB for free without an Azure subscription.
- Python 2.7.14+ or 3.4+.
- Git.
- Python Driver for Apache Cassandra.

Create a database account

Before you can create a document database, you need to create a Cassandra account with Azure Cosmos DB.

1. In a new browser window, sign in to the Azure portal.
2. In the left menu, select **Create a resource**.

3. On the **New** page, select **Databases** > **Azure Cosmos DB**.

4. On the **Create Azure Cosmos DB Account** page, enter the settings for the new Azure Cosmos DB account.

SETTING	VALUE	DESCRIPTION
Subscription	Your subscription	Select the Azure subscription that you want to use for this Azure Cosmos DB account.
Resource Group	Create new Then enter the same name as Account Name	Select **Create new**. Then enter a new resource group name for your account. For simplicity, use the same name as your Azure Cosmos account name.
Account Name	Enter a unique name	Enter a unique name to identify your Azure Cosmos DB account. Your account URI will be *cassandra.cosmos.azure.com* appended to your unique account name. The account name can use only lowercase letters, numbers, and hyphens (-), and must be between 3 and 31 characters long.

SETTING	VALUE	DESCRIPTION
API	Cassandra	The API determines the type of account to create. Azure Cosmos DB provides five APIs: Core (SQL) for document databases, Gremlin for graph databases, MongoDB for document databases, Azure Table, and Cassandra. You must create a separate account for each API. Select **Cassandra**, because in this quickstart you are creating a table that works with the Cassandra API. Learn more about the Cassandra API.
Location	Select the region closest to your users	Select a geographic location to host your Azure Cosmos DB account. Use the location that's closest to your users to give them the fastest access to the data.
Capacity mode	Provisioned throughput or Serverless	Select **Provisioned throughput** to create an account in provisioned throughput mode. Select **Serverless** to create an account in serverless mode.

Select **Review+Create**. You can skip the **Networking**, **Backup**, **Encryption** and **Tags** section.

5. The account creation takes a few minutes. Wait for the portal to display the page saying **Congratulations! Your Azure Cosmos DB account was created**.

Clone the sample application

Now let's clone a Cassandra API app from GitHub, set the connection string, and run it. You see how easy it is to work with data programmatically.

1. Open a command prompt. Create a new folder named `git-samples`. Then, close the command prompt.

```
md "C:\git-samples"
```

2. Open a git terminal window, such as git bash, and use the `cd` command to change to the new folder to install the sample app.

```
cd "C:\git-samples"
```

3. Run the following command to clone the sample repository. This command creates a copy of the sample app on your computer.

```
git clone https://github.com/Azure-Samples/azure-cosmos-db-cassandra-python-getting-started.git
```

Review the code

This step is optional. If you're interested to learn how the code creates the database resources, you can review the following snippets. The snippets are all taken from the *pyquickstart.py* file. Otherwise, you can skip ahead to Update your connection string.

- The `cluster` is initialized with `contactPoint` and `port` information that is retrieved from the Azure portal. The `cluster` then connects to the Azure Cosmos DB Cassandra API by using the `connect()` method. An authorized connection is established by using the username, password, and the default certificate or an explicit certificate if you provide one within the config file.

```
ssl_context = SSLContext(PROTOCOL_TLSv1_2)
ssl_context.verify_mode = CERT_NONE
auth_provider = PlainTextAuthProvider(username=cfg.config['username'],
password=cfg.config['password'])
cluster = Cluster([cfg.config['contactPoint']], port = cfg.config['port'],
auth_provider=auth_provider,ssl_context=ssl_context)
session = cluster.connect()
```

- A new keyspace is created.

```
print ("\nCreating Keyspace")
session.execute('CREATE KEYSPACE IF NOT EXISTS uprofile WITH replication = {\'class\':
\'NetworkTopologyStrategy\', \'datacenter\' : \'1\' }');
```

- A new table is created.

```
print ("\nCreating Table")
session.execute('CREATE TABLE IF NOT EXISTS uprofile.user (user_id int PRIMARY KEY, user_name text,
user_bcity text)');
```

- Key/value entities are inserted.

```
session.execute("INSERT INTO  uprofile.user  (user_id, user_name , user_bcity) VALUES (%s,%s,%s)",
[1,'Lybkov','Seattle'])
session.execute("INSERT INTO  uprofile.user  (user_id, user_name , user_bcity) VALUES (%s,%s,%s)",
[2,'Doniv','Dubai'])
session.execute("INSERT INTO  uprofile.user  (user_id, user_name , user_bcity) VALUES (%s,%s,%s)",
[3,'Keviv','Chennai'])
session.execute("INSERT INTO  uprofile.user  (user_id, user_name , user_bcity) VALUES (%s,%s,%s)",
[4,'Ehtevs','Pune'])
session.execute("INSERT INTO  uprofile.user  (user_id, user_name , user_bcity) VALUES (%s,%s,%s)",
[5,'Dnivog','Belgaum'])
session.execute("INSERT INTO  uprofile.user  (user_id, user_name , user_bcity) VALUES (%s,%s,%s)",
[6,'Ateegk','Narewadi'])
session.execute("INSERT INTO  uprofile.user  (user_id, user_name , user_bcity) VALUES (%s,%s,%s)",
[7,'KannabbuS','Yamkanmardi'])
```

- Query to get all key values.

```
print ("\nSelecting All")
rows = session.execute('SELECT * FROM uprofile.user')
PrintTable(rows)
```

- Query to get a key-value.

```
print ("\nSelecting Id=1")
rows = session.execute('SELECT * FROM uprofile.user where user_id=1')
PrintTable(rows)
```

Update your connection string

Now go back to the Azure portal to get your connection string information and copy it into the app. The connection string enables your app to communicate with your hosted database.

1. In your Azure Cosmos DB account in the Azure portal, select **Connection String**.

2. Use the button on the right side of the screen to copy the top value, the CONTACT POINT.

3. Open the *config.py* file.

4. Paste the CONTACT POINT value from the portal over `<FILLME>` on line 10.

   Line 10 should now look similar to

   `'contactPoint': 'cosmos-db-quickstarts.cassandra.cosmosdb.azure.com:10350'`

5. Copy the USERNAME value from the portal and paste it over `<FILLME>` on line 6.

   Line 6 should now look similar to

   `'username': 'cosmos-db-quickstart',`

6. Copy the PASSWORD value from the portal and paste it over `<FILLME>` on line 8.

   Line 8 should now look similar to

`'password' = '2Ggkr662ifxz2Mg==`';`

7. Save the *config.py* file.

Use the X509 certificate

1. Download the Baltimore CyberTrust Root certificate locally from https://cacert.omniroot.com/bc2025.crt. Rename the file using the file extension *.cer*.

   The certificate has serial number `02:00:00:b9` and SHA1 fingerprint `d4:de:20:d0:5e:66:fc:53:fe:1a:50:88:2c:78:db:28:52:ca:e4:74`.

2. Open *pyquickstart.py* and change the `path\to\cert` to point to your new certificate.

3. Save *pyquickstart.py*.

Run the Python app

1. Use the cd command in the git terminal to change into the `azure-cosmos-db-cassandra-python-getting-started` folder.

2. Run the following commands to install the required modules:

```
python -m pip install cassandra-driver
python -m pip install prettytable
python -m pip install requests
python -m pip install pyopenssl
```

3. Run the following command to start your Python application:

```
python pyquickstart.py
```

4. Verify the results as expected from the command line.

   Press CTRL+C to stop execution of the program and close the console window.

```
D:\PythonSamples\python-cassandra\PythonSamples>python qs.py

Creating Keyspace

Creating Table

Selecting All
+--------+------------+--------------+
| UserID |    Name    |     City     |
+--------+------------+--------------+
|   1    |  LyubovK   |    Dubai     |
|   2    |   JiriK    |   Toronto    |
|   3    |   IvanH    |    Mumbai    |
|   4    |   YuliaT   |   Seattle    |
|   5    |   IvanaV   |   Belgaum    |
|   6    |  LiliyaB   |   Seattle    |
|   7    | JindrichH  | Buenos Aires |
|   8    |  AdrianaS  |   Seattle    |
|   9    |   JozefM   |   Seattle    |
|   10   |   EmmaH    |   Seattle    |
|   11   | GrzegorzM  |   Seattle    |
|   12   | FryderykK  |   Seattle    |
|   13   | DesislavaL |   Seattle    |
+--------+------------+--------------+

Selecting Id=1
+--------+---------+-------+
| UserID |   Name  |  City |
+--------+---------+-------+
|   1    | LyubovK | Dubai |
+--------+---------+-------+
```

5. In the Azure portal, open **Data Explorer** to query, modify, and work with this new data.

Review SLAs in the Azure portal

The Azure portal monitors your Cosmos DB account throughput, storage, availability, latency, and consistency. Charts for metrics associated with an Azure Cosmos DB Service Level Agreement (SLA) show the SLA value compared to actual performance. This suite of metrics makes monitoring your SLAs transparent.

To review metrics and SLAs:

1. Select **Metrics** in your Cosmos DB account's navigation menu.
2. Select a tab such as **Latency**, and select a timeframe on the right. Compare the **Actual** and **SLA** lines on the charts.

3. Review the metrics on the other tabs.

Clean up resources

When you're done with your app and Azure Cosmos DB account, you can delete the Azure resources you created so you don't incur more charges. To delete the resources:

1. In the Azure portal Search bar, search for and select **Resource groups**.
2. From the list, select the resource group you created for this quickstart.

3. On the resource group **Overview** page, select **Delete resource group**.

4. In the next window, enter the name of the resource group to delete, and then select **Delete**.

Next steps

In this quickstart, you learned how to create an Azure Cosmos DB account with Cassandra API, and run a Cassandra Python app that creates a Cassandra database and container. You can now import additional data into your Azure Cosmos DB account.

Import Cassandra data into Azure Cosmos DB

Quickstart: Build a Go app with the `gocql` client to manage Azure Cosmos DB Cassandra API data

11/2/2020 • 8 minutes to read • Edit Online

APPLIES TO: Cassandra API

Azure Cosmos DB is a multi-model database service that lets you quickly create and query document, table, key-value, and graph databases with global distribution and horizontal scale capabilities. In this quickstart, you will start by creating an Azure Cosmos DB Cassandra API account. You will then run a Go application to create a Cassandra keyspace, table, and execute a few operations. This Go app uses gocql, which is a Cassandra client for the Go language.

Prerequisites

- An Azure account with an active subscription. Create one for free. Or try Azure Cosmos DB for free without an Azure subscription.
- Go installed on your computer, and a working knowledge of Go.
- Git.

Create a database account

Before you can create a database, you need to create a Cassandra account with Azure Cosmos DB.

1. In a new browser window, sign in to the Azure portal.

2. In the left menu, select **Create a resource**.

3. On the **New** page, select **Databases** > **Azure Cosmos DB**.

4. On the **Create Azure Cosmos DB Account** page, enter the settings for the new Azure Cosmos DB account.

SETTING	VALUE	DESCRIPTION
Subscription	Your subscription	Select the Azure subscription that you want to use for this Azure Cosmos DB account.
Resource Group	Create new Then enter the same name as Account Name	Select **Create new**. Then enter a new resource group name for your account. For simplicity, use the same name as your Azure Cosmos account name.
Account Name	Enter a unique name	Enter a unique name to identify your Azure Cosmos DB account. Your account URI will be *cassandra.cosmos.azure.com* appended to your unique account name. The account name can use only lowercase letters, numbers, and hyphens (-), and must be between 3 and 31 characters long.

SETTING	VALUE	DESCRIPTION
API	Cassandra	The API determines the type of account to create. Azure Cosmos DB provides five APIs: Core (SQL) for document databases, Gremlin for graph databases, MongoDB for document databases, Azure Table, and Cassandra. You must create a separate account for each API. Select **Cassandra**, because in this quickstart you are creating a table that works with the Cassandra API. Learn more about the Cassandra API.
Location	Select the region closest to your users	Select a geographic location to host your Azure Cosmos DB account. Use the location that's closest to your users to give them the fastest access to the data.
Capacity mode	Provisioned throughput or Serverless	Select **Provisioned throughput** to create an account in provisioned throughput mode. Select **Serverless** to create an account in serverless mode.

Select **Review+Create**. You can skip the **Networking**, **Backup**, **Encryption** and **Tags** section.

5. The account creation takes a few minutes. Wait for the portal to display the page saying **Congratulations! Your Azure Cosmos DB account was created**.

Clone the sample application

Start by cloning the application from GitHub.

1. Open a command prompt and create a new folder named `git-samples`.

```
md "C:\git-samples"
```

2. Open a git terminal window, such as git bash. Use the `cd` command to change into the new folder and install the sample app.

```
cd "C:\git-samples"
```

3. Run the following command to clone the sample repository. This command creates a copy of the sample app on your computer.

```
git clone https://github.com/Azure-Samples/azure-cosmos-db-cassandra-go-getting-started.git
```

Review the code

This step is optional. If you're interested to learn how the code creates the database resources, you can review the following code snippets. Otherwise, you can skip ahead to Run the application

The `GetSession` function (part of `utils\utils.go`) returns a `*gocql.Session` that is used to execute cluster operations such as insert, find etc.

```
func GetSession(cosmosCassandraContactPoint, cosmosCassandraPort, cosmosCassandraUser, cosmosCassandraPassword
string) *gocql.Session {
    clusterConfig := gocql.NewCluster(cosmosCassandraContactPoint)
    port, err := strconv.Atoi(cosmosCassandraPort)

    clusterConfig.Authenticator = gocql.PasswordAuthenticator{Username: cosmosCassandraUser, Password:
cosmosCassandraPassword}
    clusterConfig.Port = port
    clusterConfig.SslOpts = &gocql.SslOptions{Config: &tls.Config{MinVersion: tls.VersionTLS12}}
    clusterConfig.ProtoVersion = 4

    session, err := clusterConfig.CreateSession()
    ...
    return session
}
```

The Azure Cosmos DB Cassandra host is passed to the `gocql.NewCluster` function to get a `*gocql.ClusterConfig` struct that is then configured to use the username, password, port, and appropriate TLS version (HTTPS/SSL/TLS encryption Security requirement)

The `GetSession` function is then called from the `main` function (`main.go`).

```
func main() {
    session := utils.GetSession(cosmosCassandraContactPoint, cosmosCassandraPort, cosmosCassandraUser,
cosmosCassandraPassword)
    defer session.Close()
    ...
}
```

The connectivity information and credentials are accepted in the form of environment variables (resolved in the `init` method)

```
func init() {
    cosmosCassandraContactPoint = os.Getenv("COSMOSDB_CASSANDRA_CONTACT_POINT")
    cosmosCassandraPort = os.Getenv("COSMOSDB_CASSANDRA_PORT")
    cosmosCassandraUser = os.Getenv("COSMOSDB_CASSANDRA_USER")
    cosmosCassandraPassword = os.Getenv("COSMOSDB_CASSANDRA_PASSWORD")

    if cosmosCassandraContactPoint == "" || cosmosCassandraUser == "" || cosmosCassandraPassword == "" {
        log.Fatal("missing mandatory environment variables")
    }
}
```

It is then used to execute various operations (part of `operations\setup.go`) on Azure Cosmos DB starting with

`keyspace` and `table` creation.

As the name suggests, the `DropKeySpaceIfExists` function drops the `keyspace` only if it exists.

```
const dropKeyspace = "DROP KEYSPACE IF EXISTS %s"

func DropKeySpaceIfExists(keyspace string, session *gocql.Session) {
    err := utils.ExecuteQuery(fmt.Sprintf(dropKeyspace, keyspace), session)
    if err != nil {
        log.Fatal("Failed to drop keyspace", err)
    }
    log.Println("Keyspace dropped")
}
```

`CreateKeySpace` function is used to create the `keyspace` (`user_profile`)

```
const createKeyspace = "CREATE KEYSPACE %s WITH REPLICATION = { 'class' : 'NetworkTopologyStrategy',
'datacenter1' : 1 }"

func CreateKeySpace(keyspace string, session *gocql.Session) {
    err := utils.ExecuteQuery(fmt.Sprintf(createKeyspace, keyspace), session)
    if err != nil {
        log.Fatal("Failed to create keyspace", err)
    }
    log.Println("Keyspace created")
}
```

This is followed by table creation (`user`) which is taken care of `CreateUserTable` function

```
const createTable = "CREATE TABLE %s.%s (user_id int PRIMARY KEY, user_name text, user_bcity text)"

func CreateUserTable(keyspace, table string, session *gocql.Session) {
    err := session.Query(fmt.Sprintf(createTable, keyspace, table)).Exec()
    if err != nil {
        log.Fatal("failed to create table ", err)
    }
    log.Println("Table created")
}
```

Once the keyspace and table are created, we invoke CRUD operations (part of `operations\crud.go`).

`InsertUser` is used to create a `User`. It sets the user info (ID, name, and city) as the query arguments using `Bind`

```
const createQuery = "INSERT INTO %s.%s (user_id, user_name , user_bcity) VALUES (?,?,?)"

func InsertUser(keyspace, table string, session *gocql.Session, user model.User) {
    err := session.Query(fmt.Sprintf(createQuery, keyspace, table)).Bind(user.ID, user.Name, user.City).Exec()
    if err != nil {
        log.Fatal("Failed to create user", err)
    }
    log.Println("User created")
}
```

`FindUser` is used to search for a user (`model\user.go`) using a specific user ID while `Scan` binds the user attributes (returned by Cassandra) to individual variables (`userid`, `name`, `city`) -it is just one of the ways in which you can use the result obtained as the search query result

```
const selectQuery = "SELECT * FROM %s.%s where user_id = ?"

func FindUser(keyspace, table string, id int, session *gocql.Session) model.User {
    var userid int
    var name, city string
    err := session.Query(fmt.Sprintf(selectQuery, keyspace, table)).Bind(id).Scan(&userid, &name, &city)

    if err != nil {
        if err == gocql.ErrNotFound {
            log.Printf("User with id %v does not exist\n", id)
        } else {
            log.Printf("Failed to find user with id %v - %v\n", id, err)
        }
    }
    return model.User{ID: userid, Name: name, City: city}
}
```

`FindAllUsers` is used to fetch all the users. `SliceMap` is used as a shorthand to get all the user's info in the form of a slice of `map`s. Think of each `map` as key-value pairs where column name (for example, `user_id`) is the key along with its respective value.

```
const findAllUsersQuery = "SELECT * FROM %s.%s"

func FindAllUsers(keyspace, table string, session *gocql.Session) []model.User {
    var users []model.User
    results, _ := session.Query(fmt.Sprintf(findAllUsersQuery, keyspace, table)).Iter().SliceMap()

    for _, u := range results {
        users = append(users, mapToUser(u))
    }
    return users
}
```

Each `map` of user info is converted to a `User` using `mapToUser` function that simply extracts the value from its respective column and uses it to create an instance of the `User` struct

```
func mapToUser(m map[string]interface{}) model.User {
    id, _ := m["user_id"].(int)
    name, _ := m["user_name"].(string)
    city, _ := m["user_bcity"].(string)

    return model.User{ID: id, Name: name, City: city}
}
```

Run the application

As previously mentioned, the application accepts connectivity and credentials in the form the environment variables.

1. In your Azure Cosmos DB account in the Azure portal, select **Connection String**.

Copy the values for the following attributes (`CONTACT POINT`, `PORT`, `USERNAME` and `PRIMARY PASSWORD`) and set them to the respective environment variables

```
set COSMOSDB_CASSANDRA_CONTACT_POINT=<value for "CONTACT POINT">
set COSMOSDB_CASSANDRA_PORT=<value for "PORT">
set COSMOSDB_CASSANDRA_USER=<value for "USERNAME">
set COSMOSDB_CASSANDRA_PASSWORD=<value for "PRIMARY PASSWORD">
```

In the terminal window, change to the correct folder. For example:

```
cd "C:\git-samples\azure-cosmosdb-cassandra-go-getting-started"
```

2. In the terminal, run the following command to start the application.

```
go run main.go
```

3. The terminal window displays notifications for the various operations including keyspace and table setup, user creation etc.

4. In the Azure portal, open **Data Explorer** to query, modify, and work with this new data.

Review SLAs in the Azure portal

The Azure portal monitors your Cosmos DB account throughput, storage, availability, latency, and consistency. Charts for metrics associated with an Azure Cosmos DB Service Level Agreement (SLA) show the SLA value compared to actual performance. This suite of metrics makes monitoring your SLAs transparent.

To review metrics and SLAs:

1. Select **Metrics** in your Cosmos DB account's navigation menu.

2. Select a tab such as **Latency**, and select a timeframe on the right. Compare the **Actual** and **SLA** lines on the charts.

3. Review the metrics on the other tabs.

Clean up resources

When you're done with your app and Azure Cosmos DB account, you can delete the Azure resources you created so you don't incur more charges. To delete the resources:

1. In the Azure portal Search bar, search for and select **Resource groups**.

2. From the list, select the resource group you created for this quickstart.

3. On the resource group **Overview** page, select **Delete resource group**.

4. In the next window, enter the name of the resource group to delete, and then select **Delete**.

Next steps

In this quickstart, you learned how to create an Azure Cosmos DB account with Cassandra API, and run a Go app that creates a Cassandra database and container. You can now import additional data into your Azure Cosmos DB account.

Import Cassandra data into Azure Cosmos DB

Tutorial: Create a Cassandra API account in Azure Cosmos DB by using a Java application to store key/value data

11/2/2020 • 5 minutes to read • Edit Online

APPLIES TO: Cassandra API

As a developer, you might have applications that use key/value pairs. You can use a Cassandra API account in Azure Cosmos DB to store the key/value data. This tutorial describes how to use a Java application to create a Cassandra API account in Azure Cosmos DB, add a database (also called a keyspace), and add a table. The Java application uses the Java driver to create a user database that contains details such as user ID, user name, and user city.

This tutorial covers the following tasks:

- Create a Cassandra database account
- Get the account connection string
- Create a Maven project and dependencies
- Add a database and a table
- Run the app

Prerequisites

- If you don't have an Azure subscription, create a free account before you begin.
- Get the latest version of Java Development Kit (JDK).
- Download and install the Maven binary archive.
  - On Ubuntu, you can run `apt-get install maven` to install Maven.

Create a database account

1. Sign in to the Azure portal.
2. Select **Create a resource** > **Databases** > **Azure Cosmos DB**.
3. In the **New account** pane, enter the settings for the new Azure Cosmos account.

SETTING	SUGGESTED VALUE	DESCRIPTION
ID	Enter a unique name	Enter a unique name to identify this Azure Cosmos account. Because cassandra.cosmosdb.azure.com is appended to the ID that you provide to create your contact point, use a unique but identifiable ID.

SETTING	SUGGESTED VALUE	DESCRIPTION
API	Cassandra	The API determines the type of account to create. Select **Cassandra**, because in this article you will create a wide-column database that can be queried by using Cassandra Query Language (CQL) syntax.
Subscription	Your subscription	Select Azure subscription that you want to use for this Azure Cosmos account.
Resource Group	Enter a name	Select **Create New**, and then enter a new resource-group name for your account. For simplicity, you can use the same name as your ID.
Location	Select the region closest to your users	Select the geographic location in which to host your Azure Cosmos account. Use the location that's closest to your users, to give them the fastest access to the data.

4. Select **Create**.
   The account creation takes a few minutes. After the resource is created, you can see the **Deployment succeeded** notification on the right side of the portal.

Get the connection details of your account

Get the connection string information from the Azure portal, and copy it into the Java configuration file. The connection string enables your app to communicate with your hosted database.

1. From the Azure portal, go to your Azure Cosmos account.

2. Open the **Connection String** pane.

3. Copy the **CONTACT POINT**, **PORT**, **USERNAME**, and **PRIMARY PASSWORD** values to use in the next steps.

Create the project and the dependencies

The Java sample project that you use in this article is hosted in GitHub. You can run the steps in this doc or download the sample from the azure-cosmos-db-cassandra-java-getting-started repository.

After you download the files, update the connection string information within the `java-examples\src\main\resources\config.properties` file and run it.

```
cassandra_host=<FILLME_with_CONTACT POINT>
cassandra_port = 10350
cassandra_username=<FILLME_with_USERNAME>
cassandra_password=<FILLME_with_PRIMARY PASSWORD>
```

Use the following steps to build the sample from scratch:

1. From the terminal or command prompt, create a new Maven project called Cassandra-demo.

   ```
   mvn archetype:generate -DgroupId=com.azure.cosmosdb.cassandra -DartifactId=cassandra-demo -
   DarchetypeArtifactId=maven-archetype-quickstart -DinteractiveMode=false
   ```

2. Locate the `cassandra-demo` folder. Using a text editor, open the `pom.xml` file that was generated.

   Add the Cassandra dependencies and build plugins required by your project, as shown in the pom.xml file.

3. Under the `cassandra-demo\src\main` folder, create a new folder named `resources`. Under the resources folder, add the config.properties and log4j.properties files:

   - The config.properties file stores the connection endpoint and key values of the Cassandra API account.
   - The log4j.properties file defines the level of logging required for interacting with the Cassandra API.

4. Browse to the `src/main/java/com/azure/cosmosdb/cassandra/` folder. Within the cassandra folder, create another folder named `utils`. The new folder stores the utility classes required to connect to the Cassandra API account.

   Add the CassandraUtils class to create the cluster and to open and close Cassandra sessions. The cluster connects to the Cassandra API account in Azure Cosmos DB and returns a session to access. Use the Configurations class to read connection string information from the config.properties file.

5. The Java sample creates a database with user information such as user name, user ID, and user city. You need to define get and set methods to access user details in the main function.

   Create a User.java class under the `src/main/java/com/azure/cosmosdb/cassandra/` folder with get and set methods.

Add a database and a table

This section describes how to add a database (keyspace) and a table, by using CQL.

1. Under the `src\main\java\com\azure\cosmosdb\cassandra` folder, create a new folder named `repository`.

2. Create the `UserRepository` Java class and add the following code to it:

```
package com.azure.cosmosdb.cassandra.repository;
import java.util.List;
import com.datastax.driver.core.BoundStatement;
import com.datastax.driver.core.PreparedStatement;
import com.datastax.driver.core.Row;
import com.datastax.driver.core.Session;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

/**
 * Create a Cassandra session
 */
public class UserRepository {

    private static final Logger LOGGER = LoggerFactory.getLogger(UserRepository.class);
    private Session session;
    public UserRepository(Session session) {
        this.session = session;
    }

    /**
    * Create keyspace uprofile in cassandra DB
     */

    public void createKeyspace() {
         final String query = "CREATE KEYSPACE IF NOT EXISTS uprofile WITH REPLICATION = { 'class' :
'NetworkTopologyStrategy', 'datacenter1' : 1 }";
        session.execute(query);
        LOGGER.info("Created keyspace 'uprofile'");
    }

    /**
     * Create user table in cassandra DB
     */

    public void createTable() {
        final String query = "CREATE TABLE IF NOT EXISTS uprofile.user (user_id int PRIMARY KEY,
user_name text, user_bcity text)";
        session.execute(query);
        LOGGER.info("Created table 'user'");
    }
}
```

3. Locate the `src\main\java\com\azure\cosmosdb\cassandra` folder, and create a new subfolder named `examples` .

4. Create the `UserProfile` Java class. This class contains the main method that calls the createKeyspace and createTable methods you defined earlier:

```
package com.azure.cosmosdb.cassandra.examples;
import java.io.IOException;
import com.azure.cosmosdb.cassandra.repository.UserRepository;
import com.azure.cosmosdb.cassandra.util.CassandraUtils;
import com.datastax.driver.core.PreparedStatement;
import com.datastax.driver.core.Session;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

/**
 * Example class which will demonstrate following operations on Cassandra Database on CosmosDB
 * - Create Keyspace
 * - Create Table
 * - Insert Rows
 * - Select all data from a table
 * - Select a row from a table
 */

public class UserProfile {

    private static final Logger LOGGER = LoggerFactory.getLogger(UserProfile.class);
    public static void main(String[] s) throws Exception {
        CassandraUtils utils = new CassandraUtils();
        Session cassandraSession = utils.getSession();

        try {
            UserRepository repository = new UserRepository(cassandraSession);
            //Create keyspace in cassandra database
            repository.createKeyspace();
            //Create table in cassandra database
            repository.createTable();

        } finally {
            utils.close();
            LOGGER.info("Please delete your table after verifying the presence of the data in portal or
from CQL");
        }
    }
}
```

Run the app

1. Open a command prompt or terminal window. Paste the following code block.

   This code changes the directory (cd) to the folder path where you created the project. Then, it runs the `mvn clean install` command to generate the `cosmosdb-cassandra-examples.jar` file within the target folder. Finally, it runs the Java application.

   ```
   cd cassandra-demo

   mvn clean install

   java -cp target/cosmosdb-cassandra-examples.jar com.azure.cosmosdb.cassandra.examples.UserProfile
   ```

   The terminal window displays notifications that the keyspace and table are created.

2. Now, in the Azure portal, open **Data Explorer** to confirm that the keyspace and table were created.

Next steps

In this tutorial, you've learned how to create a Cassandra API account in Azure Cosmos DB, a database, and a table

by using a Java application. You can now proceed to the next article:

load sample data to the Cassandra API table.

Tutorial: Load sample data into a Cassandra API table in Azure Cosmos DB

11/2/2020 • 2 minutes to read • Edit Online

APPLIES TO: Cassandra API

As a developer, you might have applications that use key/value pairs. You can use Cassandra API account in Azure Cosmos DB to store and manage key/value data. This tutorial shows how to load sample user data to a table in a Cassandra API account in Azure Cosmos DB by using a Java application. The Java application uses the Java driver and loads user data such as user ID, user name, and user city.

This tutorial covers the following tasks:

- Load data into a Cassandra table
- Run the app

If you don't have an Azure subscription, create a free account before you begin.

Prerequisites

- This article belongs to a multi-part tutorial. Before you start with this doc, make sure to create the Cassandra API account, keyspace, and table.

Load data into the table

Use the following steps to load data into your Cassandra API table:

1. Open the "UserRepository.java" file under the "src\main\java\com\azure\cosmosdb\cassandra" folder and append the code to insert the user_id, user_name and user_bcity fields into the table:

```
/**
* Insert a row into user table
*
* @param id   user_id
* @param name user_name
* @param city user_bcity
*/
public void insertUser(PreparedStatement statement, int id, String name, String city) {
     BoundStatement boundStatement = new BoundStatement(statement);
     session.execute(boundStatement.bind(id, name, city));
}

/**
* Create a PrepareStatement to insert a row to user table
*
* @return PreparedStatement
*/
public PreparedStatement prepareInsertStatement() {
   final String insertStatement = "INSERT INTO  uprofile.user (user_id, user_name , user_bcity) VALUES
(?,?,?)";
return session.prepare(insertStatement);
}
```

2. Open the "UserProfile.java" file under the "src\main\java\com\azure\cosmosdb\cassandra" folder. This class contains the main method that calls the createKeyspace and createTable methods you defined earlier. Now

append the following code to insert some sample data into the Cassandra API table.

```
//Insert rows into user table
PreparedStatement preparedStatement = repository.prepareInsertStatement();
  repository.insertUser(preparedStatement, 1, "JohnH", "Seattle");
  repository.insertUser(preparedStatement, 2, "EricK", "Spokane");
  repository.insertUser(preparedStatement, 3, "MatthewP", "Tacoma");
  repository.insertUser(preparedStatement, 4, "DavidA", "Renton");
  repository.insertUser(preparedStatement, 5, "PeterS", "Everett");
```

Run the app

Open a command prompt or terminal window and change the folder path to where you have created the project. Run the “mvn clean install” command to generate the cosmosdb-cassandra-examples.jar file within the target folder and run the application.

```
cd "cassandra-demo"

mvn clean install

java -cp target/cosmosdb-cassandra-examples.jar com.azure.cosmosdb.cassandra.examples.UserProfile
```

You can now open Data Explorer in the Azure portal to confirm that the user information is added to the table.

Next steps

In this tutorial, you've learned how to load sample data to a Cassandra API account in Azure Cosmos DB. You can now proceed to the next article:

Query data from the Cassandra API account

Tutorial: Query data from a Cassandra API account in Azure Cosmos DB

11/2/2020 • 2 minutes to read • Edit Online

APPLIES TO: Cassandra API

As a developer, you might have applications that use key/value pairs. You can use a Cassandra API account in Azure Cosmos DB to store and query the key/value data. This tutorial shows how to query user data from a Cassandra API account in Azure Cosmos DB by using a Java application. The Java application uses the Java driver and queries user data such as user ID, user name, and user city.

This tutorial covers the following tasks:

- Query data from a Cassandra table
- Run the app

If you don't have an Azure subscription, create a free account before you begin.

Prerequisites

- This article belongs to a multi-part tutorial. Before you start, make sure to complete the previous steps to create the Cassandra API account, keyspace, table, and load sample data into the table.

Query data

Use the following steps to query data from your Cassandra API account:

1. Open the `UserRepository.java` file under the folder `src\main\java\com\azure\cosmosdb\cassandra`. Append the following code block. This code provides three methods:
   - To query all users in the database
   - To query a specific user filtered by user ID
   - To delete a table

```
/**
* Select all rows from user table
*/
public void selectAllUsers() {

  final String query = "SELECT * FROM uprofile.user";
  List<Row> rows = session.execute(query).all();

  for (Row row : rows) {
     LOGGER.info("Obtained row: {} | {} | {} ", row.getInt("user_id"), row.getString("user_name"),
row.getString("user_bcity"));
  }
}

/**
* Select a row from user table
*
* @param id user_id
*/
public void selectUser(int id) {
   final String query = "SELECT * FROM uprofile.user where user_id = 3";
   Row row = session.execute(query).one();

   LOGGER.info("Obtained row: {} | {} | {} ", row.getInt("user_id"), row.getString("user_name"),
row.getString("user_bcity"));
}

/**
* Delete user table.
*/
public void deleteTable() {
  final String query = "DROP TABLE IF EXISTS uprofile.user";
  session.execute(query);
}
```

2. Open the `UserProfile.java` file under the folder `src\main\java\com\azure\cosmosdb\cassandra`. This class contains the main method that calls the createKeyspace and createTable, insert data methods you defined earlier. Now append the following code that queries all users or a specific user:

```
LOGGER.info("Select all users");
repository.selectAllUsers();

LOGGER.info("Select a user by id (3)");
repository.selectUser(3);

LOGGER.info("Delete the users profile table");
repository.deleteTable();
```

Run the Java app

1. Open a command prompt or terminal window. Paste the following code block.

   This code changes the directory (cd) to the folder path where you created the project. Then, it runs the `mvn clean install` command to generate the `cosmosdb-cassandra-examples.jar` file within the target folder. Finally, it runs the Java application.

```
cd "cassandra-demo"

mvn clean install

java -cp target/cosmosdb-cassandra-examples.jar com.azure.cosmosdb.cassandra.examples.UserProfile
```

2. Now, in the Azure portal, open the **Data Explorer** and confirm that the user table is deleted.

Clean up resources

When they're no longer needed, you can delete the resource group, Azure Cosmos account, and all the related resources. To do so, select the resource group for the virtual machine, select **Delete**, and then confirm the name of the resource group to delete.

Next steps

In this tutorial, you've learned how to query data from a Cassandra API account in Azure Cosmos DB. You can now proceed to the next article:

Migrate data to Cassandra API account

Tutorial: Migrate your data to Cassandra API account in Azure Cosmos DB

11/2/2020 • 4 minutes to read • Edit Online

APPLIES TO: Cassandra API

As a developer, you might have existing Cassandra workloads that are running on-premises or in the cloud, and you might want to migrate them to Azure. You can migrate such workloads to a Cassandra API account in Azure Cosmos DB. This tutorial provides instructions on different options available to migrate Apache Cassandra data into the Cassandra API account in Azure Cosmos DB.

This tutorial covers the following tasks:

- Plan for migration
- Prerequisites for migration
- Migrate data using cqlsh COPY command
- Migrate data using Spark

If you don't have an Azure subscription, create a free account before you begin.

Prerequisites for migration

- **Estimate your throughput needs:** Before migrating data to the Cassandra API account in Azure Cosmos DB, you should estimate the throughput needs of your workload. In general, it's recommended to start with the average throughput required by the CRUD operations and then include the additional throughput required for the Extract Transform Load (ETL) or spiky operations. You need the following details to plan for migration:
  - **Existing data size or estimated data size:** Defines the minimum database size and throughput requirement. If you are estimating data size for a new application, you can assume that the data is uniformly distributed across the rows and estimate the value by multiplying with the data size.
  - **Required throughput:** Approximate read (query/get) and write (update/delete/insert) throughput rate. This value is required to compute the required request units along with steady state data size.
  - **The schema:** Connect to your existing Cassandra cluster through cqlsh and export the schema from Cassandra:

    ```
    cqlsh [IP] "-e DESC SCHEMA" > orig_schema.cql
    ```

    After you identify the requirements of your existing workload, you should create an Azure Cosmos account, database, and containers according to the gathered throughput requirements.
  - **Determine the RU charge for an operation:** You can determine the RUs by using any of the SDKs supported by the Cassandra API. This example shows the .NET version of getting RU charges.

```
var tableInsertStatement = table.Insert(sampleEntity);
var insertResult = await tableInsertStatement.ExecuteAsync();

foreach (string key in insertResult.Info.IncomingPayload)
  {
     byte[] valueInBytes = customPayload[key];
     double value = Encoding.UTF8.GetString(valueInBytes);
     Console.WriteLine($"CustomPayload:  {key}: {value}");
  }
```

- **Allocate the required throughput:** Azure Cosmos DB can automatically scale storage and throughput as your requirements grow. You can estimate your throughput needs by using the Azure Cosmos DB request unit calculator.
- **Create tables in the Cassandra API account:** Before you start migrating data, pre-create all your tables from the Azure portal or from cqlsh. If you are migrating to an Azure Cosmos account that has database level throughput, make sure to provide a partition key when creating the Azure Cosmos containers.
- **Increase throughput:** The duration of your data migration depends on the amount of throughput you provisioned for the tables in Azure Cosmos DB. Increase the throughput for the duration of migration. With the higher throughput, you can avoid rate limiting and migrate in less time. After you've completed the migration, decrease the throughput to save costs. It's also recommended to have the Azure Cosmos account in the same region as your source database.
- **Enable TLS:** Azure Cosmos DB has strict security requirements and standards. Be sure to enable TLS when you interact with your account. When you use CQL with SSH, you have an option to provide TLS information.

Options to migrate data

You can move data from existing Cassandra workloads to Azure Cosmos DB by using the following options:

- Using cqlsh COPY command
- Using Spark

Migrate data using cqlsh COPY command

The CQL COPY command is used to copy local data to the Cassandra API account in Azure Cosmos DB. Use the following steps to copy data:

1. Get your Cassandra API account's connection string information:
   - Sign in to the Azure portal, and navigate to your Azure Cosmos account.
   - Open the **Connection String** pane that contains all the information that you need to connect to your Cassandra API account from cqlsh.
2. Sign in to cqlsh using the connection information from the portal.
3. Use the CQL COPY command to copy local data to the Cassandra API account.

```
COPY exampleks.tablename FROM filefolderx/*.csv
```

Migrate data using Spark

Use the following steps to migrate data to the Cassandra API account with Spark:

- Provision an Azure Databricks cluster or an HDInsight cluster
- Move data to the destination Cassandra API endpoint by using the table copy operation

Migrating data by using Spark jobs is a recommended option if you have data residing in an existing cluster in Azure virtual machines or any other cloud. This option requires Spark to be set up as an intermediary for one time or regular ingestion. You can accelerate this migration by using Azure ExpressRoute connectivity between on-premises and Azure.

Clean up resources

When they're no longer needed, you can delete the resource group, the Azure Cosmos account, and all the related resources. To do so, select the resource group for the virtual machine, select **Delete**, and then confirm the name of the resource group to delete.

Next steps

In this tutorial, you've learned how to migrate your data to Cassandra API account in Azure Cosmos DB. You can now proceed to the following article to learn about other Azure Cosmos DB concepts:

Tunable data consistency levels in Azure Cosmos DB

Frequently asked questions about the Cassandra API in Azure Cosmos DB

12/4/2020 • 11 minutes to read • Edit Online

APPLIES TO: Cassandra API

This article describes the functionality differences between Apache Cassandra and Cassandra API in Azure Cosmos DB. It also provides answers to frequently asked questions about the Cassandra API in Azure Cosmos DB.

Key differences between Apache Cassandra and the Cassandra API

- Apache Cassandra recommends a 100-MB limit on the size of a partition key. The Cassandra API for Azure Cosmos DB allows up to 20 GB per partition.
- Apache Cassandra allows you to disable durable commits. You can skip writing to the commit log and go directly to the memtables. This can lead to data loss if the node goes down before memtables are flushed to SSTables on disk. Azure Cosmos DB always does durable commits to help prevent data loss.
- Apache Cassandra can see diminished performance if the workload involves many replacements or deletions. The reason is tombstones that the read workload needs to skip over to fetch the latest data. The Cassandra API won't see diminished read performance when the workload has many replacements or deletions.
- During scenarios of high replacement workloads, compaction needs to run to merge SSTables on disk. (A merge is needed because Apache Cassandra's writes are append only. Multiple updates are stored as individual SSTable entries that need to be periodically merged). This situation can also lead to lowered read performance during compaction. This performance impact doesn't happen in the Cassandra API because the API doesn't implement compaction.
- Setting a replication factor of 1 is possible with Apache Cassandra. However, it leads to low availability if the only node with the data goes down. This is not an issue with the Cassandra API for Azure Cosmos DB because there is always a replication factor of 4 (quorum of 3).
- Adding or removing nodes in Apache Cassandra requires manual intervention, along with high CPU usage on the new node while existing nodes move some of their token ranges to the new node. This situation is the same when you're decommissioning an existing node. However, the Cassandra API scales out without any issues observed in the service or application.
- There is no need to set **num_tokens** on each node in the cluster as in Apache Cassandra. Azure Cosmos DB fully manages nodes and token ranges.
- The Cassandra API is fully managed. You don't need the **nodetool** commands, such as repair and decommission, that are used in Apache Cassandra.

Other frequently asked questions

What protocol version does the Cassandra API support?

The Cassandra API for Azure Cosmos DB supports CQL version 3.x. Its CQL compatibility is based on the public Apache Cassandra GitHub repository. If you have feedback about supporting other protocols, let us know via user voice feedback or send email to askcosmosdbcassandra@microsoft.com.

Why is choosing throughput for a table a requirement?

Azure Cosmos DB sets the default throughput for your container based on where you create the table from: Azure portal or CQL.

Azure Cosmos DB provides guarantees for performance and latency, with upper bounds on operations. These

guarantees are possible when the engine can enforce governance on the tenant's operations. Setting throughput ensures that you get the guaranteed throughput and latency, because the platform reserves this capacity and guarantees operation success. You can elastically change throughput to benefit from the seasonality of your application and save costs.

The throughput concept is explained in the Request Units in Azure Cosmos DB article. The throughput for a table is equally distributed across the underlying physical partitions.

What is the throughput of a table that's created through CQL?

Azure Cosmos DB uses Request Units per second (RU/s) as a currency for providing throughput. Tables created through CQL have 400 RU by default. You can change the RU from the Azure portal.

CQL

```
CREATE TABLE keyspaceName.tablename (user_id int PRIMARY KEY, lastname text) WITH
cosmosdb_provisioned_throughput=1200
```

.NET

```
int provisionedThroughput = 400;
var simpleStatement = new SimpleStatement($"CREATE TABLE {keyspaceName}.{tableName} (user_id int PRIMARY KEY,
lastname text)");
var outgoingPayload = new Dictionary<string, byte[]>();
outgoingPayload["cosmosdb_provisioned_throughput"] = Encoding.UTF8.GetBytes(provisionedThroughput.ToString());
simpleStatement.SetOutgoingPayload(outgoingPayload);
```

What happens when throughput is used up?

Azure Cosmos DB provides guarantees for performance and latency, with upper bounds on operations. These guarantees are possible when the engine can enforce governance on the tenant's operations. Setting throughput ensures that you get the guaranteed throughput and latency, because the platform reserves this capacity and guarantees operation success.

When you go over this capacity, you get the following error message that indicates your capacity was used up:

0x1001 Overloaded: the request can't be processed because "Request Rate is large"

It's essential to see what operations (and their volume) cause this issue. You can get an idea about consumed capacity going over the provisioned capacity with metrics on the Azure portal. Then you need to ensure that capacity is consumed nearly equally across all underlying partitions. If you see that one partition is consuming most of the throughput, you have skew of workload.

Metrics are available that show you how throughput is used over hours, over days, and per seven days, across partitions or in aggregate. For more information, see Monitoring and debugging with metrics in Azure Cosmos DB.

Diagnostic logs are explained in the Azure Cosmos DB diagnostic logging article.

Does the primary key map to the partition key concept of Azure Cosmos DB?

Yes, the partition key is used to place the entity in the right location. In Azure Cosmos DB, it's used to find the right logical partition that's stored on a physical partition. The partitioning concept is well explained in the Partition and scale in Azure Cosmos DB article. The essential takeaway here is that a logical partition shouldn't go over the 20-GB limit.

What happens when I get a notification that a partition is full?

Azure Cosmos DB is a system based on service-level agreement (SLA). It provides unlimited scale, with guarantees for latency, throughput, availability, and consistency. This unlimited storage is based on horizontal scale-out of data, using partitioning as the key concept. The partitioning concept is well explained in the Partition and scale in Azure

Cosmos DB article.

You should adhere to the 20-GB limit on the number of entities or items per logical partition. To ensure that your application scales well, we recommend that you *not* create a hot partition by storing all information in one partition and querying it. This error can come only if your data is skewed: that is, you have lot of data for one partition key (more than 20 GB). You can find the distribution of data by using the storage portal. The way to fix this error is to re-create the table and choose a granular primary (partition key), which allows better distribution of data.

Can I use the Cassandra API as a key value store with millions or billions of partition keys?

Azure Cosmos DB can store unlimited data by scaling out the storage. This storage is independent of the throughput. Yes, you can always use the Cassandra API just to store and retrieve keys and values by specifying the right primary/partition key. These individual keys get their own logical partition and sit atop a physical partition without issues.

Can I create more than one table with the Cassandra API?

Yes, it's possible to create more than one table with the Cassandra API. Each of those tables is treated as unit for throughput and storage.

Can I create more than one table in succession?

Azure Cosmos DB is resource-governed system for both data and control plane activities. Containers, like collections and tables, are runtime entities that are provisioned for a given throughput capacity. The creation of these containers in quick succession isn't an expected activity and might be throttled. If you have tests that drop or create tables immediately, try to space them out.

What is the maximum number of tables that I can create?

There's no physical limit on the number of tables. If you have a large number of tables (where the total steady size goes over 10 TB of data) that need to be created, not the usual tens or hundreds, send email to askcosmosdbcassandra@microsoft.com.

What is the maximum number of keyspaces that I can create?

There's no physical limit on the number of keyspaces because they're metadata containers. If you have a large number of keyspaces, send email to askcosmosdbcassandra@microsoft.com.

Can I bring in a lot of data after starting from a normal table?

Yes. Assuming uniformly distributed partitions, the storage capacity is automatically managed and increases as you push in more data. So you can confidently import as much data as you need without managing and provisioning nodes and more. But if you're anticipating a lot of immediate data growth, it makes more sense to directly provision for the anticipated throughput rather than starting lower and increasing it immediately.

Can I use YAML file settings to configure API behavior?

The Cassandra API for Azure Cosmos DB provides protocol-level compatibility for executing operations. It hides away the complexity of management, monitoring, and configuration. As a developer/user, you don't need to worry about availability, tombstones, key cache, row cache, bloom filter, and a multitude of other settings. The Cassandra API focuses on providing the read and write performance that you need without the overhead of configuration and management.

Will the Cassandra API support node addition, cluster status, and node status commands?

The Cassandra API simplifies capacity planning and responding to the elasticity demands for throughput and storage. With Azure Cosmos DB, you provision the throughput that you need. Then you can scale it up and down any number of times through the day, without worrying about adding, deleting, or managing nodes. You don't need to use tools for node and cluster management.

What happens with various configuration settings for keyspace creation like simple/network?

Azure Cosmos DB provides global distribution out of the box for availability and low-latency reasons. You don't

need to set up replicas or other things. Writes are always durably quorum committed in any region where you write, while providing performance guarantees.

What happens with various settings for table metadata?

Azure Cosmos DB provides performance guarantees for reads, writes, and throughput. So you don't need to worry about touching any of the configuration settings and accidentally manipulating them. Those settings include bloom filter, caching, read repair chance, gc_grace, and compression memtable_flush_period.

Is time-to-live supported for Cassandra tables?

Yes, TTL is supported.

How can I monitor infrastructure along with throughput?

Azure Cosmos DB is a platform service that helps you increase productivity and not worry about managing and monitoring infrastructure. For example, you don't need to monitor node status, replica status, gc, and OS parameters earlier with various tools. You just need to take care of throughput that's available in portal metrics to see if you're getting throttled, and then increase or decrease that throughput. You can:

- Monitor SLAs
- Use metrics
- Use diagnostic logs

Which client SDKs can work with the Cassandra API?

The Apache Cassandra SDK's client drivers that use CQLv3 were used for client programs. If you have other drivers that you use or if you're facing issues, send mail to askcosmosdbcassandra@microsoft.com.

Are composite partition keys supported?

Yes, you can use regular syntax to create composite partition keys.

Can I use sstableloader for data loading?

No, sstableloader isn't supported.

Can I pair an on-premises Apache Cassandra cluster with the Cassandra API?

At present, Azure Cosmos DB has an optimized experience for a cloud environment without the overhead of operations. If you require pairing, send mail to askcosmosdbcassandra@microsoft.com with a description of your scenario. We're working on an offering to help pair the on-premises or cloud Cassandra cluster with the Cassandra API for Azure Cosmos DB.

Does the Cassandra API provide full backups?

Azure Cosmos DB provides two free full backups taken at four-hour intervals across all APIs. So you don't need to set up a backup schedule.

If you want to modify retention and frequency, send email to askcosmosdbcassandra@microsoft.com or raise a support case. Information about backup capability is provided in the Automatic online backup and restore with Azure Cosmos DB article.

How does the Cassandra API account handle failover if a region goes down?

The Cassandra API borrows from the globally distributed platform of Azure Cosmos DB. To ensure that your application can tolerate datacenter downtime, enable at least one more region for the account in the Azure portal. For more information, see High availability with Azure Cosmos DB.

You can add as many regions as you want for the account and control where it can fail over to by providing a failover priority. To use the database, you need to provide an application there too. When you do so, your customers won't experience downtime.

Does the Cassandra API index all attributes of an entity by default?

No. The Cassandra API supports secondary indexes, which behave in a similar way to Apache Cassandra. The API does not index every attribute by default.

Can I use the new Cassandra API SDK locally with the emulator?

Yes, this is supported. You can find details on how to enable this in the Use the Azure Cosmos DB Emulator for local development and testing article.

How can I migrate data from Apache Cassandra clusters to Azure Cosmos DB?

You can read about migration options in the Migrate your data to Cassandra API account in Azure Cosmos DB tutorial.

Where can I give feedback on Cassandra API features?

Provide feedback via user voice feedback.

Next steps

- Get started with elastically scaling an Azure Cosmos DB Cassandra API account.

Elastically scale an Azure Cosmos DB Cassandra API account

12/4/2020 • 4 minutes to read • Edit Online

APPLIES TO: Cassandra API

There are a variety of options to explore the elastic nature of the Azure Cosmos DB API for Cassandra. To understand how to scale effectively in Azure Cosmos DB, it is important to understand how to provision the right amount of request units (RU/s) to account for the performance demands in your system. To learn more about request units, see the request units article.

For the Cassandra API, you can retrieve the Request Unit charge for individual queries using the .NET and Java SDKs. This is helpful in determining the amount of RU/s you will need to provision in the service.

Handling rate limiting (429 errors)

Azure Cosmos DB will return rate-limited (429) errors if clients consume more resources (RU/s) than the amount that you have provisioned. The Cassandra API in Azure Cosmos DB translates these exceptions to overloaded errors on the Cassandra native protocol.

If your system is not sensitive to latency, it may be sufficient to handle the throughput rate-limiting by using retries. See the Java code sample for how to handle rate limiting transparently by using the Azure Cosmos DB extension for Cassandra retry policy in Java. You can also use the Spark extension to handle rate-limiting.

Manage scaling

If you need to minimize latency, there is a spectrum of options for managing scale and provisioning throughput (RUs) in the Cassandra API:

- Manually by using the Azure portal
- Programmatically by using the control plane features

- Programmatically by using CQL commands with a specific SDK
- Dynamically by using autoscale

The following sections explain the advantages and disadvantages of each approach. You can then decide on the best strategy to balance the scaling needs of your system, the overall cost, and efficiency needs for your solution.

Use the Azure portal

You can scale the resources in Azure Cosmos DB Cassandra API account by using Azure portal. To learn more, see the article on Provision throughput on containers and databases. This article explains the relative benefits of setting throughput at either database or container level in the Azure portal. The terms "database" and "container" mentioned in these articles map to "keyspace" and "table" respectively for the Cassandra API.

The advantage of this method is that it is a straightforward turnkey way to manage throughput capacity on the database. However, the disadvantage is that in many cases, your approach to scaling may require certain levels of automation to be both cost-effective and high performing. The next sections explain the relevant scenarios and methods.

Use the control plane

The Azure Cosmos DB's API for Cassandra provides the capability to adjust throughput programmatically by using our various control-plane features. See the Azure Resource Manager, PowerShell, and Azure CLI articles for guidance and samples.

The advantage of this method is that you can automate the scaling up or down of resources based on a timer to account for peak activity, or periods of low activity. Take a look at our sample here for how to accomplish this using Azure Functions and PowerShell.

A disadvantage with this approach may be that you cannot respond to unpredictable changing scale needs in real-time. Instead, you may need to leverage the application context in your system, at the client/SDK level, or using Autoscale.

Use CQL queries with a specific SDK

You can scale the system dynamically with code by executing the CQL ALTER commands for the given database or container.

The advantage of this approach is that it allows you to respond to scale needs dynamically and in a custom way that suits your application. With this approach, you can still leverage the standard RU/s charges and rates. If your system's scale needs are mostly predictable (around 70% or more), using SDK with CQL may be a more cost-effective method of auto-scaling than using autoscale. The disadvantage of this approach is that it can be quite complex to implement retries while rate limiting may increase latency.

Use autoscale provisioned throughput

In addition to standard (manual) or programmatic way of provisioning throughput, you can also configure Azure cosmos containers in autoscale provisioned throughput. Autoscale will automatically and instantly scale to your consumption needs within specified RU ranges without compromising SLAs. To learn more, see the Create Azure Cosmos containers and databases in autoscale article.

The advantage of this approach is that it is the easiest way to manage the scaling needs in your system. It will not apply rate-limiting **within the configured RU ranges**. The disadvantage is that, if the scaling needs in your system are predictable, autoscale may be a less cost-effective way of handling your scaling needs than using the bespoke control plane or SDK level approaches mentioned above.

To set or alter max throughput (RUs) for autoscale using CQL, use the following (replacing keyspace/table name accordingly):

```
# to set max throughput (RUs) for autoscale at keyspace level:
create keyspace <keyspace name> WITH cosmosdb_autoscale_max_throughput=5000;

# to alter max throughput (RUs) for autoscale at keyspace level:
alter keyspace <keyspace name> WITH cosmosdb_autoscale_max_throughput=4000;

# to set max throughput (RUs) for autoscale at table level:
create table <keyspace name>.<table name> (pk int PRIMARY KEY, ck int) WITH
cosmosdb_autoscale_max_throughput=5000;

# to alter max throughput (RUs) for autoscale at table level:
alter table <keyspace name>.<table name> WITH cosmosdb_autoscale_max_throughput=4000;
```

Next steps

- Get started with creating a Cassandra API account, database, and a table by using a Java application

Partitioning in Azure Cosmos DB Cassandra API

11/2/2020 • 5 minutes to read • Edit Online

APPLIES TO: Cassandra API

This article describes how partitioning works in Azure Cosmos DB Cassandra API.

Cassandra API uses partitioning to scale the individual tables in a keyspace to meet the performance needs of your application. Partitions are formed based on the value of a partition key that is associated with each record in a table. All the records in a partition have the same partition key value. Azure Cosmos DB transparently and automatically manages the placement of partitions across the physical resources to efficiently satisfy the scalability and performance needs of the table. As the throughput and storage requirements of an application increase, Azure Cosmos DB moves and balances the data across a greater number of physical machines.

From the developer perspective, partitioning behaves in the same way for Azure Cosmos DB Cassandra API as it does in native Apache Cassandra. However, there are some differences behind the scenes.

Differences between Apache Cassandra and Azure Cosmos DB

In Azure Cosmos DB, each machine on which partitions are stored is itself referred to as a physical partition. The physical partition is akin to a Virtual Machine; a dedicated compute unit, or set of physical resources. Each partition stored on this compute unit is referred to as a logical partition in Azure Cosmos DB. If you are already familiar with Apache Cassandra, you can think of logical partitions in the same way that you think of regular partitions in Cassandra.

Apache Cassandra recommends a 100-MB limit on the size of a data that can be stored in a partition. The Cassandra API for Azure Cosmos DB allows up to 20 GB per logical partition, and up to 30GB of data per physical partition. In Azure Cosmos DB, unlike Apache Cassandra, compute capacity available in the physical partition is expressed using a single metric called request units, which allows you to think of your workload in terms of requests (reads or writes) per second, rather than cores, memory, or IOPS. This can make capacity planning more straight forward, once you understand the cost of each request. Each physical partition can have up to 10000 RUs of compute available to it. You can learn more about scalability options by reading our article on elastic scale in Cassandra API.

In Azure Cosmos DB, each physical partition consists of a set of replicas, also known as replica sets, with at least 4 replicas per partition. This is in contrast to Apache Cassandra, where setting a replication factor of 1 is possible. However, this leads to low availability if the only node with the data goes down. In Cassandra API there is always a replication factor of 4 (quorum of 3). Azure Cosmos DB automatically manages replica sets, while these need to be maintained using various tools in Apache Cassandra.

Apache Cassandra has a concept of tokens, which are hashes of partition keys. The tokens are based on a murmur3 64 byte hash, with values ranging from -2^63 to -2^63 - 1. This range is commonly referred to as the "token ring" in Apache Cassandra. The token ring is distributed into token ranges, and these ranges are divided amongst the nodes present in a native Apache Cassandra cluster. Partitioning for Azure Cosmos DB is implemented in a similar way, except it uses a different hash algorithm, and has a larger internal token ring. However, externally we expose the same token range as Apache Cassandra, i.e., -2^63 to -2^63 - 1.

Primary key

All tables in Cassandra API must have a `primary key` defined. The syntax for a primary key is shown below:

```
column_name cql_type_definition PRIMARY KEY
```

Suppose we want to create a user table, which stores messages for different users:

```
CREATE TABLE uprofile.user (
   id UUID PRIMARY KEY,
   user text,
   message text);
```

In this design, we have defined the `id` field as the primary key. The primary key functions as the identifier for the record in the table and it is also used as the partition key in Azure Cosmos DB. If the primary key is defined in the previously described way, there will only be a single record in each partition. This will result in a perfectly horizontal and scalable distribution when writing data to the database, and is ideal for key-value lookup use cases. The application should provide the primary key whenever reading data from the table, to maximize read performance.

Compound primary key

Apache Cassandra also has a concept of `compound keys`. A compound `primary key` consists of more than one column; the first column is the `partition key`, and any additional columns are the `clustering keys`. The syntax for a `compound primary key` is shown below:

```
PRIMARY KEY (partition_key_column_name, clustering_column_name [, ...])
```

Suppose we want to change the above design and make it possible to efficiently retrieve messages for a given user:

```
CREATE TABLE uprofile.user (
   user text,
   id int,
   message text,
   PRIMARY KEY (user, id));
```

In this design, we are now defining `user` as the partition key, and `id` as the clustering key. You can define as many clustering keys as you wish, but each value (or a combination of values) for the clustering key must be unique in order to result in multiple records being added to the same partition, for example:

```
insert into uprofile.user (user, id, message) values ('theo', 1, 'hello');
insert into uprofile.user (user, id, message) values ('theo', 2, 'hello again');
```

When data is returned, it is sorted by the clustering key, as expected in Apache Cassandra:

```
tvkcassandra@cqlsh> select * from uprofile.user where user = 'theo';

 user | id | message
------+----+------------
 theo |  1 |       hello
 theo |  2 | hello again

(2 rows)
tvkcassandra@cqlsh>
```

With data modeled in this way, multiple records can be assigned to each partition, grouped by user. We can thus issue a query that is efficiently routed by the `partition key` (in this case, `user`) to get all the messages for a given user.

Composite partition key

Composite partition keys work essentially the same way as compound keys, except that you can specify multiple columns as a composite partition key. The syntax of composite partition keys is shown below:

```
PRIMARY KEY (
   (partition_key_column_name[, ...]),
    clustering_column_name [, ...]);
```

For example, you can have the following, where the unique combination of `firstname` and `lastname` would form the partition key, and `id` is the clustering key:

```
CREATE TABLE uprofile.user (
   firstname text,
   lastname text,
   id int,
   message text,
   PRIMARY KEY ((firstname, lastname), id) );
```

Next steps

- Learn about partitioning and horizontal scaling in Azure Cosmos DB.
- Learn about provisioned throughput in Azure Cosmos DB.
- Learn about global distribution in Azure Cosmos DB.

Secondary indexing in Azure Cosmos DB Cassandra API

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: Cassandra API

The Cassandra API in Azure Cosmos DB leverages the underlying indexing infrastructure to expose the indexing strength that is inherent in the platform. However, unlike the core SQL API, Cassandra API in Azure Cosmos DB does not index all attributes by default. Instead, it supports secondary indexing to create an index on certain attributes, which behaves the same way as Apache Cassandra.

In general, it's not advised to execute filter queries on the columns that aren't partitioned. You must use ALLOW FILTERING syntax explicitly, which results in an operation that may not perform well. In Azure Cosmos DB you can run such queries on low cardinality attributes because they fan out across partitions to retrieve the results.

It's not advised to create an index on a frequently updated column. It is prudent to create an index when you define the table. This ensures that data and indexes are in a consistent state. In case you create a new index on the existing data, currently, you can't track the index progress change for the table. If you need to track the progress for this operation, you have to request the progress change via a support ticket.

> **NOTE**
>
> Secondary index is not supported on the following objects:
>
> - data types such as frozen collection types, decimal, and variant types.
> - Static columns
> - Clustering keys

Indexing example

First, create a sample keyspace and table by running the following commands on the CQL shell prompt:

```
CREATE KEYSPACE sampleks WITH REPLICATION = {'class' : 'SimpleStrategy'};
CREATE TABLE sampleks.t1(user_id int PRIMARY KEY, lastname text) WITH cosmosdb_provisioned_throughput=400;
```

Then, insert sample user data with the following commands:

```
insert into sampleks.t1(user_id,lastname) values (1, 'nishu');
insert into sampleks.t1(user_id,lastname) values (2, 'vinod');
insert into sampleks.t1(user_id,lastname) values (3, 'bat');
insert into sampleks.t1(user_id,lastname) values (5, 'vivek');
insert into sampleks.t1(user_id,lastname) values (6, 'siddhesh');
insert into sampleks.t1(user_id,lastname) values (7, 'akash');
insert into sampleks.t1(user_id,lastname) values (8, 'Theo');
insert into sampleks.t1(user_id,lastname) values (9, 'jagan');
```

If you try executing the following statement, you will run into an error that asks you to use `ALLOW FILTERING`:

```
select user_id, lastname from sampleks.t1 where lastname='nishu';
```

Although the Cassandra API supports ALLOW FILTERING, as mentioned in the previous section, it's not recommended. You should instead create an index in the as shown in the following example:

```
CREATE INDEX ON sampleks.t1 (lastname);
```

After creating an index on the "lastname" field, you can now run the previous query successfully. With Cassandra API in Azure Cosmos DB, you do not have to provide an index name. A default index with format `tablename_columnname_idx` is used. For example, `t1_lastname_idx` is the index name for the previous table.

Dropping the index

You need to know what the index name is to drop the index. Run the `desc schema` command to get the description of your table. The output of this command includes the index name in the format `CREATE INDEX tablename_columnname_idx ON keyspacename.tablename(columnname)`. You can then use the index name to drop the index as shown in the following example:

```
drop index sampleks.t1_lastname_idx;
```

Next steps

- Learn how automatic indexing works in Azure Cosmos DB
- Apache Cassandra features supported by Azure Cosmos DB Cassandra API

Apache Cassandra and Azure Cosmos DB consistency levels

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: Cassandra API

Unlike Azure Cosmos DB, Apache Cassandra does not natively provide precisely defined consistency guarantees. Instead, Apache Cassandra provides a write consistency level and a read consistency level, to enable the high availability, consistency, and latency tradeoffs. When using Azure Cosmos DB's Cassandra API:

- The write consistency level of Apache Cassandra is mapped to the default consistency level configured on your Azure Cosmos account. Consistency for a write operation (CL) can't be changed on a per-request basis.
- Azure Cosmos DB will dynamically map the read consistency level specified by the Cassandra client driver to one of the Azure Cosmos DB consistency levels configured dynamically on a read request.

Mapping consistency levels

The following table illustrates how the native Cassandra consistency levels are mapped to the Azure Cosmos DB's consistency levels when using Cassandra API:

Apache Cassandra Write consistency level	Default Consistency Level (set on Cosmos account)	Apache Cassandra Read consistency level	Cosmos Consistency Level (dynamically set on read request)	Guarantees offered by native Apache Cassandra	Guarantees offered by Cosmos Cassandra API
ALL	Strong	ALL	Strong	Linearizability	Linearizability
		QUORUM, SERIAL	Strong	Dirty Reads; Global monotonic reads	Linearizability
		LOCAL_QUORUM	Bounded Staleness	Dirty Reads; monotonic reads within a region	Linearizabile reads within the region where the read is performed Monotonic reads in all other regions
		All other Cassandra's read consistency levels.	Bounded Staleness	Dirty Reads; non-monotonic reads	Linearizabile reads within the region where the read is performed Monotonic reads in all other regions
EACH_QUORUM	Strong	ALL	Strong	Linearizability	Linearizability
		QUORUM, SERIAL	Strong	Dirty Reads; Monotonic reads	Linearizability
		LOCAL_QUORUM, LOCAL_SERIAL	Bounded Staleness	Dirty Reads; monotonic reads within a region	Linearizabile reads within the region where the read is performed Monotonic reads in all other regions
		All other Cassandra's read consistency levels.	Consistent Prefix	Stale Reads; Dirty Reads	Consistent Prefix
QUORUM	Strong	ALL	Strong	Linearizability	Linearizability
		QUORUM, SERIAL	Strong	Linearizability	Linearizability
		LOCAL_QUORUM, LOCAL_SERIAL	Bounded Staleness	Stale Reads, Dirty Reads; monotonic reads within region	Linearizabile reads within the region where the read is performed Monotonic reads in all other regions
		All other Cassandra's read consistency levels.	Consistent Prefix	Stale Reads; Dirty Reads	Consistent Prefix
LOCAL_QUORUM, THREE	Bounded Staleness	ALL	Not Allowed	Linearizability	Not Allowed
		QUORUM, SERIAL	Not Allowed	Stale Reads; global monotonic reads	Not Allowed
		LOCAL_QUORUM, LOCAL_SERIAL	Bounded Staleness	Stale Reads; monotonic reads within region	Linearizabile reads within the region where the read is performed Monotonic reads in all other regions
		All other Cassandra's read consistency levels.	Consistent Prefix	Stale Reads; Dirty Reads	Consistent Prefix
All other Cassandra's write consistency levels	Consistent Prefix	ALL	Not Allowed	Linearizability	Not Allowed
		QUORUM, SERIAL	Not Allowed	Stale Reads; global monotonic reads	Not Allowed
		LOCAL_QUORUM, LOCAL_SERIAL	Bounded Staleness	Stale Reads; monotonic reads within a region	Linearizabile reads within the region where the read is performed Monotonic reads in all other regions
		All other Cassandra's read consistency levels.	Consistent Prefix	Stale Reads; Dirty Reads	Consistent Prefix

If your Azure Cosmos account is configured with a consistency level other than the strong consistency, you can find out the probability that your clients may get strong and consistent reads for your workloads by looking at the *Probabilistically Bounded Staleness* (PBS) metric. This metric is exposed in the Azure portal, to learn more, see Monitor Probabilistically Bounded Staleness (PBS) metric.

Probabilistic bounded staleness shows how eventual is your eventual consistency. This metric provides an insight into how often you can get a stronger consistency than the consistency level that you have currently configured on your Azure Cosmos account. In other words, you can see the probability (measured in milliseconds) of getting strongly consistent reads for a combination of write and read regions.

Next steps

Learn more about global distribution and consistency levels for Azure Cosmos DB:

- Global distribution overview
- Consistency Level overview
- High availability

Migrate data to Azure Cosmos DB Cassandra API account using Striim

12/4/2020 • 8 minutes to read • Edit Online

APPLIES TO: Cassandra API

The Striim image in the Azure marketplace offers continuous real-time data movement from data warehouses and databases to Azure. While moving the data, you can perform in-line denormalization, data transformation, enable real-time analytics, and data reporting scenarios. It's easy to get started with Striim to continuously move enterprise data to Azure Cosmos DB Cassandra API. Azure provides a marketplace offering that makes it easy to deploy Striim and migrate data to Azure Cosmos DB.

This article shows how to use Striim to migrate data from an **Oracle database** to an **Azure Cosmos DB Cassandra API account**.

Prerequisites

- If you don't have an Azure subscription, create a free account before you begin.
- An Oracle database running on-premises with some data in it.

Deploy the Striim marketplace solution

1. Sign into the Azure portal.
2. Select **Create a resource** and search for **Striim** in the Azure marketplace. Select the first option and **Create**.

3. Next, enter the configuration properties of the Striim instance. The Striim environment is deployed in a virtual machine. From the **Basics** pane, enter the **VM user name**, **VM password** (this password is used to SSH into the VM). Select your **Subscription**, **Resource Group**, and **Location details** where you'd like to

deploy Striim. Once complete, select **OK**.

4. In the **Striim Cluster settings** pane, choose the type of Striim deployment and the virtual machine size.

SETTING	VALUE	DESCRIPTION
Striim deployment type	Standalone	Striim can run in a **Standalone** or **Cluster** deployment types. Standalone mode will deploy the Striim server on a single virtual machine and you can select the size of the VMs depending on your data volume. Cluster mode will deploy the Striim server on two or more VMs with the selected size. Cluster environments with more than 2 nodes offer automatic high availability and failover. In this tutorial, you can select Standalone option. Use the default "Standard_F4s" size VM.
Name of the Striim cluster	<Striim_cluster_Name>	Name of the Striim cluster.
Striim cluster password	<Striim_cluster_password>	Password for the cluster.

After you fill the form, select **OK** to continue.

5. In the **Striim access settings** pane, configure the **Public IP address** (choose the default values), **Domain name for Striim**, **Admin password** that you'd like to use to login to the Striim UI. Configure a VNET and Subnet (choose the default values). After filling in the details, select **OK** to continue.

6. Azure will validate the deployment and make sure everything looks good; validation takes few minutes to complete. After the validation is completed, select **OK**.

7. Finally, review the terms of use and select **Create** to create your Striim instance.

Configure the source database

In this section, you configure the Oracle database as the source for data movement. You'll need the Oracle JDBC driver to connect to Oracle. To read changes from your source Oracle database, you can either use the LogMiner or the XStream APIs. The Oracle JDBC driver must be present in Striim's Java classpath to read, write, or persist data from Oracle database.

Download the ojdbc8.jar driver onto your local machine. You will install it in the Striim cluster later.

Configure target database

In this section, you will configure the Azure Cosmos DB Cassandra API account as the target for data movement.

1. Create an Azure Cosmos DB Cassandra API account using the Azure portal.

2. Navigate to the **Data Explorer** pane in your Azure Cosmos account. Select **New Table** to create a new container. Assume that you are migrating *products* and *orders* data from Oracle database to Azure Cosmos DB. Create a new Keyspace named **StriimDemo** with an Orders container. Provision the container with **1000 RUs**(this example uses 1000 RUs, but you should use the throughput estimated for your workload), and **/ORDER_ID** as the primary key. These values will differ depending on your source data.

Configure Oracle to Azure Cosmos DB data flow

1. Now, let's get back to Striim. Before interacting with Striim, install the Oracle JDBC driver that you downloaded earlier.

2. Navigate to the Striim instance that you deployed in the Azure portal. Select the **Connect** button in the upper menu bar and from the **SSH** tab, copy the URL in **Login using VM local account** field.

3. Open a new terminal window and run the SSH command you copied from the Azure portal. This article uses terminal in a MacOS, you can follow the similar instructions using PuTTY or a different SSH client on a Windows machine. When prompted, type **yes** to continue and enter the **password** you have set for the virtual machine in the previous step.

```
Striim — striimdemo@striimdemo-masternode:~ — ssh striimdemo@striimdemo.westus.cloud
Striim-Edwards-MacBook-Pro:Striim edwardbell$ ssh striimdemo@striimdemo.westus.cloudapp.azure.com
Password:
Last login: Tue Jul 16 21:04:40 2019 from 208.89.164.26
[striimdemo@striimdemo-masternode ~]$
```

4. Now, open a new terminal tab to copy the **ojdbc8.jar** file you downloaded previously. Use the following SCP command to copy the jar file from your local machine to the tmp folder of the Striim instance running in Azure:

```
cd <Directory_path_where_the_Jar_file_exists>
scp ojdbc8.jar striimdemo@striimdemo.westus.cloudapp.azure.com:/tmp
```

```
Jars — ssh ◂ scp ojdbc8.jar striimdemo@striimdemo.westus.cloudapp.azure.com:/tmp — 134×27
...e:~ — ssh striimdemo@striimdemo.westus.cloudapp.azure.com ...striimdemo@striimdemo.westus.cloudapp.azure.com:/tmp
Last login: Tue Jul 16 14:03:40 on ttys001
Striim-Edwards-MacBook-Pro:Striim edwardbell$ cd /Users/edwardbell/Desktop/Jars
Striim-Edwards-MacBook-Pro:Jars edwardbell$ scp ojdbc8.jar striimdemo@striimdemo.westus.cloudapp.azure.com:/tmp
Password:
ojdbc8.jar                                                    82% 3360KB   2.3MB/s   00:00 ETA
```

5. Next, navigate back to the window where you did SSH to the Striim instance and Login as sudo. Move the **ojdbc8.jar** file from the **/tmp** directory into the **lib** directory of your Striim instance with the following commands:

```
sudo su
cd /tmp
mv ojdbc8.jar /opt/striim/lib
chmod +x ojdbc8.jar
```

```
Striim — root@striimdemo-masternode:/opt/striim/lib — ssh striimdemo@striimdemo.westus.cloudapp.azure.com — 13...
.../lib — ssh striimdemo@striimdemo.westus.cloudapp.azure.com ~/Desktop/Jars — -bash
Striim-Edwards-MacBook-Pro:Striim edwardbell$ ssh striimdemo@striimdemo.westus.cloudapp.azure.com
Password:
Last login: Tue Jul 16 21:05:02 2019 from 208.89.164.26
[striimdemo@striimdemo-masternode ~]$ sudo su
[sudo] password for striimdemo:
[root@striimdemo-masternode striimdemo]# cd /tmp
[root@striimdemo-masternode tmp]# ls
hsperfdata_root  hsperfdata_striim  jna--891986052  ojdbc8.jar
[root@striimdemo-masternode tmp]# mv ojdbc8.jar /opt/striim/lib
[root@striimdemo-masternode tmp]# cd /opt/striim/lib
[root@striimdemo-masternode lib]# chmod +x ojdbc8.jar
```

6. From the same terminal window, restart the Striim server by executing the following commands:

```
Systemctl stop striim-node
Systemctl stop striim-dbms
Systemctl start striim-dbms
Systemctl start striim-node
```

7. Striim will take a minute to start up. If you'd like to see the status, run the following command:

```
tail -f /opt/striim/logs/striim-node.log
```

8. Now, navigate back to Azure and copy the Public IP address of your Striim VM.

9. To navigate to the Striim's Web UI, open a new tab in a browser and copy the public IP followed by: 9080. Sign in by using the **admin** username, along with the admin password you specified in the Azure portal.

10. Now you'll arrive at Striim's home page. There are three different panes – **Dashboards**, **Apps**, and **SourcePreview**. The Dashboards pane allows you to move data in real time and visualize it. The Apps pane contains your streaming data pipelines, or data flows. On the right hand of the page is SourcePreview where you can preview your data before moving it.

11. Select the **Apps** pane, we'll focus on this pane for now. There are a variety of sample apps that you can use to learn about Striim, however in this article you will create our own. Select the **Add App** button in the top

right-hand corner.

12. There are a few different ways to create Striim applications. Select **Start from Scratch** for this scenario.

13. Give a friendly name for your application, something like **oraToCosmosDB** and select **Save**.

14. You'll arrive at the Flow Designer, where you can drag and drop out of the box connectors to create your streaming applications. Type **Oracle** in the search bar, drag and drop the **Oracle CDC** source onto the app canvas.

15. Enter the source configuration properties of your Oracle instance. The source name is just a naming convention for the Striim application, you can use a name such as **src_onPremOracle**. Also enter other details like Adapter type, connection URL, username, password, table name. Select **Save** to continue.

16. Now, click the wave icon of the stream to connect the target Azure Cosmos DB instance.

17. Before configuring the target, make sure you have added a Baltimore root certificate to Striim's Java environment.

18. Enter the configuration properties of your target Azure Cosmos DB instance and select **Save** to continue. Here are the key parameters to note:

- **Adapter** - Use **DatabaseWriter**. When writing to Azure Cosmos DB Cassandra API, DatabaseWriter is required. The Cassandra driver 3.6.0 is bundled with Striim. If the DatabaseWriter exceeds the number of RUs provisioned on your Azure Cosmos container, the application will crash.
- **Connection URL** - Specify your Azure Cosmos DB JDBC connection URL. The URL is in the format `jdbc:cassandra://<contactpoint>:10350/<databaseName>?SSL=true`
- **Username** - Specify your Azure Cosmos account name.
- **Password** - Specify the primary key of your Azure Cosmos account.
- **Tables** - Target tables must have primary keys and primary keys can not be updated.

19. Now, we'll go ahead and run the Striim application. In the upper menu bar select **Created**, and then **Deploy App**. In the deployment window you can specify if you want to run certain parts of your application on specific parts of your deployment topology. Since we're running in a simple deployment topology through

Azure, we'll use the default option.

20. Now, we'll go ahead and preview the stream to see data flowing through the Striim. Click the wave icon and click the eye icon next to it. After deploying, you can preview the stream to see data flowing through. Select the **wave** icon and the **eyeball** next to it. Select the **Deployed** button in the top menu bar, and select **Start App**.

21. By using a **CDC(Change Data Capture)** reader, Striim will pick up only new changes on the database. If you have data flowing through your source tables, you'll see it. However, since this is a sample table, the source that isn't connected to any application. If you use a sample data generator, you can insert a chain of events into your Oracle database.

22. You'll see data flowing through the Striim platform. Striim picks up all the metadata associated with your table as well, which is helpful to monitor the data and make sure that the data lands on the right target.

23. Finally, let's sign into Azure and navigate to your Azure Cosmos account. Refresh the Data Explorer, and you can see that data has arrived.

By using the Striim solution in Azure, you can continuously migrate data to Azure Cosmos DB from various sources such as Oracle, Cassandra, MongoDB, and various others to Azure Cosmos DB. To learn more please visit the Striim website, download a free 30-day trial of Striim, and for any issues when setting up the migration path with Striim, file a support request.

Next steps

- If you are migrating data to Azure Cosmos DB SQL API, see how to migrate data to Cassandra API account using Striim
- Monitor and debug your data with Azure Cosmos DB metrics

Migrate data from Oracle to Azure Cosmos DB Cassandra API account using Blitzz

12/4/2020 • 5 minutes to read • Edit Online

APPLIES TO: Cassandra API

Cassandra API in Azure Cosmos DB has become a great choice for enterprise workloads that are running on Oracle for a variety of reasons such as:

- **Better scalability and availability:** It eliminates single points of failure, better scalability, and availability for your applications.
- **Significant cost savings:** You can save cost with Azure Cosmos DB, which includes the cost of VM's, bandwidth, and any applicable Oracle licenses. Additionally, you don't have to manage the data centers, servers, SSD storage, networking, and electricity costs.
- **No overhead of managing and monitoring:** As a fully managed cloud service, Azure Cosmos DB removes the overhead of managing and monitoring a myriad of settings.

There are various ways to migrate database workloads from one platform to another. Blitzz is a tool that offers a secure and reliable way to perform zero downtime migration from a variety of databases to Azure Cosmos DB. This article describes the steps required to migrate data from Oracle database to Azure Cosmos DB Cassandra API using Blitzz.

Benefits using Blitzz for migration

Blitzz's migration solution follows a step by step approach to migrate complex operational workloads. The following are some of the key aspects of Blitzz's zero-downtime migration plan:

- It offers automatic migration of business logic (tables, indexes, views) from Oracle database to Azure Cosmos DB. You don't have to create schemas manually.
- Blitzz offers high-volume and parallel database replication. It enables both the source and target platforms to be in-sync during the migration by using a technique called Change-Data-Capture (CDC). By using CDC, Blitzz continuously pulls a stream of changes from the source database(Oracle) and applies it to the destination database(Azure Cosmos DB).
- It is fault-tolerant and guarantees exactly once delivery of data even during a hardware or software failure in the system.
- It secures the data during transit using a variety of security methodologies like TLS/SSL, encryption.
- It offers services to convert complex business logic written in PL/SQL to equivalent business logic in Azure Cosmos DB.

Steps to migrate data

This section describes the steps required to setup Blitzz and migrates data from Oracle database to Azure Cosmos DB.

1. From the computer where you plan to install the Blitzz replicant, add a security certificate. This certificate is required by the Blitzz replicant to establish a TLS connection with the specified Azure Cosmos DB account. You can add the certificate with the following steps:

```
wget https://cacert.omniroot.com/bc2025.crt
mv bc2025.crt bc2025.cer
keytool -keystore $JAVA_HOME/lib/security/cacerts -importcert -alias bc2025ca -file bc2025.cer
```

2. ou can get the Blitzz installation and the binary files either by requesting a demo on the Blitzz website. Alternatively, you can also send an email to the team.

3. From the CLI terminal, set up the source database configuration. Open the configuration file using `vi conf/conn/oracle.yml` command and add a comma-separated list of IP addresses of the oracle nodes, port number, username, password, and any other required details. The following code shows an example configuration file:

```
type: ORACLE

host: localhost
port: 53546

service-name: IO

username: '<Username of your Oracle database>'
password: '<Password of your Oracle database>'

conn-cnt: 30
use-ssl: false
```

After filling out the configuration details, save and close the file.

4. Optionally, you can set up the source database filter file. The filter file specifies which schemas or tables to migrate. Open the configuration file using `vi filter/oracle_filter.yml` command and enter the following configuration details:

```
allow:
- schema: “io_blitzz”
Types: [TABLE]
```

After filling out the database filter details, save and close the file.

5. Next you will set up the configuration of the destination database. Before you define the configuration, create an Azure Cosmos DB Cassandra API account. Choose the right partition key from your data and then create a Keyspace, and a table to store the migrated data.

6. Before migrating the data, increase the container throughput to the amount required for your application to migrate quickly. For example, you can increase the throughput to 100000 RUs. Scaling the throughput before starting the migration will help you to migrate your data in less time.

You must decrease the throughput after the migration is complete. Based on the amount of data stored and RUs required for each operation, you can estimate the throughput required after data migration. To learn more on how to estimate the RUs required, see Provision throughput on containers and databases and Estimate RU/s using the Azure Cosmos DB capacity planner articles.

7. Get the **Contact Point, Port, Username**, and **Primary Password** of your Azure Cosmos account from the **Connection String** pane. You will use these values in the configuration file.

8. From the CLI terminal, set up the destination database configuration. Open the configuration file using `vi conf/conn/cosmosdb.yml` command and add a comma-separated list of host URI, port number, username, password, and other required parameters. The following is an example of contents in the configuration file:

```
type: COSMOSDB

host: `<Azure Cosmos account's Contact point>`
port: 10350

username: 'blitzzdemo'
password: `<Your Azure Cosmos account's primary password>'

max-connections: 30
use-ssl: false
```

9. Next migrate the data using Blitzz. You can run the Blizz replicant in **full** or **snapshot** mode:

   - **Full mode** – In this mode, the replicant continues to run after migration and it listens for any changes on the source Oracle system. If it detects any changes, they are replicated on the target Azure Cosmos account in real time.

   - **Snapshot mode** – In this mode, you can perform schema migration and one-time data replication. Real-time replication isn't supported with this option.

   By using the above two modes, migration can be performed with zero downtime.

10. To migrate data, from the Blitzz replicant CLI terminal, run the following command:

```
./bin/replicant full conf/conn/oracle.yaml conf/conn/cosmosdb.yaml --filter filter/oracle_filter.yaml --
replace-existing
```

The replicant UI shows the replication progress. Once the schema migration and snapshot operation are done, the progress shows 100%. After the migration is complete, you can validate the data on the target Azure Cosmos database.

11. Because you have used full mode for migration, you can perform operations such as insert, update, or delete data on the source Oracle database. Later you can validate that they are replicated real time on the target Azure Cosmos database. After the migration, make sure to decrease the throughput configured for your Azure Cosmos container.

12. You can stop the replicant any point and restart it with **--resume** switch. The replication resumes from the point it has stopped without compromising on data consistency. The following command shows how to use the resume switch.

```
./bin/replicant full conf/conn/oracle.yaml conf/conn/cosmosdb.yaml --filter filter/oracle_filter.yaml --
replace-existing --resume
```

To learn more on the data migration to destination, real-time migration, see the Blitzz replicant demo.

Next steps

- Provision throughput on containers and databases
- Partition key best practices
- Estimate RU/s using the Azure Cosmos DB capacity planner articles

Migrate data from Cassandra to Azure Cosmos DB Cassandra API account using Blitzz

12/4/2020 • 5 minutes to read • Edit Online

APPLIES TO: Cassandra API

Cassandra API in Azure Cosmos DB has become a great choice for enterprise workloads running on Apache Cassandra for a variety of reasons such as:

- **No overhead of managing and monitoring:** It eliminates the overhead of managing and monitoring a myriad of settings across OS, JVM, and yaml files and their interactions.
- **Significant cost savings:** You can save cost with Azure Cosmos DB, which includes the cost of VM's, bandwidth, and any applicable licenses. Additionally, you don't have to manage the data centers, servers, SSD storage, networking, and electricity costs.
- **Ability to use existing code and tools:** Azure Cosmos DB provides wire protocol level compatibility with existing Cassandra SDKs and tools. This compatibility ensures you can use your existing codebase with Azure Cosmos DB Cassandra API with trivial changes.

There are various ways to migrate database workloads from one platform to another. Blitzz is a tool that offers a secure and reliable way to perform zero downtime migration from a variety of databases to Azure Cosmos DB. This article describes the steps required to migrate data from Apache Cassandra database to Azure Cosmos DB Cassandra API using Blitzz.

Benefits using Blitzz for migration

Blitzz's migration solution follows a step by step approach to migrate complex operational workloads. The following are some of the key aspects of Blitzz's zero-downtime migration plan:

- It offers automatic migration of business logic (tables, indexes, views) from Apache Cassandra database to Azure Cosmos DB. You don't have to create schemas manually.
- Blitzz offers high-volume and parallel database replication. It enables both the source and target platforms to be in-sync during the migration by using a technique called Change-Data-Capture (CDC). By using CDC, Blitzz continuously pulls a stream of changes from the source database(Apache Cassandra) and applies it to the destination database(Azure Cosmos DB).
- It is fault-tolerant and provides exactly once delivery of data even during a hardware or software failure in the system.
- It secures the data during transit using a variety of security methodologies like TLS, encryption.

Steps to migrate data

This section describes the steps required to set up Blitzz and migrates data from Apache Cassandra database to Azure Cosmos DB.

1. From the computer where you plan to install the Blitzz replicant, add a security certificate. This certificate is required by the Blitzz replicant to establish a TLS connection with the specified Azure Cosmos DB account. You can add the certificate with the following steps:

```
wget https://cacert.omniroot.com/bc2025.crt
mv bc2025.crt bc2025.cer
keytool -keystore $JAVA_HOME/lib/security/cacerts -importcert -alias bc2025ca -file bc2025.cer
```

2. You can get the Blitzz installation and the binary files either by requesting a demo on the Blitzz website. Alternatively, you can also send an email to the team.

3. From the CLI terminal, set up the source database configuration. Open the configuration file using `vi conf/conn/cassandra.yml` command and add a comma-separated list of IP addresses of the Cassandra nodes, port number, username, password, and any other required details. The following is an example of contents in the configuration file:

```
type: CASSANDRA

host: 172.17.0.2
port: 9042

username: 'cassandra'
password: 'cassandra'

max-connections: 30
```

After filling out the configuration details, save and close the file.

4. Optionally, you can set up the source database filter file. The filter file specifies which schemas or tables to migrate. Open the configuration file using `vi filter/cassandra_filter.yml` command and enter the following configuration details:

```
allow:
- schema: “io_blitzz”
Types: [TABLE]
```

After filling out the database filter details, save and close the file.

5. Next you will set up the destination database configuration. Before you define the configuration, create an Azure Cosmos DB Cassandra API account and then create a Keyspace, and a table to store the migrated data. Because you are migrating from Apache Cassandra to Cassandra API in Azure Cosmos DB, you can use the same partition key that you have used with Apache cassandra.

6. Before migrating the data, increase the container throughput to the amount required for your application to migrate quickly. For example, you can increase the throughput to 100000 RUs. Scaling the throughput before starting the migration will help you to migrate your data in less time.

Decrease the throughput after the migration is complete. Based on the amount of data stored and RUs required for each operation, you can estimate the throughput required after data migration. To learn more on how to estimate the RUs required, see Provision throughput on containers and databases and Estimate RU/s using the Azure Cosmos DB capacity planner articles.

7. Get the **Contact Point, Port, Username**, and **Primary Password** of your Azure Cosmos account from the **Connection String** pane. You will use these values in the configuration file.

8. From the CLI terminal, set up the destination database configuration. Open the configuration file using `vi conf/conn/cosmosdb.yml` command and add a comma-separated list of host URI, port number, username, password, and other required parameters. The following example shows the contents of the configuration file:

```
type: COSMOSDB

host: '<Azure Cosmos account's Contact point>'
port: 10350

username: 'blitzzdemo'
password: '<Your Azure Cosmos account's primary password>'

max-connections: 30
```

9. Next migrate the data using Blitzz. You can run the Blizz replicant in **full** or **snapshot** mode:

   - **Full mode** – In this mode, the replicant continues to run after migration and it listens for any changes on the source Apache Cassandra system. If it detects any changes, they are replicated on the target Azure Cosmos account in real time.

   - **Snapshot mode** – In this mode, you can perform schema migration and one-time data replication. Real-time replication isn't supported with this option.

   By using the above two modes, migration can be performed with zero downtime.

10. To migrate data, from the Blitzz replicant CLI terminal, run the following command:

```
./bin/replicant full conf/conn/cassandra.yaml conf/conn/cosmosdb.yaml --filter
filter/cassandra_filter.yaml --replace-existing
```

The replicant UI shows the replication progress. Once the schema migration and snapshot operation are done, the progress shows 100%. After the migration is complete, you can validate the data on the target Azure Cosmos database.

```
nileshgohad@nileshgohad-ThinkPad-T480s<Blitzz>
File  Edit  View  Search  Terminal  Help

                          Cassandra   -->   CosmosDB

Progress: ====================================100.00%====================================

    Elapsed time: 00:03:22     Initial load time: 00:02:45           Rate: 0.00

Table name      Rows        Size      Inserted    Deleted     Updated       Rate

partsupp           8000   409816000        8000          0           0    0.00
orders            15000  3840405000       15000          0           0    0.00
lineitem          60175 24652253300       60175          0           0    0.00
nation               25     2560200          25          0           0    0.00
region                5      512020           5          0           0    0.00
customer           1500   384034500        1500          0           0    0.00
part               2000   614446000        2000          0           0    0.00
supplier            100    20482300         100          0           0    0.00
```

11. Because you have used full mode for migration, you can perform operations such as insert, update, or delete data on the source Apache Cassandra database. Later validate that they are replicated real time on the target Azure Cosmos database. After the migration, make sure to decrease the throughput configured for your Azure Cosmos container.

12. You can stop the replicant any point and restart it with **--resume** switch. The replication resumes from the point it has stopped without compromising on data consistency. The following command shows how to use the resume switch.

```
./bin/replicant full conf/conn/cassandra.yaml conf/conn/cosmosdb.yaml --filter
filter/cassandra_filter.yaml --replace-existing --resume
```

To learn more on the data migration to destination, real-time migration, see the Blitzz replicant demo.

Next steps

- Provision throughput on containers and databases
- Partition key best practices
- Estimate RU/s using the Azure Cosmos DB capacity planner articles

Migrate data from Cassandra to Azure Cosmos DB Cassandra API account using Azure Databricks

12/4/2020 • 3 minutes to read • Edit Online

APPLIES TO: Cassandra API

Cassandra API in Azure Cosmos DB has become a great choice for enterprise workloads running on Apache Cassandra for a variety of reasons such as:

- **No overhead of managing and monitoring:** It eliminates the overhead of managing and monitoring a myriad of settings across OS, JVM, and yaml files and their interactions.
- **Significant cost savings:** You can save cost with Azure Cosmos DB, which includes the cost of VM's, bandwidth, and any applicable licenses. Additionally, you don't have to manage the data centers, servers, SSD storage, networking, and electricity costs.
- **Ability to use existing code and tools:** Azure Cosmos DB provides wire protocol level compatibility with existing Cassandra SDKs and tools. This compatibility ensures you can use your existing codebase with Azure Cosmos DB Cassandra API with trivial changes.

There are various ways to migrate database workloads from one platform to another. Azure Databricks is a platform as a service offering for Apache Spark that offers a way to perform offline migrations at large scale. This article describes the steps required to migrate data from native Apache Cassandra keyspaces/tables to Azure Cosmos DB Cassandra API using Azure Databricks.

Prerequisites

- Provision an Azure Cosmos DB Cassandra API account
- Review the basics of connecting to Azure Cosmos DB Cassandra API
- Review the supported features in Azure Cosmos DB Cassandra API to ensure compatibility.
- Ensure you have already created empty keyspace and tables in your target Azure Cosmos DB Cassandra API account
- Use cqlsh or hosted shell for validation

Provision an Azure Databricks cluster

You can follow instructions to Provision an Azure Databricks cluster. However, please note Apache Spark 3.x is not currently supported for the Apache Cassandra Connector. You will need to provision a Databricks runtime with a supported v2.x version of Apache Spark. We recommend version 6.6 of Databricks runtime:

Add dependencies

You will need to add the Apache Spark Cassandra connector library to your cluster in order to connect to both native and Cosmos DB Cassandra endpoints. In your cluster select libraries -> install new -> maven -> search packages:

Type `Cassandra` in the search box, and select the latest `spark-cassandra-connector` maven repository available, then select install:

NOTE

Ensure that you restart the Databricks cluster after the Cassandra Connector library has been installed.

Create Scala Notebook for migration

Create a Scala Notebook in Databricks with the following code. Replace your source and target cassandra configurations with corresponding credentials, and source/target keyspaces and tables, then run:

```
import com.datastax.spark.connector._
import com.datastax.spark.connector.cql._
import org.apache.spark.SparkContext

// source cassandra configs
val nativeCassandra = Map(
    "spark.cassandra.connection.host" -> "<Source Cassandra Host>",
    "spark.cassandra.connection.port" -> "9042",
    "spark.cassandra.auth.username" -> "<USERNAME>",
    "spark.cassandra.auth.password" -> "<PASSWORD>",
    "spark.cassandra.connection.ssl.enabled" -> "false",
    "keyspace" -> "<KEYSPACE>",
    "table" -> "<TABLE>"
)

//target cassandra configs
val cosmosCassandra = Map(
    "spark.cassandra.connection.host" -> "<USERNAME>.cassandra.cosmos.azure.com",
    "spark.cassandra.connection.port" -> "10350",
    "spark.cassandra.auth.username" -> "<USERNAME>",
    "spark.cassandra.auth.password" -> "<PASSWORD>",
    "spark.cassandra.connection.ssl.enabled" -> "true",
    "keyspace" -> "<KEYSPACE>",
    "table" -> "<TABLE>",
    //throughput related settings below - tweak these depending on data volumes.
    "spark.cassandra.output.batch.size.rows"-> "1",
    "spark.cassandra.connection.connections_per_executor_max" -> "10",
    "spark.cassandra.output.concurrent.writes" -> "1000",
    "spark.cassandra.concurrent.reads" -> "512",
    "spark.cassandra.output.batch.grouping.buffer.size" -> "1000",
    "spark.cassandra.connection.keep_alive_ms" -> "600000000"
)

//Read from native Cassandra
val DFfromNativeCassandra = sqlContext
  .read
  .format("org.apache.spark.sql.cassandra")
  .options(nativeCassandra)
  .load

//Write to CosmosCassandra
DFfromNativeCassandra
  .write
  .format("org.apache.spark.sql.cassandra")
  .options(cosmosCassandra)
  .save
```

NOTE

The `spark.cassandra.output.concurrent.writes` and `connections_per_executor_max` configurations are important for avoiding rate limiting, which happens when requests to Cosmos DB exceed provisioned throughput (request units). You may need to adjust these settings depending on the number of executors in the Spark cluster, and potentially the size (and therefore RU cost) of each record being written to the target tables.

Next steps

- Provision throughput on containers and databases
- Partition key best practices
- Estimate RU/s using the Azure Cosmos DB capacity planner articles
- Elastic Scale in Azure Cosmos DB Cassandra API

Change feed in the Azure Cosmos DB API for Cassandra

11/2/2020 • 3 minutes to read • Edit Online

APPLIES TO: Cassandra API

Change feed support in the Azure Cosmos DB API for Cassandra is available through the query predicates in the Cassandra Query Language (CQL). Using these predicate conditions, you can query the change feed API. Applications can get the changes made to a table using the primary key (also known as the partition key) as is required in CQL. You can then take further actions based on the results. Changes to the rows in the table are captured in the order of their modification time and the sort order per partition key.

The following example shows how to get a change feed on all the rows in a Cassandra API Keyspace table using .NET. The predicate COSMOS_CHANGEFEED_START_TIME() is used directly within CQL to query items in the change feed from a specified start time (in this case current datetime). You can download the full sample, for C# here and for Java here.

In each iteration, the query resumes at the last point changes were read, using paging state. We can see a continuous stream of new changes to the table in the Keyspace. We will see changes to rows that are inserted, or updated. Watching for delete operations using change feed in Cassandra API is currently not supported.

- Java
- C#

```
        Session cassandraSession = utils.getSession();

        try {
           DateTimeFormatter dtf = DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss");
            LocalDateTime now = LocalDateTime.now().minusHours(6).minusMinutes(30);
            String query="SELECT * FROM uprofile.user where COSMOS_CHANGEFEED_START_TIME()='"
              + dtf.format(now)+ "'";

          byte[] token=null;
          System.out.println(query);
          while(true)
          {
           SimpleStatement st=new  SimpleStatement(query);
           st.setFetchSize(100);
           if(token!=null)
            st.setPagingStateUnsafe(token);

           ResultSet result=cassandraSession.execute(st) ;
           token=result.getExecutionInfo().getPagingState().toBytes();

           for(Row row:result)
           {
            System.out.println(row.getString("user_name"));
           }
          }


        } finally {
            utils.close();
            LOGGER.info("Please delete your table after verifying the presence of the data in portal or from
CQL");
        }
```

In order to get the changes to a single row by primary key, you can add the primary key in the query. The following example shows how to track changes for the row where "user_id = 1"

- C#
- Java

```
    String query="SELECT * FROM uprofile.user where user_id=1 and COSMOS_CHANGEFEED_START_TIME()='"
              + dtf.format(now)+ "'";
    SimpleStatement st=new  SimpleStatement(query);
```

Current limitations

The following limitations are applicable when using change feed with Cassandra API:

- Inserts and updates are currently supported. Delete operation is not yet supported. As a workaround, you can add a soft marker on rows that are being deleted. For example, add a field in the row called "deleted" and set it to "true".
- Last update is persisted as in core SQL API and intermediate updates to the entity are not available.

Error handling

The following error codes and messages are supported when using change feed in Cassandra API:

- **HTTP error code 429** - When the change feed is rate limited, it returns an empty page.

Next steps

- Manage Azure Cosmos DB Cassandra API resources using Azure Resource Manager templates

Create a container in Azure Cosmos DB Cassandra API

11/2/2020 • 2 minutes to read • Edit Online

APPLIES TO: Cassandra API

This article explains the different ways to create a container in Azure Cosmos DB Cassandra API. It shows how to create a container using Azure portal, Azure CLI, PowerShell, or supported SDKs. This article demonstrates how to create a container, specify the partition key, and provision throughput.

This article explains the different ways to create a container in Azure Cosmos DB Cassandra API. If you are using a different API, see API for MongoDB, Gremlin API, Table API, and SQL API articles to create the container.

> **NOTE**
> When creating containers, make sure you don't create two containers with the same name but different casing. That's because some parts of the Azure platform are not case-sensitive, and this can result in confusion/collision of telemetry and actions on containers with such names.

Create using Azure portal

1. Sign in to the Azure portal.
2. Create a new Azure Cosmos account, or selectan existing account.
3. Open the **Data Explorer** pane, and select **New Table**. Next, provide the following details:
   - Indicate whether you are creating a new keyspace, or using an existing one.
   - Enter a table name.
   - Enter the properties and specify a primary key.
   - Enter a throughput to be provisioned (for example, 1000 RUs).
   - Select **OK**.

> **NOTE**
>
> For Cassandra API, the primary key is used as the partition key.

Create using .NET SDK

```
// Create a Cassandra table with a partition/primary key and provision 1000 RU/s throughput.
session.Execute(CREATE TABLE myKeySpace.myTable(
    user_id int PRIMARY KEY,
    firstName text,
    lastName text) WITH cosmosdb_provisioned_throughput=1000);
```

If you encounter timeout exception when creating a collection, do a read operation to validate if the collection was created successfully. The read operation throws an exception until the collection create operation is successful. For the list of status codes supported by the create operation see the HTTP Status Codes for Azure Cosmos DB article.

Create using Azure CLI

Create a Cassandra table with Azure CLI. For a listing of all Azure CLI samples across all Azure Cosmos DB APIs see, Azure CLI samples for Azure Cosmos DB.

Create using PowerShell

Create a Cassandra table with PowerShell. For a listing of all PowerShell samples across all Azure Cosmos DB APIs see, PowerShell Samples

Next steps

- Partitioning in Azure Cosmos DB
- Request Units in Azure Cosmos DB
- Provision throughput on containers and databases
- Work with Azure Cosmos account

Provision database, container or autoscale throughput on Azure Cosmos DB Cassandra API resources

11/2/2020 • 2 minutes to read • Edit Online

APPLIES TO: Cassandra API

This article explains how to provision throughput in Azure Cosmos DB Cassandra API. You can provision standard(manual) or autoscale throughput on a container, or a database and share it among the containers within the database. You can provision throughput using Azure portal, Azure CLI, or Azure Cosmos DB SDKs.

If you are using a different API, see SQL API, API for MongoDB, Gremlin API articles to provision the throughput.

Azure portal

1. Sign in to the Azure portal.
2. Create a new Azure Cosmos account, or select an existing Azure Cosmos account.
3. Open the **Data Explorer** pane, and select **New Table**. Next, provide the following details:
   - Indicate whether you are creating a new keyspace or using an existing one. Select the **Provision database throughput** option if you want to provision throughput at the keyspace level.
   - Enter the table ID within the CQL command.
   - Enter a primary key value (for example, `/userrID`).
   - Enter a throughput that you want to provision (for example, 1000 RUs).
   - Select **OK**.

> **NOTE**
> If you are provisioning throughput on a container in an Azure Cosmos account configured with Cassandra API, use `/myPrimaryKey` for the partition key path.

.NET SDK

Provision throughput for a Cassandra table

```
// Create a Cassandra table with a partition (primary) key and provision throughput of 400 RU/s
session.Execute("CREATE TABLE myKeySpace.myTable(
    user_id int PRIMARY KEY,
    firstName text,
    lastName text) WITH cosmosdb_provisioned_throughput=400");
```

Similar commands can be issued through any CQL-compliant driver.

Alter or change throughput for a Cassandra table

```
// Altering the throughput too can be done through code by issuing following command
session.Execute("ALTER TABLE myKeySpace.myTable WITH cosmosdb_provisioned_throughput=5000");
```

Similar command can be executed through any CQL compliant driver.

```
// Create a Cassandra keyspace and provision throughput of 400 RU/s
session.Execute("CREATE KEYSPACE IF NOT EXISTS myKeySpace WITH cosmosdb_provisioned_throughput=400");
```

Azure Resource Manager

Azure Resource Manager templates can be used to provision autoscale throughput on database or container-level resources for all Azure Cosmos DB APIs. See Azure Resource Manager templates for Azure Cosmos DB for samples.

Azure CLI

Azure CLI can be used to provision autoscale throughput on a database or container-level resources for all Azure Cosmos DB APIs. For samples see Azure CLI Samples for Azure Cosmos DB.

Azure PowerShell

Azure PowerShell can be used to provision autoscale throughput on a database or container-level resources for all Azure Cosmos DB APIs. For samples see Azure PowerShell samples for Azure Cosmos DB.

Next steps

See the following articles to learn about throughput provisioning in Azure Cosmos DB:

- Request units and throughput in Azure Cosmos DB

Find the request unit charge for operations executed in Azure Cosmos DB Cassandra API

11/2/2020 • 2 minutes to read • Edit Online

APPLIES TO: Cassandra API

Azure Cosmos DB supports many APIs, such as SQL, MongoDB, Cassandra, Gremlin, and Table. Each API has its own set of database operations. These operations range from simple point reads and writes to complex queries. Each database operation consumes system resources based on the complexity of the operation.

The cost of all database operations is normalized by Azure Cosmos DB and is expressed by Request Units (or RUs, for short). You can think of RUs as a performance currency abstracting the system resources such as CPU, IOPS, and memory that are required to perform the database operations supported by Azure Cosmos DB. No matter which API you use to interact with your Azure Cosmos container, costs are always measured by RUs. Whether the database operation is a write, point read, or query, costs are always measured in RUs. To learn more, see the request units and it's considerations article.

This article presents the different ways you can find the request unit (RU) consumption for any operation executed against a container in Azure Cosmos DB Cassandra API. If you are using a different API, see API for MongoDB, SQL API, Gremlin API, and Table API articles to find the RU/s charge.

When you perform operations against the Azure Cosmos DB Cassandra API, the RU charge is returned in the incoming payload as a field named `RequestCharge`. You have multiple options for retrieving the RU charge.

Use the .NET SDK

When you use the .NET SDK, you can retrieve the incoming payload under the `Info` property of a `RowSet` object:

```
RowSet rowSet = session.Execute("SELECT table_name FROM system_schema.tables;");
double requestCharge = BitConverter.ToDouble(rowSet.Info.IncomingPayload["RequestCharge"].Reverse().ToArray(),
0);
```

For more information, see Quickstart: Build a Cassandra app by using the .NET SDK and Azure Cosmos DB.

Use the Java SDK

When you use the Java SDK, you can retrieve the incoming payload by calling the `getExecutionInfo()` method on a `ResultSet` object:

```
ResultSet resultSet = session.execute("SELECT table_name FROM system_schema.tables;");
Double requestCharge = resultSet.getExecutionInfo().getIncomingPayload().get("RequestCharge").getDouble();
```

For more information, see Quickstart: Build a Cassandra app by using the Java SDK and Azure Cosmos DB.

Next steps

To learn about optimizing your RU consumption, see these articles:

- Request units and throughput in Azure Cosmos DB
- Optimize provisioned throughput cost in Azure Cosmos DB

- Optimize query cost in Azure Cosmos DB

Connect to Azure Cosmos DB Cassandra API from Spark

11/2/2020 • 4 minutes to read • Edit Online

APPLIES TO: Cassandra API

This article is one among a series of articles on Azure Cosmos DB Cassandra API integration from Spark. The articles cover connectivity, Data Definition Language(DDL) operations, basic Data Manipulation Language(DML) operations, and advanced Azure Cosmos DB Cassandra API integration from Spark.

Prerequisites

- Provision an Azure Cosmos DB Cassandra API account.
- Provision your choice of Spark environment [Azure Databricks | Azure HDInsight-Spark | Others].

Dependencies for connectivity

- **Spark connector for Cassandra:** Spark connector is used to connect to Azure Cosmos DB Cassandra API. Identify and use the version of the connector located in Maven central that is compatible with the Spark and Scala versions of your Spark environment.
- **Azure Cosmos DB helper library for Cassandra API:** In addition to the Spark connector, you need another library called azure-cosmos-cassandra-spark-helper from Azure Cosmos DB. This library contains custom connection factory and retry policy classes.

  The retry policy in Azure Cosmos DB is configured to handle HTTP status code 429("Request Rate Large") exceptions. The Azure Cosmos DB Cassandra API translates these exceptions into overloaded errors on the Cassandra native protocol, and you can retry with back-offs. Because Azure Cosmos DB uses provisioned throughput model, request rate limiting exceptions occur when the ingress/egress rates increase. The retry policy protects your spark jobs against data spikes that momentarily exceed the throughput allocated for your container.

  > **NOTE**
  > The retry policy can protect your spark jobs against momentary spikes only. If you have not configured enough RUs required to run your workload, then the retry policy is not applicable and the retry policy class rethrows the exception.

- **Azure Cosmos DB account connection details:** Your Azure Cassandra API account name, account endpoint, and key.

Spark connector throughput configuration parameters

The following table lists Azure Cosmos DB Cassandra API-specific throughput configuration parameters provided by the connector. For a detailed list of all configuration parameters, see configuration reference page of the Spark Cassandra Connector GitHub repository.

PROPERTY NAME	DEFAULT VALUE	DESCRIPTION
spark.cassandra.output.batch.size.rows	1	Number of rows per single batch. Set this parameter to 1. This parameter is used to achieve higher throughput for heavy workloads.
spark.cassandra.connection.connections_per_executor_max	None	Maximum number of connections per node per executor. 10*n is equivalent to 10 connections per node in an n-node Cassandra cluster. So, if you require 5 connections per node per executor for a 5 node Cassandra cluster, then you should set this configuration to 25. Modify this value based on the degree of parallelism or the number of executors that your spark jobs are configured for.
spark.cassandra.output.concurrent.writes	100	Defines the number of parallel writes that can occur per executor. Because you set "batch.size.rows" to 1, make sure to scale up this value accordingly. Modify this value based on the degree of parallelism or the throughput that you want to achieve for your workload.
spark.cassandra.concurrent.reads	512	Defines the number of parallel reads that can occur per executor. Modify this value based on the degree of parallelism or the throughput that you want to achieve for your workload
spark.cassandra.output.throughput_mb_per_sec	None	Defines the total write throughput per executor. This parameter can be used as an upper limit for your spark job throughput, and base it on the provisioned throughput of your Cosmos container.
spark.cassandra.input.reads_per_sec	None	Defines the total read throughput per executor. This parameter can be used as an upper limit for your spark job throughput, and base it on the provisioned throughput of your Cosmos container.
spark.cassandra.output.batch.grouping.buffer.size	1000	Defines the number of batches per single spark task that can be stored in memory before sending to Cassandra API
spark.cassandra.connection.keep_alive_ms	60000	Defines the period of time until which unused connections are available.

Adjust the throughput and degree of parallelism of these parameters based on the workload you expect for your spark jobs, and the throughput you have provisioned for your Cosmos DB account.

Connecting to Azure Cosmos DB Cassandra API from Spark

cqlsh

The following commands detail how to connect to Azure CosmosDB Cassandra API from cqlsh. This is useful for validation as you run through the samples in Spark.

From Linux/Unix/Mac:

```
export SSL_VERSION=TLSv1_2
export SSL_VALIDATE=false
cqlsh.py YOUR-COSMOSDB-ACCOUNT-NAME.cassandra.cosmosdb.azure.com 10350 -u YOUR-COSMOSDB-ACCOUNT-NAME -p YOUR-COSMOSDB-ACCOUNT-KEY --ssl
```

1. Azure Databricks

The article below covers Azure Databricks cluster provisioning, cluster configuration for connecting to Azure Cosmos DB Cassandra API, and several sample notebooks that cover DDL operations, DML operations and more.
Work with Azure Cosmos DB Cassandra API from Azure databricks

2. Azure HDInsight-Spark

The article below covers HDinsight-Spark service, provisioning, cluster configuration for connecting to Azure Cosmos DB Cassandra API, and several sample notebooks that cover DDL operations, DML operations and more.
Work with Azure Cosmos DB Cassandra API from Azure HDInsight-Spark

3. Spark environment in general

While the sections above were specific to Azure Spark-based PaaS services, this section covers any general Spark environment. Connector dependencies, imports, and Spark session configuration are detailed below. The "Next steps" section covers code samples for DDL operations, DML operations and more.

Connector dependencies:

1. Add the maven coordinates to get the Cassandra connector for Spark
2. Add the maven coordinates for the Azure Cosmos DB helper library for Cassandra API

Imports:

```
import org.apache.spark.sql.cassandra._
//Spark connector
import com.datastax.spark.connector._
import com.datastax.spark.connector.cql.CassandraConnector

//CosmosDB library for multiple retry
import com.microsoft.azure.cosmosdb.cassandra
```

Spark session configuration:

```
//Connection-related
spark.conf.set("spark.cassandra.connection.host","YOUR_ACCOUNT_NAME.cassandra.cosmosdb.azure.com")
spark.conf.set("spark.cassandra.connection.port","10350")
spark.conf.set("spark.cassandra.connection.ssl.enabled","true")
spark.conf.set("spark.cassandra.auth.username","YOUR_ACCOUNT_NAME")
spark.conf.set("spark.cassandra.auth.password","YOUR_ACCOUNT_KEY")
spark.conf.set("spark.cassandra.connection.factory",
"com.microsoft.azure.cosmosdb.cassandra.CosmosDbConnectionFactory")

//Throughput-related. You can adjust the values as needed
spark.conf.set("spark.cassandra.output.batch.size.rows", "1")
spark.conf.set("spark.cassandra.connection.connections_per_executor_max", "10")
spark.conf.set("spark.cassandra.output.concurrent.writes", "1000")
spark.conf.set("spark.cassandra.concurrent.reads", "512")
spark.conf.set("spark.cassandra.output.batch.grouping.buffer.size", "1000")
spark.conf.set("spark.cassandra.connection.keep_alive_ms", "600000000")
```

Next steps

The following articles demonstrate Spark integration with Azure Cosmos DB Cassandra API.

- DDL operations
- Create/insert operations
- Read operations
- Upsert operations
- Delete operations
- Aggregation operations
- Table copy operations

Access Azure Cosmos DB Cassandra API data from Azure Databricks

11/2/2020 • 2 minutes to read • Edit Online

APPLIES TO: Cassandra API

This article details how to workwith Azure Cosmos DB Cassandra API from Spark on Azure Databricks.

Prerequisites

- Provision an Azure Cosmos DB Cassandra API account
- Review the basics of connecting to Azure Cosmos DB Cassandra API
- Provision an Azure Databricks cluster
- Review the code samples for working with Cassandra API
- Use cqlsh for validation if you so prefer
- **Cassandra API instance configuration for Cassandra connector:**

  The connector for Cassandra API requires the Cassandra connection details to be initialized as part of the spark context. When you launch a Databricks notebook, the spark context is already initialized and it is not advisable to stop and reinitialize it. One solution is to add the Cassandra API instance configuration at a cluster level, in the cluster spark configuration. This is a one-time activity per cluster. Add the following code to the Spark configuration as a space separated key value pair:

  ```
  spark.cassandra.connection.host YOUR_COSMOSDB_ACCOUNT_NAME.cassandra.cosmosdb.azure.com
  spark.cassandra.connection.port 10350
  spark.cassandra.connection.ssl.enabled true
  spark.cassandra.auth.username YOUR_COSMOSDB_ACCOUNT_NAME
  spark.cassandra.auth.password YOUR_COSMOSDB_KEY
  ```

Add the required dependencies

- **Cassandra Spark connector:** - To integrate Azure Cosmos DB Cassandra API with Spark, the Cassandra connector should be attached to the Azure Databricks cluster. To attach the cluster:
  - Review the Databricks runtime version, the Spark version. Then find the maven coordinates that are compatible with the Cassandra Spark connector, and attach it to the cluster. See "Upload a Maven package or Spark package" article to attach the connector library to the cluster. For example, maven coordinate for "Databricks Runtime version 4.3", "Spark 2.3.1", and "Scala 2.11" is `spark-cassandra-connector_2.11-2.3.1`
- **Azure Cosmos DB Cassandra API-specific library:** - A custom connection factory is required to configure the retry policy from the Cassandra Spark connector to Azure Cosmos DB Cassandra API. Add the `com.microsoft.azure.cosmosdb:azure-cosmos-cassandra-spark-helper:1.0.0` maven coordinates to attach the library to the cluster.

Sample notebooks

A list of Azure Databricks sample notebooks are available in GitHub repo for you to download. These samples include how to connect to Azure Cosmos DB Cassandra API from Spark and perform different CRUD operations on the data. You can also import all the notebooks into your Databricks cluster workspace and run it.

Accessing Azure Cosmos DB Cassandra API from Spark Scala programs

Spark programs to be run as automated processes on Azure Databricks are submitted to the cluster by using spark-submit) and scheduled to run through the Azure Databricks jobs.

The following are links to help you get started building Spark Scala programs to interact with Azure Cosmos DB Cassandra API.

- How to connect to Azure Cosmos DB Cassandra API from a Spark Scala program
- How to run a Spark Scala program as an automated job on Azure Databricks
- Complete list of code samples for working with Cassandra API

Next steps

Get started with creating a Cassandra API account, database, and a table by using a Java application.

Access Azure Cosmos DB Cassandra API from Spark on YARN with HDInsight

11/2/2020 • 3 minutes to read • Edit Online

APPLIES TO: Cassandra API

This article covers how to access Azure Cosmos DB Cassandra API from Spark on YARN with HDInsight-Spark from spark-shell. HDInsight is Microsoft's Hortonworks Hadoop PaaS on Azure that leverages object storage for HDFS, and comes in several flavors including Spark. While the content in this document references HDInsight-Spark, it is applicable to all Hadoop distributions.

Prerequisites

- Provision Azure Cosmos DB Cassandra API
- Review the basics of connecting to Azure Cosmos DB Cassandra API
- Provision a HDInsight-Spark cluster
- Review the code samples for working with Cassandra API
- Use cqlsh for validation if you so prefer
- **Cassandra API configuration in Spark2** - The Spark connector for Cassandra requires that the Cassandra connection details to be initialized as part of the Spark context. When you launch a Jupyter notebook, the spark session and context are already initialized and it is not advisable to stop and reinitialize the Spark context unless it's complete with every configuration set as part of the HDInsight default Jupyter notebook start-up. One workaround is to add the Cassandra instance details to Ambari, Spark2 service configuration directly. This is a one-time activity per cluster that requires a Spark2 service restart.

  1. Go to Ambari, Spark2 service and select configs
  2. Then go to custom spark2-defaults and add a new property with the following, and restart Spark2 service:

```
spark.cassandra.connection.host=YOUR_COSMOSDB_ACCOUNT_NAME.cassandra.cosmosdb.azure.com<br>
spark.cassandra.connection.port=10350<br>
spark.cassandra.connection.ssl.enabled=true<br>
spark.cassandra.auth.username=YOUR_COSMOSDB_ACCOUNT_NAME<br>
spark.cassandra.auth.password=YOUR_COSMOSDB_KEY<br>
```

Access Azure Cosmos DB Cassandra API from Spark shell

Spark shell is used for testing/exploration purposes.

- Launch spark-shell with the required maven dependencies compatible with your cluster's Spark version.

```
spark-shell --packages "com.datastax.spark:spark-cassandra-
connector_2.11:2.3.0,com.microsoft.azure.cosmosdb:azure-cosmos-cassandra-spark-helper:1.0.0"
```

- Execute some DDL and DML operations

```
import org.apache.spark.rdd.RDD
import org.apache.spark.{SparkConf, SparkContext}

import spark.implicits._
import org.apache.spark.sql.functions._
import org.apache.spark.sql.Column
import org.apache.spark.sql.types.{StructType, StructField, StringType,
IntegerType,LongType,FloatType,DoubleType, TimestampType}
import org.apache.spark.sql.cassandra._

//Spark connector
import com.datastax.spark.connector._
import com.datastax.spark.connector.cql.CassandraConnector

//CosmosDB library for multiple retry
import com.microsoft.azure.cosmosdb.cassandra

// Specify connection factory for Cassandra
spark.conf.set("spark.cassandra.connection.factory",
"com.microsoft.azure.cosmosdb.cassandra.CosmosDbConnectionFactory")

// Parallelism and throughput configs
spark.conf.set("spark.cassandra.output.batch.size.rows", "1")
spark.conf.set("spark.cassandra.connection.connections_per_executor_max", "10")
spark.conf.set("spark.cassandra.output.concurrent.writes", "100")
spark.conf.set("spark.cassandra.concurrent.reads", "512")
spark.conf.set("spark.cassandra.output.batch.grouping.buffer.size", "1000")
spark.conf.set("spark.cassandra.connection.keep_alive_ms", "60000000") //Increase this number as needed
```

- Run CRUD operations

```
//1) Create table if it does not exist
val cdbConnector = CassandraConnector(sc)
cdbConnector.withSessionDo(session => session.execute("CREATE TABLE IF NOT EXISTS
books_ks.books(book_id TEXT PRIMARY KEY,book_author TEXT, book_name TEXT,book_pub_year INT,book_price
FLOAT) WITH cosmosdb_provisioned_throughput=4000;"))

//2) Delete data from potential prior runs
cdbConnector.withSessionDo(session => session.execute("DELETE FROM books_ks.books WHERE book_id IN
('b00300','b00001','b00023','b00501','b09999','b01001','b00999','b03999','b02999','b000009');"))

//3) Generate a few rows
val booksDF = Seq(
("b00001", "Arthur Conan Doyle", "A study in scarlet", 1887,11.33),
("b00023", "Arthur Conan Doyle", "A sign of four", 1890,22.45),
("b01001", "Arthur Conan Doyle", "The adventures of Sherlock Holmes", 1892,19.83),
("b00501", "Arthur Conan Doyle", "The memoirs of Sherlock Holmes", 1893,14.22),
("b00300", "Arthur Conan Doyle", "The hounds of Baskerville", 1901,12.25)
).toDF("book_id", "book_author", "book_name", "book_pub_year","book_price")

//4) Persist
booksDF.write.mode("append").format("org.apache.spark.sql.cassandra").options(Map( "table" -> "books",
"keyspace" -> "books_ks", "output.consistency.level" -> "ALL", "ttl" -> "10000000")).save()

//5) Read the data in the table
spark.read.format("org.apache.spark.sql.cassandra").options(Map( "table" -> "books", "keyspace" ->
"books_ks")).load.show
```

Access Azure Cosmos DB Cassandra API from Jupyter notebooks

HDInsight-Spark comes with Zeppelin and Jupyter notebook services. They are both web-based notebook environments that support Scala and Python. Notebooks are great for interactive exploratory analytics and collaboration, but not meant for operational/productionized processes.

The following Jupyter notebooks can be uploaded into your HDInsight Spark cluster and provide ready samples for working with Azure Cosmos DB Cassandra API. Be sure to review the first notebook `1.0-ReadMe.ipynb` to review Spark service configuration for connecting to Azure Cosmos DB Cassandra API.

Download these notebooks under azure-cosmos-db-cassandra-api-spark-notebooks-jupyter to your machine.

How to upload:

When you launch Jupyter, navigate to Scala. Create a directory first and then upload the notebooks to the directory. The upload button is on the top, right hand-side.

How to run:

Run through the notebooks, and each notebook cell sequentially. Click the run button at the top of each notebook to execute all cells, or shift+enter for each cell.

Access with Azure Cosmos DB Cassandra API from your Spark Scala program

For automated processes in production, Spark programs are submitted to the cluster via spark-submit.

Next steps

- How to build a Spark Scala program in an IDE and submit it to the HDInsight Spark cluster through Livy for execution
- How to connect to Azure Cosmos DB Cassandra API from a Spark Scala program
- Complete list of code samples for working with Cassandra API

DDL operations in Azure Cosmos DB Cassandra API from Spark

11/2/2020 • 2 minutes to read • Edit Online

APPLIES TO: Cassandra API

This article details keyspace and table DDL operations against Azure Cosmos DB Cassandra API from Spark.

Cassandra API-related configuration

```
import org.apache.spark.sql.cassandra._

//Spark connector
import com.datastax.spark.connector._
import com.datastax.spark.connector.cql.CassandraConnector

//CosmosDB library for multiple retry
import com.microsoft.azure.cosmosdb.cassandra

//Connection-related
spark.conf.set("spark.cassandra.connection.host","YOUR_ACCOUNT_NAME.cassandra.cosmosdb.azure.com")
spark.conf.set("spark.cassandra.connection.port","10350")
spark.conf.set("spark.cassandra.connection.ssl.enabled","true")
spark.conf.set("spark.cassandra.auth.username","YOUR_ACCOUNT_NAME")
spark.conf.set("spark.cassandra.auth.password","YOUR_ACCOUNT_KEY")
spark.conf.set("spark.cassandra.connection.factory",
"com.microsoft.azure.cosmosdb.cassandra.CosmosDbConnectionFactory")

//Throughput-related...adjust as needed
spark.conf.set("spark.cassandra.output.batch.size.rows", "1")
spark.conf.set("spark.cassandra.connection.connections_per_executor_max", "10")
spark.conf.set("spark.cassandra.output.concurrent.writes", "1000")
spark.conf.set("spark.cassandra.concurrent.reads", "512")
spark.conf.set("spark.cassandra.output.batch.grouping.buffer.size", "1000")
spark.conf.set("spark.cassandra.connection.keep_alive_ms", "600000000")
```

Keyspace DDL operations

Create a keyspace

```
//Cassandra connector instance
val cdbConnector = CassandraConnector(sc)

// Create keyspace
cdbConnector.withSessionDo(session => session.execute("CREATE KEYSPACE IF NOT EXISTS books_ks WITH REPLICATION
= {'class': 'SimpleStrategy', 'replication_factor': 1 } "))
```

Validate in cqlsh

Run the following command in cqlsh and you should see the keyspace you created earlier.

```
DESCRIBE keyspaces;
```

Drop a keyspace

```
val cdbConnector = CassandraConnector(sc)
cdbConnector.withSessionDo(session => session.execute("DROP KEYSPACE books_ks"))
```

Validate in cqlsh

```
DESCRIBE keyspaces;
```

Table DDL operations

Considerations:

- Throughput can be assigned at the table level by using the create table statement.
- One partition key can store 20 GB of data.
- One record can store a maximum of 2 MB of data.
- One partition key range can store multiple partition keys.

Create a table

```
cdbConnector.withSessionDo(session => session.execute("CREATE TABLE IF NOT EXISTS books_ks1.books(book_id
TEXT,book_author TEXT, book_name TEXT,book_pub_year INT,book_price FLOAT, PRIMARY KEY(book_id,book_pub_year))
WITH cosmosdb_provisioned_throughput=4000 , WITH default_time_to_live=630720000;"))
```

Validate in cqlsh

Run the following command in cqlsh and you should see the table named “books:

```
USE books_ks;
DESCRIBE books;
```

Provisioned throughput and default TTL values are not shown in the output of the previous command, you can get these values from the portal.

Alter table

You can alter the following values by using the alter table command:

- provisioned throughput
- time-to-live value
  Column changes are currently not supported.

```
val cdbConnector = CassandraConnector(sc)
cdbConnector.withSessionDo(session => session.execute("ALTER TABLE books_ks.books WITH
cosmosdb_provisioned_throughput=8000, WITH default_time_to_live=0;"))
```

Drop table

```
val cdbConnector = CassandraConnector(sc)
cdbConnector.withSessionDo(session => session.execute("DROP TABLE IF EXISTS books_ks.books;"))
```

Validate in cqlsh

Run the following command in cqlsh and you should see that the “books” table is no longer available:

```
USE books_ks;
DESCRIBE tables;
```

Next steps

After creating the keyspace and the table, proceed to the following articles for CRUD operations and more:

- Create/insert operations
- Read operations
- Upsert operations
- Delete operations
- Aggregation operations
- Table copy operations

Create/Insert data into Azure Cosmos DB Cassandra API from Spark

11/2/2020 • 2 minutes to read • Edit Online

APPLIES TO: Cassandra API

This article describes how to insert sample data into a table in Azure Cosmos DB Cassandra API from Spark.

Cassandra API configuration

```
import org.apache.spark.sql.cassandra._
//Spark connector
import com.datastax.spark.connector._
import com.datastax.spark.connector.cql.CassandraConnector

//CosmosDB library for multiple retry
import com.microsoft.azure.cosmosdb.cassandra

//Connection-related
spark.conf.set("spark.cassandra.connection.host","YOUR_ACCOUNT_NAME.cassandra.cosmosdb.azure.com")
spark.conf.set("spark.cassandra.connection.port","10350")
spark.conf.set("spark.cassandra.connection.ssl.enabled","true")
spark.conf.set("spark.cassandra.auth.username","YOUR_ACCOUNT_NAME")
spark.conf.set("spark.cassandra.auth.password","YOUR_ACCOUNT_KEY")
spark.conf.set("spark.cassandra.connection.factory",
"com.microsoft.azure.cosmosdb.cassandra.CosmosDbConnectionFactory")
//Throughput-related...adjust as needed
spark.conf.set("spark.cassandra.output.batch.size.rows", "1")
spark.conf.set("spark.cassandra.connection.connections_per_executor_max", "10")
spark.conf.set("spark.cassandra.output.concurrent.writes", "1000")
spark.conf.set("spark.cassandra.concurrent.reads", "512")
spark.conf.set("spark.cassandra.output.batch.grouping.buffer.size", "1000")
spark.conf.set("spark.cassandra.connection.keep_alive_ms", "600000000")
```

Dataframe API

Create a Dataframe with sample data

```
// Generate a dataframe containing five records
val booksDF = Seq(
   ("b00001", "Arthur Conan Doyle", "A study in scarlet", 1887),
   ("b00023", "Arthur Conan Doyle", "A sign of four", 1890),
   ("b01001", "Arthur Conan Doyle", "The adventures of Sherlock Holmes", 1892),
   ("b00501", "Arthur Conan Doyle", "The memoirs of Sherlock Holmes", 1893),
   ("b00300", "Arthur Conan Doyle", "The hounds of Baskerville", 1901)
).toDF("book_id", "book_author", "book_name", "book_pub_year")

//Review schema
booksDF.printSchema

//Print
booksDF.show
```

> **NOTE**
> "Create if not exists" functionality, at a row level, is not yet supported.

Persist to Azure Cosmos DB Cassandra API

When saving data, you can also set time-to-live and consistency policy settings as shown in the following example:

```
//Persist
booksDF.write
  .mode("append")
  .format("org.apache.spark.sql.cassandra")
  .options(Map( "table" -> "books", "keyspace" -> "books_ks", "output.consistency.level" -> "ALL", "ttl" ->
"10000000"))
  .save()
```

> **NOTE**
> Column-level TTL is not supported yet.

Validate in cqlsh

```
use books_ks;
select * from books;
```

Resilient Distributed Database (RDD) API

Create a RDD with sample data

```
//Delete records created in the previous section
val cdbConnector = CassandraConnector(sc)
cdbConnector.withSessionDo(session => session.execute("delete from books_ks.books where book_id in
('b00300','b00001','b00023','b00501','b09999','b01001','b00999','b03999','b02999');"))

//Create RDD
val booksRDD = sc.parallelize(Seq(
   ("b00001", "Arthur Conan Doyle", "A study in scarlet", 1887),
   ("b00023", "Arthur Conan Doyle", "A sign of four", 1890),
   ("b01001", "Arthur Conan Doyle", "The adventures of Sherlock Holmes", 1892),
   ("b00501", "Arthur Conan Doyle", "The memoirs of Sherlock Holmes", 1893),
   ("b00300", "Arthur Conan Doyle", "The hounds of Baskerville", 1901)
))

//Review
booksRDD.take(2).foreach(println)
```

> **NOTE**
> Create if not exists functionality is not yet supported.

Persist to Azure Cosmos DB Cassandra API

When saving data to Cassandra API, you can also set time-to-live and consistency policy settings as shown in the following example:

```
import com.datastax.spark.connector.writer._

//Persist
booksRDD.saveToCassandra("books_ks", "books", SomeColumns("book_id", "book_author", "book_name",
"book_pub_year"),writeConf = WriteConf(ttl = TTLOption.constant(900000),consistencyLevel =
ConsistencyLevel.ALL))
```

Validate in cqlsh

```
use books_ks;
select * from books;
```

Next steps

After inserting data into the Azure Cosmos DB Cassandra API table, proceed to the following articles to perform other operations on the data stored in Cosmos DB Cassandra API:

- Read operations
- Upsert operations
- Delete operations
- Aggregation operations
- Table copy operations

Read data from Azure Cosmos DB Cassandra API tables using Spark

11/2/2020 • 2 minutes to read • Edit Online

APPLIES TO: Cassandra API

This article describes how to read data stored in Azure Cosmos DB Cassandra API from Spark.

Cassandra API configuration

```
import org.apache.spark.sql.cassandra._
//Spark connector
import com.datastax.spark.connector._
import com.datastax.spark.connector.cql.CassandraConnector

//CosmosDB library for multiple retry
import com.microsoft.azure.cosmosdb.cassandra

//Connection-related
spark.conf.set("spark.cassandra.connection.host","YOUR_ACCOUNT_NAME.cassandra.cosmosdb.azure.com")
spark.conf.set("spark.cassandra.connection.port","10350")
spark.conf.set("spark.cassandra.connection.ssl.enabled","true")
spark.conf.set("spark.cassandra.auth.username","YOUR_ACCOUNT_NAME")
spark.conf.set("spark.cassandra.auth.password","YOUR_ACCOUNT_KEY")
spark.conf.set("spark.cassandra.connection.factory",
"com.microsoft.azure.cosmosdb.cassandra.CosmosDbConnectionFactory")
//Throughput-related...adjust as needed
spark.conf.set("spark.cassandra.output.batch.size.rows", "1")
spark.conf.set("spark.cassandra.connection.connections_per_executor_max", "10")
spark.conf.set("spark.cassandra.output.concurrent.writes", "1000")
spark.conf.set("spark.cassandra.concurrent.reads", "512")
spark.conf.set("spark.cassandra.output.batch.grouping.buffer.size", "1000")
spark.conf.set("spark.cassandra.connection.keep_alive_ms", "600000000")
```

Dataframe API

Read table using session.read.format command

```
val readBooksDF = sqlContext
  .read
  .format("org.apache.spark.sql.cassandra")
  .options(Map( "table" -> "books", "keyspace" -> "books_ks"))
  .load

readBooksDF.explain
readBooksDF.show
```

Read table using spark.read.cassandraFormat

```
val readBooksDF = spark.read.cassandraFormat("books", "books_ks", "").load()
```

Read specific columns in table

```
val readBooksDF = spark
  .read
  .format("org.apache.spark.sql.cassandra")
  .options(Map( "table" -> "books", "keyspace" -> "books_ks"))
  .load
  .select("book_name","book_author", "book_pub_year")

readBooksDF.printSchema
readBooksDF.explain
readBooksDF.show
```

Apply filters

You can push down predicates to the database to allow for better optimized Spark queries. A predicate is a condition on a query that returns true or false, typically located in the WHERE clause. A predicate push down filters the data in the database query, reducing the number of entries retrieved from the database and improving query performance. By default the Spark Dataset API will automatically push down valid WHERE clauses to the database.

```
val df = spark.read.cassandraFormat("books", "books_ks").load
df.explain
val dfWithPushdown = df.filter(df("book_pub_year") > 1891)
dfWithPushdown.explain

readBooksDF.printSchema
readBooksDF.explain
readBooksDF.show
```

The PushedFilters section of the physical plan includes the GreaterThan push down filter.

```
1 val df = spark.read.cassandraFormat("books", "books_ks").load
2 df.explain
3 val dfWithPushdown = df.filter(df("book_pub_year") > 1891)
4 dfWithPushdown.explain

df: org.apache.spark.sql.DataFrame = [book_id: string, book_pub_year: integer ... 3 more fields]
dfWithPushdown: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] = [book_id: string, book_pub_year: integer ... 3 more fields]
== Physical Plan ==
*(1) Scan org.apache.spark.sql.cassandra.CassandraSourceRelation@6cf0038b [book_id#534,book_pub_year#535,book_author#536,book_name#537,book_price#538] PushedFilters: [], ReadSchema: struct<b
ook_id:string,book_pub_year:int,book_author:string,book_name:string,book_price:float>
== Physical Plan ==
*(1) Filter isnotnull(book_pub_year#535)
+- *(1) Scan org.apache.spark.sql.cassandra.CassandraSourceRelation@6cf0038b [book_id#534,book_pub_year#535,book_author#536,book_name#537,book_price#538] PushedFilters: [IsNotNull(book_pub_y
ear), *GreaterThan(book_pub_year,1891)], ReadSchema: struct<book_id:string,book_pub_year:int,book_author:string,book_name:string,book_price:float>
df: org.apache.spark.sql.DataFrame = [book_id: string, book_pub_year: int ... 3 more fields]
dfWithPushdown: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] = [book_id: string, book_pub_year: int ... 3 more fields]
Command took 0.67 seconds -- by thvankra@microsoft.com at 9/1/2020, 9:59:09 PM on cluster1
```

RDD API

Read table

```
val bookRDD = sc.cassandraTable("books_ks", "books")
bookRDD.take(5).foreach(println)
```

Read specific columns in table

```
val booksRDD = sc.cassandraTable("books_ks", "books").select("book_id","book_name").cache
booksRDD.take(5).foreach(println)
```

SQL Views

Create a temporary view from a dataframe

```
spark
  .read
  .format("org.apache.spark.sql.cassandra")
  .options(Map( "table" -> "books", "keyspace" -> "books_ks"))
  .load.createOrReplaceTempView("books_vw")
```

Run queries against the view

```
select * from books_vw where book_pub_year > 1891
```

Next steps

The following are additional articles on working with Azure Cosmos DB Cassandra API from Spark:

- Upsert operations
- Delete operations
- Aggregation operations
- Table copy operations

Upsert data into Azure Cosmos DB Cassandra API from Spark

11/2/2020 • 2 minutes to read • Edit Online

APPLIES TO: Cassandra API

This article describes how to upsert data into Azure Cosmos DB Cassandra API from Spark.

Cassandra API configuration

```
import org.apache.spark.sql.cassandra._
//Spark connector
import com.datastax.spark.connector._
import com.datastax.spark.connector.cql.CassandraConnector

//CosmosDB library for multiple retry
import com.microsoft.azure.cosmosdb.cassandra

//Connection-related
spark.conf.set("spark.cassandra.connection.host","YOUR_ACCOUNT_NAME.cassandra.cosmosdb.azure.com")
spark.conf.set("spark.cassandra.connection.port","10350")
spark.conf.set("spark.cassandra.connection.ssl.enabled","true")
spark.conf.set("spark.cassandra.auth.username","YOUR_ACCOUNT_NAME")
spark.conf.set("spark.cassandra.auth.password","YOUR_ACCOUNT_KEY")
spark.conf.set("spark.cassandra.connection.factory",
"com.microsoft.azure.cosmosdb.cassandra.CosmosDbConnectionFactory")
//Throughput-related...adjust as needed
spark.conf.set("spark.cassandra.output.batch.size.rows", "1")
spark.conf.set("spark.cassandra.connection.connections_per_executor_max", "10")
spark.conf.set("spark.cassandra.output.concurrent.writes", "1000")
spark.conf.set("spark.cassandra.concurrent.reads", "512")
spark.conf.set("spark.cassandra.output.batch.grouping.buffer.size", "1000")
spark.conf.set("spark.cassandra.connection.keep_alive_ms", "600000000")
```

Dataframe API

Create a dataframe

```
// (1) Update: Changing author name to include prefix of "Sir"
// (2) Insert: adding a new book

val booksUpsertDF = Seq(
    ("b00001", "Sir Arthur Conan Doyle", "A study in scarlet", 1887),
    ("b00023", "Sir Arthur Conan Doyle", "A sign of four", 1890),
    ("b01001", "Sir Arthur Conan Doyle", "The adventures of Sherlock Holmes", 1892),
    ("b00501", "Sir Arthur Conan Doyle", "The memoirs of Sherlock Holmes", 1893),
    ("b00300", "Sir Arthur Conan Doyle", "The hounds of Baskerville", 1901),
    ("b09999", "Sir Arthur Conan Doyle", "The return of Sherlock Holmes", 1905)
    ).toDF("book_id", "book_author", "book_name", "book_pub_year")
booksUpsertDF.show()
```

Upsert data

```
// Upsert is no different from create
booksUpsertDF.write
  .mode("append")
  .format("org.apache.spark.sql.cassandra")
  .options(Map( "table" -> "books", "keyspace" -> "books_ks"))
  .save()
```

Update data

```
//This runs on the driver, leverage only for one off updates
cdbConnector.withSessionDo(session => session.execute("update books_ks.books set book_price=99.33 where
book_id ='b00300';"))
```

RDD API

> **NOTE**
>
> Upsert from the RDD API is same as the create operation

Next steps

Proceed to the following articles to perform other operations on the data stored in Azure Cosmos DB Cassandra API tables:

- Delete operations
- Aggregation operations
- Table copy operations

Delete data in Azure Cosmos DB Cassandra API tables from Spark

11/2/2020 • 5 minutes to read • Edit Online

APPLIES TO: Cassandra API

This article describes how to delete data in Azure Cosmos DB Cassandra API tables from Spark.

Cassandra API configuration

```
import org.apache.spark.sql.cassandra._
//Spark connector
import com.datastax.spark.connector._
import com.datastax.spark.connector.cql.CassandraConnector

//CosmosDB library for multiple retry
import com.microsoft.azure.cosmosdb.cassandra

//Connection-related
spark.conf.set("spark.cassandra.connection.host","YOUR_ACCOUNT_NAME.cassandra.cosmosdb.azure.com")
spark.conf.set("spark.cassandra.connection.port","10350")
spark.conf.set("spark.cassandra.connection.ssl.enabled","true")
spark.conf.set("spark.cassandra.auth.username","YOUR_ACCOUNT_NAME")
spark.conf.set("spark.cassandra.auth.password","YOUR_ACCOUNT_KEY")
spark.conf.set("spark.cassandra.connection.factory",
"com.microsoft.azure.cosmosdb.cassandra.CosmosDbConnectionFactory")
//Throughput-related...adjust as needed
spark.conf.set("spark.cassandra.output.batch.size.rows", "1")
spark.conf.set("spark.cassandra.connection.connections_per_executor_max", "10")
spark.conf.set("spark.cassandra.output.concurrent.writes", "1000")
spark.conf.set("spark.cassandra.concurrent.reads", "512")
spark.conf.set("spark.cassandra.output.batch.grouping.buffer.size", "1000")
spark.conf.set("spark.cassandra.connection.keep_alive_ms", "600000000")
```

Sample data generator

We will use this code fragment to generate sample data:

```
//Create dataframe
val booksDF = Seq(
   ("b00001", "Arthur Conan Doyle", "A study in scarlet", 1887,11.33),
   ("b00023", "Arthur Conan Doyle", "A sign of four", 1890,22.45),
   ("b01001", "Arthur Conan Doyle", "The adventures of Sherlock Holmes", 1892,19.83),
   ("b00501", "Arthur Conan Doyle", "The memoirs of Sherlock Holmes", 1893,14.22),
   ("b00300", "Arthur Conan Doyle", "The hounds of Baskerville", 1901,12.25)
).toDF("book_id", "book_author", "book_name", "book_pub_year","book_price")

//Persist
booksDF.write
  .mode("append")
  .format("org.apache.spark.sql.cassandra")
  .options(Map( "table" -> "books", "keyspace" -> "books_ks", "output.consistency.level" -> "ALL", "ttl" ->
"10000000"))
  .save()
```

Dataframe API

Delete rows that match a condition

```
//1) Create dataframe
val deleteBooksDF = spark
  .read
  .format("org.apache.spark.sql.cassandra")
  .options(Map( "table" -> "books", "keyspace" -> "books_ks"))
  .load
  .filter("book_pub_year = 1887")

//2) Review execution plan
deleteBooksDF.explain

//3) Review table data before execution
println("==================")
println("1) Before")
deleteBooksDF.show
println("==================")

//4) Delete selected records in dataframe
println("==================")
println("2a) Starting delete")

//Reuse connection for each partition
val cdbConnector = CassandraConnector(sc)
deleteBooksDF.foreachPartition(partition => {
  cdbConnector.withSessionDo(session =>
    partition.foreach{ book =>
        val delete = s"DELETE FROM books_ks.books where book_id='"+book.getString(0) +"';"
        session.execute(delete)
    })
})

println("2b) Completed delete")
println("==================")

//5) Review table data after delete operation
println("3) After")
spark
  .read
  .format("org.apache.spark.sql.cassandra")
  .options(Map( "table" -> "books", "keyspace" -> "books_ks"))
  .load
  .show
```

Output:

```
== Physical Plan ==
*(1) Filter (isnotnull(book_pub_year#486) && (book_pub_year#486 = 1887))
+- *(1) Scan org.apache.spark.sql.cassandra.CassandraSourceRelation@197cfae4
[book_id#482,book_author#483,book_name#484,book_price#485,book_pub_year#486]
PushedFilters: [IsNotNull(book_pub_year), EqualTo(book_pub_year,1887)],
ReadSchema: struct<book_id:string,book_author:string,book_name:string,book_price:float,book_pub_year:int>
==================
1) Before
+-------+------------------+------------------+----------+-------------+
|book_id|       book_author|         book_name|book_price|book_pub_year|
+-------+------------------+------------------+----------+-------------+
| b00001|Arthur Conan Doyle|A study in scarlet|     11.33|         1887|
+-------+------------------+------------------+----------+-------------+

==================
==================
2a) Starting delete
2b) Completed delete
==================
3) After
+-------+------------------+--------------------+----------+-------------+
|book_id|       book_author|           book_name|book_price|book_pub_year|
+-------+------------------+--------------------+----------+-------------+
| b00300|Arthur Conan Doyle|The hounds of Bas...|     12.25|         1901|
| b03999|Arthur Conan Doyle|The adventure of ...|      null|         1892|
| b00023|Arthur Conan Doyle|      A sign of four|     22.45|         1890|
| b00501|Arthur Conan Doyle|The memoirs of Sh...|     14.22|         1893|
| b01001|Arthur Conan Doyle|The adventures of...|     19.83|         1892|
| b02999|Arthur Conan Doyle|  A case of identity|      15.0|         1891|
+-------+------------------+--------------------+----------+-------------+

deleteBooksDF: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] = [book_id: string, book_author: string
... 3 more fields]
cdbConnector: com.datastax.spark.connector.cql.CassandraConnector =
com.datastax.spark.connector.cql.CassandraConnector@187deb43
```

Delete all the rows in the table

```
//1) Create dataframe
val deleteBooksDF = spark
  .read
  .format("org.apache.spark.sql.cassandra")
  .options(Map( "table" -> "books", "keyspace" -> "books_ks"))
  .load

//2) Review execution plan
deleteBooksDF.explain

//3) Review table data before execution
println("==================")
println("1) Before")
deleteBooksDF.show
println("==================")

//4) Delete records in dataframe
println("==================")
println("2a) Starting delete")

//Reuse connection for each partition
val cdbConnector = CassandraConnector(sc)
deleteBooksDF.foreachPartition(partition => {
  cdbConnector.withSessionDo(session =>
    partition.foreach{ book =>
        val delete = s"DELETE FROM books_ks.books where book_id='"+book.getString(0) +"';"
        session.execute(delete)
    })
})

println("2b) Completed delete")
println("==================")

//5) Review table data after delete operation
println("3) After")
spark
  .read
  .format("org.apache.spark.sql.cassandra")
  .options(Map( "table" -> "books", "keyspace" -> "books_ks"))
  .load
  .show
```

Output:

```
== Physical Plan ==
*(1) Scan org.apache.spark.sql.cassandra.CassandraSourceRelation@495377d7
[book_id#565,book_author#566,book_name#567,book_price#568,book_pub_year#569]
PushedFilters: [],
ReadSchema: struct<book_id:string,book_author:string,book_name:string,book_price:float,book_pub_year:int>
==================
1) Before
+-------+------------------+--------------------+----------+-------------+
|book_id|       book_author|           book_name|book_price|book_pub_year|
+-------+------------------+--------------------+----------+-------------+
| b00300|Arthur Conan Doyle|The hounds of Bas...|     12.25|         1901|
| b03999|Arthur Conan Doyle|The adventure of ...|      null|         1892|
| b00023|Arthur Conan Doyle|      A sign of four|     22.45|         1890|
| b00501|Arthur Conan Doyle|The memoirs of Sh...|     14.22|         1893|
| b01001|Arthur Conan Doyle|The adventures of...|     19.83|         1892|
| b02999|Arthur Conan Doyle|  A case of identity|      15.0|         1891|
+-------+------------------+--------------------+----------+-------------+

==================
==================
2a) Starting delete
2b) Completed delete
==================
3) After
+-------+-----------+---------+----------+-------------+
|book_id|book_author|book_name|book_price|book_pub_year|
+-------+-----------+---------+----------+-------------+
+-------+-----------+---------+----------+-------------+
```

RDD API

Delete all the rows in the table

```
//1) Create RDD with all rows
val deleteBooksRDD =
    sc.cassandraTable("books_ks", "books")

//2) Review table data before execution
println("==================")
println("1) Before")
deleteBooksRDD.collect.foreach(println)
println("==================")

//3) Delete selected records in dataframe
println("==================")
println("2a) Starting delete")

/* Option 1:
// Not supported currently
sc.cassandraTable("books_ks", "books")
  .where("book_pub_year = 1891")
  .deleteFromCassandra("books_ks", "books")
*/

//Option 2: CassandraConnector and CQL
//Reuse connection for each partition
val cdbConnector = CassandraConnector(sc)
deleteBooksRDD.foreachPartition(partition => {
    cdbConnector.withSessionDo(session =>
    partition.foreach{book =>
        val delete = s"DELETE FROM books_ks.books where book_id='"+ book.getString(0) +"';"
        session.execute(delete)
    }
   )
})

println("Completed delete")
println("==================")

println("2b) Completed delete")
println("==================")

//5) Review table data after delete operation
println("3) After")
sc.cassandraTable("books_ks", "books").collect.foreach(println)
```

Output:

```
==================
1) Before
CassandraRow{book_id: b00300, book_author: Arthur Conan Doyle, book_name: The hounds of Baskerville,
book_price: 12.25, book_pub_year: 1901}
CassandraRow{book_id: b00001, book_author: Arthur Conan Doyle, book_name: A study in scarlet, book_price:
11.33, book_pub_year: 1887}
CassandraRow{book_id: b00023, book_author: Arthur Conan Doyle, book_name: A sign of four, book_price: 22.45,
book_pub_year: 1890}
CassandraRow{book_id: b00501, book_author: Arthur Conan Doyle, book_name: The memoirs of Sherlock Holmes,
book_price: 14.22, book_pub_year: 1893}
CassandraRow{book_id: b01001, book_author: Arthur Conan Doyle, book_name: The adventures of Sherlock Holmes,
book_price: 19.83, book_pub_year: 1892}
==================
==================
2a) Starting delete
Completed delete
==================
2b) Completed delete
==================
3) After
deleteBooksRDD:
com.datastax.spark.connector.rdd.CassandraTableScanRDD[com.datastax.spark.connector.CassandraRow] =
CassandraTableScanRDD[126] at RDD at CassandraRDD.scala:19
cdbConnector: com.datastax.spark.connector.cql.CassandraConnector =
com.datastax.spark.connector.cql.CassandraConnector@317927
```

Delete specific columns

```
//1) Create RDD
val deleteBooksRDD =
    sc.cassandraTable("books_ks", "books")

//2) Review table data before execution
println("==================")
println("1) Before")
deleteBooksRDD.collect.foreach(println)
println("==================")

//3) Delete specific column values
println("==================")
println("2a) Starting delete of book price")

sc.cassandraTable("books_ks", "books")
  .deleteFromCassandra("books_ks", "books",SomeColumns("book_price"))

println("Completed delete")
println("==================")

println("2b) Completed delete")
println("==================")

//5) Review table data after delete operation
println("3) After")
sc.cassandraTable("books_ks", "books").take(4).foreach(println)
```

Output:

```
==================
1) Before
CassandraRow{book_id: b00300, book_author: Arthur Conan Doyle, book_name: The hounds of Baskerville,
book_price: 20.0, book_pub_year: 1901}
CassandraRow{book_id: b00001, book_author: Arthur Conan Doyle, book_name: A study in scarlet, book_price:
23.0, book_pub_year: 1887}
CassandraRow{book_id: b00023, book_author: Arthur Conan Doyle, book_name: A sign of four, book_price: 11.0,
book_pub_year: 1890}
CassandraRow{book_id: b00501, book_author: Arthur Conan Doyle, book_name: The memoirs of Sherlock Holmes,
book_price: 5.0, book_pub_year: 1893}
CassandraRow{book_id: b01001, book_author: Arthur Conan Doyle, book_name: The adventures of Sherlock Holmes,
book_price: 10.0, book_pub_year: 1892}
==================
==================
2a) Starting delete of book price
Completed delete
==================
2b) Completed delete
==================
3) After
CassandraRow{book_id: b00300, book_author: Arthur Conan Doyle, book_name: The hounds of Baskerville,
book_price: null, book_pub_year: 1901}
CassandraRow{book_id: b00001, book_author: Arthur Conan Doyle, book_name: A study in scarlet, book_price:
null, book_pub_year: 1887}
CassandraRow{book_id: b00023, book_author: Arthur Conan Doyle, book_name: A sign of four, book_price: null,
book_pub_year: 1890}
CassandraRow{book_id: b00501, book_author: Arthur Conan Doyle, book_name: The memoirs of Sherlock Holmes,
book_price: null, book_pub_year: 1893}
deleteBooksRDD:
com.datastax.spark.connector.rdd.CassandraTableScanRDD[com.datastax.spark.connector.CassandraRow] =
CassandraTableScanRDD[145] at RDD at CassandraRDD.scala:19
```

Next steps

To perform aggregation and data copy operations, refer -

- Aggregation operations
- Table copy operations

Aggregate operations on Azure Cosmos DB Cassandra API tables from Spark

11/2/2020 • 3 minutes to read • Edit Online

APPLIES TO: Cassandra API

This article describes basic aggregation operations against Azure Cosmos DB Cassandra API tables from Spark.

> **NOTE**
>
> Server-side filtering, and server-side aggregation is currently not supported in Azure Cosmos DB Cassandra API.

Cassandra API configuration

```
import org.apache.spark.sql.cassandra._
//Spark connector
import com.datastax.spark.connector._
import com.datastax.spark.connector.cql.CassandraConnector

//CosmosDB library for multiple retry
import com.microsoft.azure.cosmosdb.cassandra

//Connection-related
spark.conf.set("spark.cassandra.connection.host","YOUR_ACCOUNT_NAME.cassandra.cosmosdb.azure.com")
spark.conf.set("spark.cassandra.connection.port","10350")
spark.conf.set("spark.cassandra.connection.ssl.enabled","true")
spark.conf.set("spark.cassandra.auth.username","YOUR_ACCOUNT_NAME")
spark.conf.set("spark.cassandra.auth.password","YOUR_ACCOUNT_KEY")
spark.conf.set("spark.cassandra.connection.factory",
"com.microsoft.azure.cosmosdb.cassandra.CosmosDbConnectionFactory")
//Throughput-related...adjust as needed
spark.conf.set("spark.cassandra.output.batch.size.rows", "1")
spark.conf.set("spark.cassandra.connection.connections_per_executor_max", "10")
spark.conf.set("spark.cassandra.output.concurrent.writes", "1000")
spark.conf.set("spark.cassandra.concurrent.reads", "512")
spark.conf.set("spark.cassandra.output.batch.grouping.buffer.size", "1000")
spark.conf.set("spark.cassandra.connection.keep_alive_ms", "600000000")
```

Sample data generator

```
// Generate a simple dataset containing five values
val booksDF = Seq(
   ("b00001", "Arthur Conan Doyle", "A study in scarlet", 1887,11.33),
   ("b00023", "Arthur Conan Doyle", "A sign of four", 1890,22.45),
   ("b01001", "Arthur Conan Doyle", "The adventures of Sherlock Holmes", 1892,19.83),
   ("b00501", "Arthur Conan Doyle", "The memoirs of Sherlock Holmes", 1893,14.22),
   ("b00300", "Arthur Conan Doyle", "The hounds of Baskerville", 1901,12.25)
).toDF("book_id", "book_author", "book_name", "book_pub_year","book_price")

booksDF.write
  .mode("append")
  .format("org.apache.spark.sql.cassandra")
  .options(Map( "table" -> "books", "keyspace" -> "books_ks", "output.consistency.level" -> "ALL", "ttl" ->
"10000000"))
  .save()
```

Count operation

RDD API

```
sc.cassandraTable("books_ks", "books").count
```

Output:

```
res48: Long = 5
```

Dataframe API

Count against dataframes is currently not supported. The sample below shows how to execute a dataframe count after persisting the dataframe to memory as a workaround.

Choose a storage option from the following available options, to avoid running into "out of memory" issues:

- MEMORY_ONLY: This is the default storage option. Stores RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, some partitions will not be cached and they are recomputed on the fly each time they're needed.
- MEMORY_AND_DISK: Stores RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, store the partitions that don't fit on disk, and whenever required, read them from the location they are stored.
- MEMORY_ONLY_SER (Java/Scala): Stores RDD as serialized Java objects- one-byte array per partition. This option is space-efficient when compared to deserialized objects, especially when using a fast serializer, but more CPU-intensive to read.
- MEMORY_AND_DISK_SER (Java/Scala): This storage option is like MEMORY_ONLY_SER, the only difference is that it spills partitions that don't fit in the disk memory instead of recomputing them when they're needed.
- DISK_ONLY: Stores the RDD partitions on the disk only.
- MEMORY_ONLY_2, MEMORY_AND_DISK_2...: Same as the levels above, but replicates each partition on two cluster nodes.
- OFF_HEAP (experimental): Similar to MEMORY_ONLY_SER, but it stores the data in off-heap memory, and it requires off-heap memory to be enabled ahead of time.

```
//Workaround
import org.apache.spark.storage.StorageLevel

//Read from source
val readBooksDF = spark
  .read
  .cassandraFormat("books", "books_ks", "")
  .load()

//Explain plan
readBooksDF.explain

//Materialize the dataframe
readBooksDF.persist(StorageLevel.MEMORY_ONLY)

//Subsequent execution against this DF hits the cache
readBooksDF.count

//Persist as temporary view
readBooksDF.createOrReplaceTempView("books_vw")
```

SQL

```
select * from books_vw;
select count(*) from books_vw where book_pub_year > 1900;
select count(book_id) from books_vw;
select book_author, count(*) as count from books_vw group by book_author;
select count(*) from books_vw;
```

Average operation

RDD API

```
sc.cassandraTable("books_ks", "books").select("book_price").as((c: Double) => c).mean
```

Output:

```
res24: Double = 16.016000175476073
```

Dataframe API

```
spark
  .read
  .cassandraFormat("books", "books_ks", "")
  .load()
  .select("book_price")
  .agg(avg("book_price"))
  .show
```

Output:

```
+------------------+
|   avg(book_price)|
+------------------+
|16.016000175476073|
+------------------+
```

SQL

```
select avg(book_price) from books_vw;
```

Output:

```
16.016000175476073
```

Min operation

RDD API

```
sc.cassandraTable("books_ks", "books").select("book_price").as((c: Float) => c).min
```

Output:

```
res31: Float = 11.33
```

Dataframe API

```
spark
  .read
  .cassandraFormat("books", "books_ks", "")
  .load()
  .select("book_id","book_price")
  .agg(min("book_price"))
  .show
```

Output:

```
+---------------+
|min(book_price)|
+---------------+
|          11.33|
+---------------+
```

SQL

```
select min(book_price) from books_vw;
```

Output:

```
11.33
```

Max operation

RDD API

```
sc.cassandraTable("books_ks", "books").select("book_price").as((c: Float) => c).max
```

Dataframe API

```
spark
  .read
  .cassandraFormat("books", "books_ks", "")
  .load()
  .select("book_price")
  .agg(max("book_price"))
  .show
```

Output:

```
+---------------+
|max(book_price)|
+---------------+
|          22.45|
+---------------+
```

SQL

```
select max(book_price) from books_vw;
```

Output:

```
22.45
```

Sum operation

RDD API

```
sc.cassandraTable("books_ks", "books").select("book_price").as((c: Float) => c).sum
```

Output:

```
res46: Double = 80.08000087738037
```

Dataframe API

```
spark
  .read
  .cassandraFormat("books", "books_ks", "")
  .load()
  .select("book_price")
  .agg(sum("book_price"))
  .show
```

Output:

```
+-----------------+
|  sum(book_price)|
+-----------------+
|80.08000087738037|
+-----------------+
```

SQL

```
select sum(book_price) from books_vw;
```

Output:

```
80.08000087738037
```

Top or comparable operation

RDD API

```
val readCalcTopRDD = sc.cassandraTable("books_ks",
"books").select("book_name","book_price").sortBy(_.getFloat(1), false)
readCalcTopRDD.zipWithIndex.filter(_._2 < 3).collect.foreach(println)
//delivers the first top n items without collecting the rdd to the driver.
```

Output:

```
(CassandraRow{book_name: A sign of four, book_price: 22.45},0)
(CassandraRow{book_name: The adventures of Sherlock Holmes, book_price: 19.83},1)
(CassandraRow{book_name: The memoirs of Sherlock Holmes, book_price: 14.22},2)
readCalcTopRDD: org.apache.spark.rdd.RDD[com.datastax.spark.connector.CassandraRow] = MapPartitionsRDD[430]
at sortBy at command-2371828989676374:1
```

Dataframe API

```
import org.apache.spark.sql.functions._

val readBooksDF = spark.read.format("org.apache.spark.sql.cassandra")
  .options(Map( "table" -> "books", "keyspace" -> "books_ks"))
  .load
  .select("book_name","book_price")
  .orderBy(desc("book_price"))
  .limit(3)

//Explain plan
readBooksDF.explain

//Top
readBooksDF.show
```

Output:

```
== Physical Plan ==
TakeOrderedAndProject(limit=3, orderBy=[book_price#1840 DESC NULLS LAST], output=
[book_name#1839,book_price#1840])
+- *(1) Scan org.apache.spark.sql.cassandra.CassandraSourceRelation@29cd5f58 [book_name#1839,book_price#1840]
PushedFilters: [], ReadSchema: struct<book_name:string,book_price:float>
+--------------------+----------+
|           book_name|book_price|
+--------------------+----------+
|      A sign of four|     22.45|
|The adventures of...|     19.83|
|The memoirs of Sh...|     14.22|
+--------------------+----------+

import org.apache.spark.sql.functions._
readBooksDF: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] = [book_name: string, book_price: float]
```

SQL

```
select book_name,book_price from books_vw order by book_price desc limit 3;
```

Next steps

To perform table copy operations, see:

- Table copy operations

Table copy operations on Azure Cosmos DB Cassandra API from Spark

11/2/2020 • 2 minutes to read • Edit Online

APPLIES TO: Cassandra API

This article describes how to copy data between tables in Azure Cosmos DB Cassandra API from Spark. The commands described in this article can also be used to copy data from Apache Cassandra tables to Azure Cosmos DB Cassandra API tables.

Cassandra API configuration

```
import org.apache.spark.sql.cassandra._
//Spark connector
import com.datastax.spark.connector._
import com.datastax.spark.connector.cql.CassandraConnector

//CosmosDB library for multiple retry
import com.microsoft.azure.cosmosdb.cassandra

//Connection-related
spark.conf.set("spark.cassandra.connection.host","YOUR_ACCOUNT_NAME.cassandra.cosmosdb.azure.com")
spark.conf.set("spark.cassandra.connection.port","10350")
spark.conf.set("spark.cassandra.connection.ssl.enabled","true")
spark.conf.set("spark.cassandra.auth.username","YOUR_ACCOUNT_NAME")
spark.conf.set("spark.cassandra.auth.password","YOUR_ACCOUNT_KEY")
spark.conf.set("spark.cassandra.connection.factory",
"com.microsoft.azure.cosmosdb.cassandra.CosmosDbConnectionFactory")
//Throughput-related...adjust as needed
spark.conf.set("spark.cassandra.output.batch.size.rows", "1")
spark.conf.set("spark.cassandra.connection.connections_per_executor_max", "10")
spark.conf.set("spark.cassandra.output.concurrent.writes", "1000")
spark.conf.set("spark.cassandra.concurrent.reads", "512")
spark.conf.set("spark.cassandra.output.batch.grouping.buffer.size", "1000")
spark.conf.set("spark.cassandra.connection.keep_alive_ms", "600000000")
```

Insert sample data

```
val booksDF = Seq(
   ("b00001", "Arthur Conan Doyle", "A study in scarlet", 1887,11.33),
   ("b00023", "Arthur Conan Doyle", "A sign of four", 1890,22.45),
   ("b01001", "Arthur Conan Doyle", "The adventures of Sherlock Holmes", 1892,19.83),
   ("b00501", "Arthur Conan Doyle", "The memoirs of Sherlock Holmes", 1893,14.22),
   ("b00300", "Arthur Conan Doyle", "The hounds of Baskerville", 1901,12.25)
).toDF("book_id", "book_author", "book_name", "book_pub_year","book_price")

booksDF.write
  .mode("append")
  .format("org.apache.spark.sql.cassandra")
  .options(Map( "table" -> "books", "keyspace" -> "books_ks", "output.consistency.level" -> "ALL", "ttl" ->
"10000000"))
  .save()
```

Copy data between tables

Copy data between tables (destination table exists)

```
//1) Create destination table
val cdbConnector = CassandraConnector(sc)
cdbConnector.withSessionDo(session => session.execute("CREATE TABLE IF NOT EXISTS books_ks.books_copy(book_id
TEXT PRIMARY KEY,book_author TEXT, book_name TEXT,book_pub_year INT,book_price FLOAT) WITH
cosmosdb_provisioned_throughput=4000;"))

//2) Read from one table
val readBooksDF = sqlContext
  .read
  .format("org.apache.spark.sql.cassandra")
  .options(Map( "table" -> "books", "keyspace" -> "books_ks"))
  .load

//3) Save to destination table
readBooksDF.write
  .cassandraFormat("books_copy", "books_ks", "")
  .save()

//4) Validate copy to destination table
sqlContext
  .read
  .format("org.apache.spark.sql.cassandra")
  .options(Map( "table" -> "books_copy", "keyspace" -> "books_ks"))
  .load
  .show
```

Copy data between tables (destination table does not exist)

```
import com.datastax.spark.connector._

//1) Read from source table
val readBooksDF = sqlContext
  .read
  .format("org.apache.spark.sql.cassandra")
  .options(Map( "table" -> "books", "keyspace" -> "books_ks"))
  .load

//2) Creates an empty table in the keyspace based off of source table
val newBooksDF = readBooksDF
newBooksDF.createCassandraTable(
    "books_ks",
    "books_new",
    partitionKeyColumns = Some(Seq("book_id"))
    //clusteringKeyColumns = Some(Seq("some column"))
    )

//3) Saves the data from the source table into the newly created table
newBooksDF.write
  .cassandraFormat("books_new", "books_ks","")
  .save()

//4) Validate table creation and data load
sqlContext
  .read
  .format("org.apache.spark.sql.cassandra")
  .options(Map( "table" -> "books_new", "keyspace" -> "books_ks"))
  .load
  .show
```

The output-

```
+-------+------------------+--------------------+----------+-------------+
|book_id|       book_author|           book_name|book_price|book_pub_year|
+-------+------------------+--------------------+----------+-------------+
| b00300|Arthur Conan Doyle|The hounds of Bas...|     12.25|         1901|
| b00001|Arthur Conan Doyle|  A study in scarlet|     11.33|         1887|
| b00023|Arthur Conan Doyle|      A sign of four|     22.45|         1890|
| b00501|Arthur Conan Doyle|The memoirs of Sh...|     14.22|         1893|
| b01001|Arthur Conan Doyle|The adventures of...|     19.83|         1892|
+-------+------------------+--------------------+----------+-------------+

import com.datastax.spark.connector._
readBooksDF: org.apache.spark.sql.DataFrame = [book_id: string, book_author: string ... 3 more fields]
newBooksDF: org.apache.spark.sql.DataFrame = [book_id: string, book_author: string ... 3 more fields]
```

Next steps

- Get started with creating a Cassandra API account, database, and a table by using a Java application.
- Load sample data to the Cassandra API table by using a Java application.
- Query data from the Cassandra API account by using a Java application.

Manage Azure Cosmos DB Cassandra API resources using Azure Resource Manager templates

12/4/2020 • 5 minutes to read • Edit Online

APPLIES TO: Cassandra API

In this article, you learn how to use Azure Resource Manager templates to help deploy and manage your Azure Cosmos DB accounts, keyspaces, and tables.

This article has examples for Cassandra API accounts only, to find examples for other API type accounts see: use Azure Resource Manager templates with Azure Cosmos DB's API for SQL, Gremlin, MongoDB, Table articles.

IMPORTANT

- Account names are limited to 44 characters, all lowercase.
- To change the throughput values, redeploy the template with updated RU/s.
- When you add or remove locations to an Azure Cosmos account, you can't simultaneously modify other properties. These operations must be done separately.

To create any of the Azure Cosmos DB resources below, copy the following example template into a new json file. You can optionally create a parameters json file to use when deploying multiple instances of the same resource with different names and values. There are many ways to deploy Azure Resource Manager templates including, Azure portal, Azure CLI, Azure PowerShell and GitHub.

Azure Cosmos account for Cassandra with autoscale provisioned throughput

This template creates an Azure Cosmos account in two regions with options for consistency and failover, with a keyspace and table configured for autoscale throughput. This template is also available for one-click deploy from Azure Quickstart Templates Gallery.

Deploy to Azure

```
{
   "$schema": "https://schema.management.azure.com/schemas/2019-04-01/deploymentTemplate.json#",
   "contentVersion": "1.0.0.0",
   "parameters": {
      "accountName": {
         "type": "string",
         "defaultValue": "[concat('cassandra-', uniqueString(resourceGroup().id))]",
         "metadata": {
            "description": "Cosmos DB account name, max length 44 characters"
         }
      },
      "location": {
         "type": "string",
         "defaultValue": "[resourceGroup().location]",
         "metadata": {
            "description": "Location for the Cosmos DB account."
         }
      },
      "primaryRegion": {
         "type": "string",
```

```
         "metadata": {
            "description": "The primary replica region for the Cosmos DB account."
         }
      },
      "secondaryRegion": {
         "type": "string",
         "metadata": {
            "description": "The secondary replica region for the Cosmos DB account."
         }
      },
      "defaultConsistencyLevel": {
         "type": "string",
         "defaultValue": "Session",
         "allowedValues": [
            "Eventual",
            "ConsistentPrefix",
            "Session",
            "BoundedStaleness",
            "Strong"
         ],
         "metadata": {
            "description": "The default consistency level of the Cosmos DB account."
         }
      },
      "maxStalenessPrefix": {
         "type": "int",
         "defaultValue": 100000,
         "minValue": 10,
         "maxValue": 1000000,
         "metadata": {
            "description": "Max stale requests. Required for BoundedStaleness. Valid ranges, Single Region:
10 to 1000000. Multi Region: 100000 to 1000000."
         }
      },
      "maxIntervalInSeconds": {
         "type": "int",
         "defaultValue": 300,
         "minValue": 5,
         "maxValue": 86400,
         "metadata": {
            "description": "Max lag time (seconds). Required for BoundedStaleness. Valid ranges, Single
Region: 5 to 84600. Multi Region: 300 to 86400."
         }
      },
      "automaticFailover": {
         "type": "bool",
         "defaultValue": true,
         "allowedValues": [
            true,
            false
         ],
         "metadata": {
            "description": "Enable automatic failover for regions"
         }
      },
      "keyspaceName": {
         "type": "string",
         "metadata": {
            "description": "The name for the Cassandra Keyspace"
         }
      },
      "tableName": {
         "type": "string",
         "metadata": {
            "description": "The name for the Cassandra table"
         }
      },
      "autoscaleMaxThroughput": {
         "type": "int",
```

```
            "defaultValue": 4000,
            "minValue": 4000,
            "maxValue": 1000000,
            "metadata": {
               "description": "Maximum autoscale throughput for the Cassandra table"
            }
         }
      },
      "variables": {
         "accountName": "[toLower(parameters('accountName'))]",
         "consistencyPolicy": {
            "Eventual": {
               "defaultConsistencyLevel": "Eventual"
            },
            "ConsistentPrefix": {
               "defaultConsistencyLevel": "ConsistentPrefix"
            },
            "Session": {
               "defaultConsistencyLevel": "Session"
            },
            "BoundedStaleness": {
               "defaultConsistencyLevel": "BoundedStaleness",
               "maxStalenessPrefix": "[parameters('maxStalenessPrefix')]",
               "maxIntervalInSeconds": "[parameters('maxIntervalInSeconds')]"
            },
            "Strong": {
               "defaultConsistencyLevel": "Strong"
            }
         },
         "locations": [
            {
               "locationName": "[parameters('primaryRegion')]",
               "failoverPriority": 0,
               "isZoneRedundant": false
            },
            {
               "locationName": "[parameters('secondaryRegion')]",
               "failoverPriority": 1,
               "isZoneRedundant": false
            }
         ]
      },
      "resources": [
         {
            "type": "Microsoft.DocumentDB/databaseAccounts",
            "name": "[variables('accountName')]",
            "apiVersion": "2020-04-01",
            "location": "[parameters('location')]",
            "kind": "GlobalDocumentDB",
            "properties": {
               "capabilities": [ { "name": "EnableCassandra" } ],
               "consistencyPolicy": "[variables('consistencyPolicy')[parameters('defaultConsistencyLevel')]]",
               "locations": "[variables('locations')]",
               "databaseAccountOfferType": "Standard",
               "enableAutomaticFailover": "[parameters('automaticFailover')]"
            }
         },
         {
            "type": "Microsoft.DocumentDB/databaseAccounts/cassandraKeyspaces",
            "name": "[concat(variables('accountName'), '/', parameters('keyspaceName'))]",
            "apiVersion": "2020-04-01",
            "dependsOn": [
               "[resourceId('Microsoft.DocumentDB/databaseAccounts/', variables('accountName'))]"
            ],
            "properties": {
               "resource": {
                  "id": "[parameters('keyspaceName')]"
               }
            }
```

```
            }
        },
        {
            "type": "Microsoft.DocumentDb/databaseAccounts/cassandraKeyspaces/tables",
            "name": "[concat(variables('accountName'), '/', parameters('keyspaceName'), '/',
parameters('tableName'))]",
            "apiVersion": "2020-04-01",
            "dependsOn": [
                "[resourceId('Microsoft.DocumentDB/databaseAccounts/cassandraKeyspaces',
variables('accountName'), parameters('keyspaceName'))]"
            ],
            "properties": {
                "resource": {
                    "id": "[parameters('tableName')]",
                    "schema": {
                        "columns": [
                            {
                                "name": "loadid",
                                "type": "uuid"
                            },
                            {
                                "name": "machine",
                                "type": "uuid"
                            },
                            {
                                "name": "cpu",
                                "type": "int"
                            },
                            {
                                "name": "mtime",
                                "type": "int"
                            },
                            {
                                "name": "load",
                                "type": "float"
                            }
                        ],
                        "partitionKeys": [
                            { "name": "machine" },
                            { "name": "cpu" },
                            { "name": "mtime" }
                        ],
                        "clusterKeys": [
                            {
                                "name": "loadid",
                                "orderBy": "asc"
                            }
                        ]
                    }
                },
                "options": {
                    "autoscaleSettings": {
                        "maxThroughput": "[parameters('autoscaleMaxThroughput')]"
                    }
                }
            }
        }
    ]
}
```

Azure Cosmos account for Cassandra with standard provisioned throughput

This template creates an Azure Cosmos account in two regions with options for consistency and failover, with a keyspace and table configured for standard throughput. This template is also available for one-click deploy from Azure Quickstart Templates Gallery.

Deploy to Azure

```
{
    "$schema": "https://schema.management.azure.com/schemas/2019-04-01/deploymentTemplate.json#",
    "contentVersion": "1.0.0.0",
    "parameters": {
        "accountName": {
            "type": "string",
            "defaultValue": "[concat('sql-', uniqueString(resourceGroup().id))]",
            "metadata": {
                "description": "Cosmos DB account name, max length 44 characters"
            }
        },
        "location": {
            "type": "string",
            "defaultValue": "[resourceGroup().location]",
            "metadata": {
                "description": "Location for the Cosmos DB account."
            }
        },
        "primaryRegion":{
            "type":"string",
            "metadata": {
                "description": "The primary replica region for the Cosmos DB account."
            }
        },
        "secondaryRegion":{
            "type":"string",
            "metadata": {
              "description": "The secondary replica region for the Cosmos DB account."
          }
        },
        "defaultConsistencyLevel": {
            "type": "string",
            "defaultValue": "Session",
            "allowedValues": [ "Eventual", "ConsistentPrefix", "Session", "BoundedStaleness", "Strong" ],
            "metadata": {
                "description": "The default consistency level of the Cosmos DB account."
            }
        },
        "maxStalenessPrefix": {
            "type": "int",
            "defaultValue": 100000,
            "minValue": 10,
            "maxValue": 2147483647,
            "metadata": {
                "description": "Max stale requests. Required for BoundedStaleness. Valid ranges, Single
Region: 10 to 1000000. Multi Region: 100000 to 1000000."
            }
        },
        "maxIntervalInSeconds": {
            "type": "int",
            "defaultValue": 300,
            "minValue": 5,
            "maxValue": 86400,
            "metadata": {
                "description": "Max lag time (seconds). Required for BoundedStaleness. Valid ranges, Single
Region: 5 to 84600. Multi Region: 300 to 86400."
            }
        },
        "automaticFailover": {
            "type": "bool",
            "defaultValue": true,
            "allowedValues": [ true, false ],
            "metadata": {
                "description": "Enable automatic failover for regions"
```

```
            }
        },
        "keyspaceName": {
            "type": "string",
            "metadata": {
                "description": "The name for the Cassandra Keyspace"
            }
        },
        "tableName": {
            "type": "string",
            "metadata": {
                "description": "The name for the first Cassandra table"
            }
        },
        "throughput": {
            "type": "int",
            "defaultValue": 400,
            "minValue": 400,
            "maxValue": 1000000,
            "metadata": {
                "description": "The throughput for both Cassandra tables"
            }
        }
    },
    "variables": {
        "accountName": "[toLower(parameters('accountName'))]",
        "consistencyPolicy": {
            "Eventual": {
                "defaultConsistencyLevel": "Eventual"
            },
            "ConsistentPrefix": {
                "defaultConsistencyLevel": "ConsistentPrefix"
            },
            "Session": {
                "defaultConsistencyLevel": "Session"
            },
            "BoundedStaleness": {
                "defaultConsistencyLevel": "BoundedStaleness",
                "maxStalenessPrefix": "[parameters('maxStalenessPrefix')]",
                "maxIntervalInSeconds": "[parameters('maxIntervalInSeconds')]"
            },
            "Strong": {
                "defaultConsistencyLevel": "Strong"
            }
        },
        "locations":
        [
            {
                "locationName": "[parameters('primaryRegion')]",
                "failoverPriority": 0,
                "isZoneRedundant": false
            },
            {
                "locationName": "[parameters('secondaryRegion')]",
                "failoverPriority": 1,
                "isZoneRedundant": false
            }
        ]
    },
    "resources":
    [
        {
            "type": "Microsoft.DocumentDB/databaseAccounts",
            "name": "[variables('accountName')]",
            "apiVersion": "2020-03-01",
            "location": "[parameters('location')]",
            "kind": "GlobalDocumentDB",
            "properties": {
                "capabilities": [{ "name": "EnableCassandra" }],
```

```
                "consistencyPolicy": "[variables('consistencyPolicy')
[parameters('defaultConsistencyLevel')]]",
                "locations": "[variables('locations')]",
                "databaseAccountOfferType": "Standard",
                "enableAutomaticFailover": "[parameters('automaticFailover')]"
            }
        },
        {
            "type": "Microsoft.DocumentDB/databaseAccounts/cassandraKeyspaces",
            "name": "[concat(variables('accountName'), '/', parameters('keyspaceName'))]",
            "apiVersion": "2020-03-01",
            "dependsOn": [ "[resourceId('Microsoft.DocumentDB/databaseAccounts/', variables('accountName'))]"
],
            "properties":{
                "resource":{
                    "id": "[parameters('keyspaceName')]"
                }
            }
        },
        {
            "type": "Microsoft.DocumentDb/databaseAccounts/cassandraKeyspaces/tables",
            "name": "[concat(variables('accountName'), '/', parameters('keyspaceName'), '/',
parameters('tableName'))]",
            "apiVersion": "2020-03-01",
            "dependsOn": [ "[resourceId('Microsoft.DocumentDB/databaseAccounts/cassandraKeyspaces',
variables('accountName'), parameters('keyspaceName'))]" ],
            "properties":
            {
                "resource":{
                    "id":  "[parameters('tableName')]",
                    "schema": {
                        "columns": [
                            { "name": "loadid", "type": "uuid" },
                            { "name": "machine", "type": "uuid" },
                            { "name": "cpu", "type": "int" },
                            { "name": "mtime", "type": "int" },
                            { "name": "load", "type": "float" }
                        ],
                        "partitionKeys": [
                            { "name": "machine" },
                            { "name": "cpu" },
                            { "name": "mtime" }
                        ],
                        "clusterKeys": [
                            { "name": "loadid", "orderBy": "asc" }
                        ]
                    },
                    "options": { "throughput": "[parameters('throughput')]" }
                }
            }
        }
    ]
}
```

Next steps

Here are some additional resources:

- Azure Resource Manager documentation
- Azure Cosmos DB resource provider schema
- Azure Cosmos DB Quickstart templates
- Troubleshoot common Azure Resource Manager deployment errors

Azure PowerShell samples for Azure Cosmos DB Cassandra API

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: Cassandra API

The following table includes links to commonly used Azure PowerShell scripts for Azure Cosmos DB. Use the links on the right to navigate to API specific samples. Common samples are the same across all APIs. Reference pages for all Azure Cosmos DB PowerShell cmdlets are available in the Azure PowerShell Reference. Please check for updates to `Az.CosmosDB` regularly. You can also fork these PowerShell samples for Cosmos DB from our GitHub repository, Cosmos DB PowerShell Samples on GitHub.

Common Samples

TASK	DESCRIPTION
Update an account	Update a Cosmos DB account's default consistency level.
Update an account's regions	Update a Cosmos DB account's regions.
Change failover priority or trigger failover	Change the regional failover priority of an Azure Cosmos account or trigger a manual failover.
Account keys or connection strings	Get primary and secondary keys, connection strings or regenerate an account key of an Azure Cosmos DB account.
Create a Cosmos Account with IP Firewall	Create an Azure Cosmos DB account with IP Firewall enabled.

Cassandra API Samples

TASK	DESCRIPTION
Create an account, keyspace and table	Creates an Azure Cosmos account, keyspace and table.
Create an account, keyspace and table with autoscale	Creates an Azure Cosmos account, keyspace and table with autoscale.
List or get keyspaces or tables	List or get keyspaces or tables.
Throughput operations	Throughput operations for a keyspace or table including get, update and migrate between autoscale and standard throughput.
Lock resources from deletion	Prevent resources from being deleted with resource locks.

Azure CLI samples for Azure Cosmos DB Cassandra API

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: Cassandra API

The following table includes links to sample Azure CLI scripts for Azure Cosmos DB. Use the links on the right to navigate to API specific samples. Common samples are the same across all APIs. Reference pages for all Azure Cosmos DB CLI commands are available in the Azure CLI Reference. Azure Cosmos DB CLI script samples can also be found in the Azure Cosmos DB CLI GitHub Repository.

These samples require Azure CLI version 2.12.1 or later. Run `az --version` to find the version. If you need to install or upgrade, see Install Azure CLI

Common Samples

These samples apply to all Azure Cosmos DB APIs

TASK	DESCRIPTION
Add or failover regions	Add a region, change failover priority, trigger a manual failover.
Account keys and connection strings	List account keys, read-only keys, regenerate keys and list connection strings.
Secure with IP firewall	Create a Cosmos account with IP firewall configured.
Secure new account with service endpoints	Create a Cosmos account and secure with service-endpoints.
Secure existing account with service endpoints	Update a Cosmos account to secure with service-endpoints when the subnet is eventually configured.

Cassandra API Samples

TASK	DESCRIPTION
Create an Azure Cosmos account, keyspace and table	Creates an Azure Cosmos DB account, keyspace, and table for Cassandra API.
Create an Azure Cosmos account, keyspace and table with autoscale	Creates an Azure Cosmos DB account, keyspace, and table with autoscale for Cassandra API.
Throughput operations	Read, update and migrate between autoscale and standard throughput on a keyspace and table.
Lock resources from deletion	Prevent resources from being deleted with resource locks.

Azure Cosmos DB's API for MongoDB

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: Azure Cosmos DB API for MongoDB

Azure Cosmos DB is Microsoft's globally distributed, multi-model database service for mission-critical applications. Azure Cosmos DB provides turn-key global distribution, elastic scaling of throughput and storage worldwide, single-digit millisecond latencies at the 99th percentile, and guaranteed high availability, all backed by industry-leading SLAs. Azure Cosmos DB automatically indexes data without requiring you to deal with schema and index management. It is multi-model and supports document, key-value, graph, and columnar data models. Azure Cosmos DB service implements wire protocols for common NoSQL APIs including Cassandra, MongoDB, Gremlin, and Azure Table Storage. This allows you to use your familiar NoSQL client drivers and tools to interact with your Cosmos database.

> **NOTE**
>
> The serverless capacity mode is now available on Azure Cosmos DB's API for MongoDB.

Wire protocol compatibility

Azure Cosmos DB implements the wire protocol for MongoDB. This implementation allows transparent compatibility with native MongoDB client SDKs, drivers, and tools. Azure Cosmos DB does host the MongoDB database engine. The details of the supported features by MongoDB can be found here:

- Azure Cosmos DB's API for Mongo DB engine version 3.6
- Azure Cosmos DB's API for Mongo DB engine version 3.2

By default, new accounts created using Azure Cosmos DB's API for MongoDB are compatible with version 3.6 of the MongoDB wire protocol. Any MongoDB client driver that understands this protocol version should be able to natively connect to Cosmos DB.

Key benefits

The key benefits of Cosmos DB as a fully managed, globally distributed, database as a service are described here. Additionally, by natively implementing wire protocols of popular NoSQL APIs, Cosmos DB provides the following benefits:

- Easily migrate your application to Cosmos DB while preserving significant portions of your application logic.
- Keep your application portable and continue to remain cloud vendor-agnostic.
- Get industry leading, financially backed SLAs for the common NoSQL APIs powered by Cosmos DB.

- Elastically scale the provisioned throughput and storage for your Cosmos databases based on your need and pay only for the throughput and storage you need. This leads to significant cost savings.
- Turnkey, global distribution with multi-region writes replication.

Cosmos DB's API for MongoDB

Follow the quickstarts to create an Azure Cosmos account and migrate your existing MongoDB application to use Azure Cosmos DB, or build a new one:

- Migrate an existing MongoDB Node.js web app.
- Build a web app using Azure Cosmos DB's API for MongoDB and .NET SDK
- Build a console app using Azure Cosmos DB's API for MongoDB and Java SDK

Next steps

Here are a few pointers to get you started:

- Follow the Connect a MongoDB application to Azure Cosmos DB tutorial to learn how to get your account connection string information.
- Follow the Use Studio 3T with Azure Cosmos DB tutorial to learn how to create a connection between your Cosmos database and MongoDB app in Studio 3T.
- Follow the Import MongoDB data into Azure Cosmos DB tutorial to import your data to a Cosmos database.
- Connect to a Cosmos account using Robo 3T.
- Learn how to Configure read preferences for globally distributed apps.
- Find the solutions to commonly found errors in our Troubleshooting guide

Note: This article describes a feature of Azure Cosmos DB that provides wire protocol compatibility with MongoDB databases. Microsoft does not run MongoDB databases to provide this service. Azure Cosmos DB is not affiliated with MongoDB, Inc.

Quickstart: Build a .NET web app using Azure Cosmos DB's API for MongoDB

12/4/2020 • 6 minutes to read • Edit Online

APPLIES TO: Azure Cosmos DB API for MongoDB

Azure Cosmos DB is Microsoft's fast NoSQL database with open APIs for any scale. You can quickly create and query document, key/value and graph databases, all of which benefit from the global distribution and horizontal scale capabilities at the core of Cosmos DB.

This quickstart demonstrates how to create a Cosmos account with Azure Cosmos DB's API for MongoDB. You'll then build and deploy a tasks list web app built using the MongoDB .NET driver.

Prerequisites to run the sample app

To run the sample, you'll need Visual Studio and a valid Azure Cosmos DB account.

If you don't already have Visual Studio, download Visual Studio 2019 Community Edition with the **ASP.NET and web development** workload installed with setup.

If you don't have an Azure subscription, create a free account before you begin.

Create a database account

1. In a new browser window, sign in to the Azure portal.
2. In the left menu, select **Create a resource**.

3. On the **New** page, select **Databases** > **Azure Cosmos DB**.

4. On the **Create Azure Cosmos DB Account** page, enter the settings for the new Azure Cosmos DB account.

SETTING	VALUE	DESCRIPTION
Subscription	Your subscription	Select the Azure subscription that you want to use for this Azure Cosmos DB account.
Resource Group	Create new Then enter the same name as Account Name	Select **Create new**. Then enter a new resource group name for your account. For simplicity, use the same name as your Azure Cosmos DB account name.
Account Name	Enter a unique name	Enter a unique name to identify your Azure Cosmos DB account. Your account URI will be *mongo.cosmos.azure.com* appended to your unique account name. The account name can use only lowercase letters, numbers, and hyphens (-), and must be between 3 and 31 characters long.

SETTING	VALUE	DESCRIPTION
API	Azure Cosmos DB for Mongo DB API	The API determines the type of account to create. Azure Cosmos DB provides five APIs: Core (SQL) for document databases, Gremlin for graph databases, Azure Cosmos DB for Mongo DB API for document databases, Azure Table, and Cassandra. Currently, you must create a separate account for each API. Select **Azure Cosmos DB for Mongo DB API** because in this quickstart you are creating a collection that works with MongoDB. Learn more about Azure Cosmos DB for MongoDB API.
Location	Select the region closest to your users	Select a geographic location to host your Azure Cosmos DB account. Use the location that's closest to your users to give them the fastest access to the data.
Capacity mode	Provisioned throughput or Serverless	Select **Provisioned throughput** to create an account in provisioned throughput mode. Select **Serverless** to create an account in serverless mode. **Note**: Only MongoDB API version 3.6 is supported by serverless accounts. Choosing 3.2 as the version will force the account in provisioned throughput mode.

Select **Review+Create**. You can skip the **Network** and **Tags** section.

Home > New >

Create Azure Cosmos DB Account

Basics | Networking | Backup Policy | Encryption | Tags | Review + create

Azure Cosmos DB is a globally distributed, multi-model, fully managed database service. Try it for free, for 30 days with unlimited renewals. Go to production starting at $24/month per database, multiple containers included. Learn more

Project Details

Select the subscription to manage deployed resources and costs. Use resource groups like folders to organize and manage all your resources.

Subscription * — Visual Studio Enterprise

Resource Group * — Create new

Instance Details

Account Name * — Enter account name

API * — Azure Cosmos DB for MongoDB API

Notebooks (Preview) — On / Off

Location * — (US) West US

Capacity mode — Provisioned throughput / Serverless (preview)

Learn more about capacity mode

With Azure Cosmos DB free tier, you will get 400 RU/s and 5 GB of storage for free in an account. You can enable free tier on up to one account per subscription. Estimated $24/month discount per account.

Apply Free Tier Discount — Apply / Do Not Apply

Account Type — Production / Non-Production

Version — 3.6

Geo-Redundancy — Enable / Disable

Multi-region Writes — Enable / Disable

*Up to 33% off multi-region writes is available to qualifying new accounts only. Offer limited to accounts with both account locations and geo-redundancy, and applies only to multi-region writes in those same regions. Both Geo-Redundancy and Multi-region Writes must be enabled in account settings. Actual discount will vary based on number of qualifying regions selected.

Review + create | Previous | Next: Networking

5. The account creation takes a few minutes. Wait for the portal to display the **Congratulations! Your Azure Cosmos DB for Mongo DB API account is ready** page.

The sample described in this article is compatible with MongoDB.Driver version 2.6.1.

Clone the sample app

First, download the sample app from GitHub.

1. Open a command prompt, create a new folder named git-samples, then close the command prompt.

```
mkdir "C:\git-samples"
```

2. Open a git terminal window, such as git bash, and use the `cd` command to change to the new folder to install the sample app.

```
cd "C:\git-samples"
```

3. Run the following command to clone the sample repository. This command creates a copy of the sample app on your computer.

```
git clone https://github.com/Azure-Samples/azure-cosmos-db-mongodb-dotnet-getting-started.git
```

If you don't wish to use git, you can also download the project as a ZIP file.

Review the code

This step is optional. If you're interested in learning how the database resources are created in the code, you can review the following snippets. Otherwise, you can skip ahead to Update your connection string.

The following snippets are all taken from the Dal.cs file in the DAL directory.

- Initialize the client.

```
    MongoClientSettings settings = new MongoClientSettings();
    settings.Server = new MongoServerAddress(host, 10255);
    settings.UseSsl = true;
    settings.SslSettings = new SslSettings();
    settings.SslSettings.EnabledSslProtocols = SslProtocols.Tls12;

    MongoIdentity identity = new MongoInternalIdentity(dbName, userName);
    MongoIdentityEvidence evidence = new PasswordEvidence(password);

    settings.Credential = new MongoCredential("SCRAM-SHA-1", identity, evidence);

    MongoClient client = new MongoClient(settings);
```

- Retrieve the database and the collection.

```
private string dbName = "Tasks";
private string collectionName = "TasksList";

var database = client.GetDatabase(dbName);
var todoTaskCollection = database.GetCollection<MyTask>(collectionName);
```

- Retrieve all documents.

```
collection.Find(new BsonDocument()).ToList();
```

Create a task and insert it into the collection

```
public void CreateTask(MyTask task)
{
    var collection = GetTasksCollectionForEdit();
    try
    {
        collection.InsertOne(task);
    }
    catch (MongoCommandException ex)
    {
        string msg = ex.Message;
    }
}
```

Similarly, you can update and delete documents by using the collection.UpdateOne() and collection.DeleteOne() methods.

Update your connection string

Now go back to the Azure portal to get your connection string information and copy it into the app.

1. In the Azure portal, in your Cosmos account, in the left navigation click **Connection String**, and then click **Read-write Keys**. You'll use the copy buttons on the right side of the screen to copy the Username, Password, and Host into the Dal.cs file in the next step.

2. Open the **Dal.cs** file in the **DAL** directory.

3. Copy your **username** value from the portal (using the copy button) and make it the value of the

username in your **Dal.cs** file.

4. Then copy your **host** value from the portal and make it the value of the **host** in your **Dal.cs** file.

5. Finally copy your **password** value from the portal and make it the value of the **password** in your **Dal.cs** file.

You've now updated your app with all the info it needs to communicate with Cosmos DB.

Run the web app

1. In Visual Studio, right-click on the project in **Solution Explorer** and then click **Manage NuGet Packages**.

2. In the NuGet **Browse** box, type *MongoDB.Driver*.

3. From the results, install the **MongoDB.Driver** library. This installs the MongoDB.Driver package as well as all dependencies.

4. Click CTRL + F5 to run the application. Your app displays in your browser.

5. Click **Create** in the browser and create a few new tasks in your task list app.

Review SLAs in the Azure portal

The Azure portal monitors your Cosmos DB account throughput, storage, availability, latency, and consistency. Charts for metrics associated with an Azure Cosmos DB Service Level Agreement (SLA) show the SLA value compared to actual performance. This suite of metrics makes monitoring your SLAs transparent.

To review metrics and SLAs:

1. Select **Metrics** in your Cosmos DB account's navigation menu.

2. Select a tab such as **Latency**, and select a timeframe on the right. Compare the **Actual** and **SLA** lines on the charts.

3. Review the metrics on the other tabs.

Clean up resources

When you're done with your app and Azure Cosmos DB account, you can delete the Azure resources you created so you don't incur more charges. To delete the resources:

1. In the Azure portal Search bar, search for and select **Resource groups**.

2. From the list, select the resource group you created for this quickstart.

3. On the resource group **Overview** page, select **Delete resource group**.

4. In the next window, enter the name of the resource group to delete, and then select **Delete**.

Next steps

In this quickstart, you've learned how to create a Cosmos account, create a collection and run a console app. You can now import additional data to your Cosmos database.

Import MongoDB data into Azure Cosmos DB

Quickstart: Create a console app with Java and the MongoDB API in Azure Cosmos DB

12/4/2020 • 6 minutes to read • Edit Online

APPLIES TO: Azure Cosmos DB API for MongoDB

In this quickstart, you create and manage an Azure Cosmos DB for MongoDB API account from the Azure portal, and add data by using a Java SDK app cloned from GitHub. Azure Cosmos DB is a multi-model database service that lets you quickly create and query document, table, key-value, and graph databases with global distribution and horizontal scale capabilities.

Prerequisites

- An Azure account with an active subscription. Create one for free. Or try Azure Cosmos DB for free without an Azure subscription. You can also use the Azure Cosmos DB Emulator with the connection string

  ```
  .mongodb://localhost:C2y6yDjf5/R+ob0N8A7Cgv30VRDJIWEHLM+4QDU5DE2nQ9nDuVTqobD4b8mGGyPMbIZnqyMsEcaGQy67XIw/Jw==@localhost:10255/admin?ssl=true
  ```

  .
- Java Development Kit (JDK) version 8.
- Maven. Or run `apt-get install maven` to install Maven.
- Git.

Create a database account

1. In a new browser window, sign in to the Azure portal.
2. In the left menu, select **Create a resource**.

3. On the **New** page, select **Databases** > **Azure Cosmos DB**.

4. On the **Create Azure Cosmos DB Account** page, enter the settings for the new Azure Cosmos DB account.

SETTING	VALUE	DESCRIPTION
Subscription	Your subscription	Select the Azure subscription that you want to use for this Azure Cosmos DB account.
Resource Group	Create new Then enter the same name as Account Name	Select **Create new**. Then enter a new resource group name for your account. For simplicity, use the same name as your Azure Cosmos DB account name.
Account Name	Enter a unique name	Enter a unique name to identify your Azure Cosmos DB account. Your account URI will be *mongo.cosmos.azure.com* appended to your unique account name. The account name can use only lowercase letters, numbers, and hyphens (-), and must be between 3 and 31 characters long.

SETTING	VALUE	DESCRIPTION
API	Azure Cosmos DB for Mongo DB API	The API determines the type of account to create. Azure Cosmos DB provides five APIs: Core (SQL) for document databases, Gremlin for graph databases, Azure Cosmos DB for Mongo DB API for document databases, Azure Table, and Cassandra. Currently, you must create a separate account for each API. Select **Azure Cosmos DB for Mongo DB API** because in this quickstart you are creating a collection that works with MongoDB. Learn more about Azure Cosmos DB for MongoDB API.
Location	Select the region closest to your users	Select a geographic location to host your Azure Cosmos DB account. Use the location that's closest to your users to give them the fastest access to the data.
Capacity mode	Provisioned throughput or Serverless	Select **Provisioned throughput** to create an account in provisioned throughput mode. Select **Serverless** to create an account in serverless mode. **Note**: Only MongoDB API version 3.6 is supported by serverless accounts. Choosing 3.2 as the version will force the account in provisioned throughput mode.

Select **Review+Create**. You can skip the **Network** and **Tags** section.

Home > New >

Create Azure Cosmos DB Account

Basics | Networking | Backup Policy | Encryption | Tags | Review + create

Azure Cosmos DB is a globally distributed, multi-model, fully managed database service. Try it for free, for 30 days with unlimited renewals. Go to production starting at $24/month per database, multiple containers included. Learn more

Project Details

Select the subscription to manage deployed resources and costs. Use resource groups like folders to organize and manage all your resources.

Subscription * : Visual Studio Enterprise

Resource Group * :

Create new

Instance Details

Account Name * : Enter account name

API * : Azure Cosmos DB for MongoDB API

Notebooks (Preview) : On | Off

Location * : (US) West US

Capacity mode : Provisioned throughput | Serverless (preview)

Learn more about capacity mode

With Azure Cosmos DB free tier, you will get 400 RU/s and 5 GB of storage for free in an account. You can enable free tier on up to one account per subscription. Estimated $24/month discount per account.

Apply Free Tier Discount : Apply | Do Not Apply

Account Type : Production | Non-Production

Version : 3.6

Geo-Redundancy : Enable | Disable

Multi-region Writes : Enable | Disable

*Up to 33% off multi-region writes is available to qualifying new accounts only. Offer limited to accounts with both account locations and geo-redundancy, and applies only to multi-region writes in those same regions. Both Geo-Redundancy and Multi-region Writes must be enabled in account settings. Actual discount will vary based on number of qualifying regions selected.

Review + create | Previous | Next: Networking

5. The account creation takes a few minutes. Wait for the portal to display the **Congratulations! Your Azure Cosmos DB for Mongo DB API account is ready** page.

Add a collection

Name your new database **db**, and your new collection **coll**.

You can now use the Data Explorer tool in the Azure portal to create an Azure Cosmos DB's API for MongoDB database and container.

1. Select **Data Explorer** > **New Container**.

   The **Add Container** area is displayed on the far right, you may need to scroll right to see it.

2. In the **Add container** page, enter the settings for the new container.

SETTING	SUGGESTED VALUE	DESCRIPTION
Database ID	db	Enter *db* as the name for the new database. Database names must contain from 1 through 255 characters, and they cannot contain `/, \\, #, ?`, or a trailing space. Check the **Provision database throughput** option, it allows you to share the throughput provisioned to the database across all the containers within the database. This option also helps with cost savings.
Throughput	400	Leave the throughput at 400 request units per second (RU/s). If you want to reduce latency, you can scale up the throughput later. You can also choose Autoscale mode, which will give you a range of RU/s that will dynamically increase and decrease as needed.
Collection ID	coll	Enter `coll` as the name for your new container. Container IDs have the same character requirements as database names.
Storage capacity	Fixed (10GB)	Enter *Fixed (10GB)* for this application. If you select *Unlimited*, you will have to create a `Shard Key`, which all items inserted will require.
Shard key	/_id	The sample described in this article does not use a Shard Key, so setting it to */_id* will use the automatically generated ID field as the shard key. Learn more about sharding, also known as partitioning, in Partitioning in Azure Cosmos DB

Select **OK**. The Data Explorer displays the new database and container.

Clone the sample application

Now let's clone an app from GitHub, set the connection string, and run it. You'll see how easy it is to work with data programmatically.

1. Open a command prompt, create a new folder named git-samples, then close the command prompt.

```
md "C:\git-samples"
```

2. Open a git terminal window, such as git bash, and use the `cd` command to change to the new folder to install the sample app.

```
cd "C:\git-samples"
```

3. Run the following command to clone the sample repository. This command creates a copy of the sample app on your computer.

```
git clone https://github.com/Azure-Samples/azure-cosmos-db-mongodb-java-getting-started.git
```

4. Then open the code in your favorite editor.

Review the code

This step is optional. If you're interested in learning how the database resources are created in the code, you can review the following snippets. Otherwise, you can skip ahead to Update your connection string.

The following snippets are all taken from the *Program.java* file.

This console app uses the MongoDB Java driver.

- The DocumentClient is initialized.

```
MongoClientURI uri = new MongoClientURI("FILLME");`

MongoClient mongoClient = new MongoClient(uri);
```

- A new database and collection are created.

```
MongoDatabase database = mongoClient.getDatabase("db");

MongoCollection<Document> collection = database.getCollection("coll");
```

- Some documents are inserted using `MongoCollection.insertOne`

```
Document document = new Document("fruit", "apple")
collection.insertOne(document);
```

- Some queries are performed using `MongoCollection.find`

```
Document queryResult = collection.find(Filters.eq("fruit", "apple")).first();
System.out.println(queryResult.toJson());
```

Update your connection string

Now go back to the Azure portal to get your connection string information and copy it into the app.

1. From your Azure Cosmos DB account, select **Quick Start**, select **Java**, then copy the connection string to your clipboard.

2. Open the *Program.java* file, replace the argument to the MongoClientURI constructor with the connection string. You've now updated your app with all the info it needs to communicate with Azure Cosmos DB.

Run the console app

1. Run `mvn package` in a terminal to install required npm modules

2. Run `mvn exec:java -D exec.mainClass=GetStarted.Program` in a terminal to start your Java application.

You can now use Robomongo / Studio 3T to query, modify, and work with this new data.

Review SLAs in the Azure portal

The Azure portal monitors your Cosmos DB account throughput, storage, availability, latency, and consistency. Charts for metrics associated with an Azure Cosmos DB Service Level Agreement (SLA) show the SLA value compared to actual performance. This suite of metrics makes monitoring your SLAs transparent.

To review metrics and SLAs:

1. Select **Metrics** in your Cosmos DB account's navigation menu.

2. Select a tab such as **Latency**, and select a timeframe on the right. Compare the **Actual** and **SLA** lines on the charts.

3. Review the metrics on the other tabs.

Clean up resources

When you're done with your app and Azure Cosmos DB account, you can delete the Azure resources you created so you don't incur more charges. To delete the resources:

1. In the Azure portal Search bar, search for and select **Resource groups**.

2. From the list, select the resource group you created for this quickstart.

3. On the resource group **Overview** page, select **Delete resource group**.

4. In the next window, enter the name of the resource group to delete, and then select **Delete**.

Next steps

In this quickstart, you learned how to create an Azure Cosmos DB API for Mongo DB account, add a database and container using Data Explorer, and add data using a Java console app. You can now import additional data to your Cosmos database.

Import MongoDB data into Azure Cosmos DB

Quickstart: Migrate an existing MongoDB Node.js web app to Azure Cosmos DB

12/4/2020 • 6 minutes to read • Edit Online

APPLIES TO: Azure Cosmos DB API for MongoDB

In this quickstart, you create and manage an Azure Cosmos DB for Mongo DB API account by using the Azure Cloud Shell, and with a MEAN (MongoDB, Express, Angular, and Node.js) app cloned from GitHub. Azure Cosmos DB is a multi-model database service that lets you quickly create and query document, table, key-value, and graph databases with global distribution and horizontal scale capabilities.

Prerequisites

- An Azure account with an active subscription. If you don't have an Azure subscription, create a free account before you begin. Or try Azure Cosmos DB for free without an Azure subscription. You can also use the Azure Cosmos DB Emulator with the connection string

  ```
  .mongodb://localhost:C2y6yDjf5/R+ob0N8A7Cgv30VRDJIWEHLM+4QDU5DE2nQ9nDuVTqobD4b8mGGyPMbIZnqyMsEcaGQy67XIw/Jw==@localhost:10255/admin?ssl=true
  ```

  .

- Node.js, and a working knowledge of Node.js.
- Git.
- Use Azure Cloud Shell using the bash environment.

  Launch Cloud Shell

- If you prefer, install the Azure CLI to run CLI reference commands.
  - If you're using a local install, sign in with Azure CLI by using the az login command. To finish the authentication process, follow the steps displayed in your terminal. See Sign in with Azure CLI for additional sign-in options.
  - When you're prompted, install Azure CLI extensions on first use. For more information about extensions, see Use extensions with Azure CLI.
  - Run az version to find the version and dependent libraries that are installed. To upgrade to the latest version, run az upgrade.
- This article requires version 2.0 or later of the Azure CLI. If using Azure Cloud Shell, the latest version is already installed.

Clone the sample application

Run the following commands to clone the sample repository. This sample repository contains the default MEAN.js application.

1. Open a command prompt, create a new folder named git-samples, then close the command prompt.

   ```
   mkdir "C:\git-samples"
   ```

2. Open a git terminal window, such as git bash, and use the `cd` command to change to the new folder to install the sample app.

   ```
   cd "C:\git-samples"
   ```

3. Run the following command to clone the sample repository. This command creates a copy of the sample app on your computer.

   ```
   git clone https://github.com/prashanthmadi/mean
   ```

Run the application

This MongoDB app written in Node.js connects to your Azure Cosmos DB database, which supports MongoDB client. In other words, it is transparent to the application that the data is stored in an Azure Cosmos DB database.

Install the required packages and start the application.

```
cd mean
npm install
npm start
```

The application will try to connect to a MongoDB source and fail, go ahead and exit the application when the output returns " [MongoError: connect ECONNREFUSED 127.0.0.1:27017]".

Sign in to Azure

If you are using an installed Azure CLI, sign in to your Azure subscription with the az login command and follow the on-screen directions. You can skip this step if you're using the Azure Cloud Shell.

```
az login
```

Add the Azure Cosmos DB module

If you are using an installed Azure CLI, check to see if the `cosmosdb` component is already installed by running the `az` command. If `cosmosdb` is in the list of base commands, proceed to the next command. You can skip this step if you're using the Azure Cloud Shell.

If `cosmosdb` is not in the list of base commands, reinstall Azure CLI.

Create a resource group

Create a resource group with the az group create. An Azure resource group is a logical container into which Azure resources like web apps, databases and storage accounts are deployed and managed.

The following example creates a resource group in the West Europe region. Choose a unique name for the resource group.

If you are using Azure Cloud Shell, select **Try It**, follow the onscreen prompts to login, then copy the command into the command prompt.

```
az group create --name myResourceGroup --location "West Europe"
```

Create an Azure Cosmos DB account

Create a Cosmos account with the az cosmosdb create command.

In the following command, please substitute your own unique Cosmos account name where you see the `<cosmosdb-name>` placeholder. This unique name will be used as part of your Cosmos DB endpoint (`https://<cosmosdb-name>.documents.azure.com/`), so the name needs to be unique across all Cosmos accounts in Azure.

```
az cosmosdb create --name <cosmosdb-name> --resource-group myResourceGroup --kind MongoDB
```

The `--kind MongoDB` parameter enables MongoDB client connections.

When the Azure Cosmos DB account is created, the Azure CLI shows information similar to the following example.

> **NOTE**
>
> This example uses JSON as the Azure CLI output format, which is the default. To use another output format, see Output formats for Azure CLI commands.

```
{
  "databaseAccountOfferType": "Standard",
  "documentEndpoint": "https://<cosmosdb-name>.documents.azure.com:443/",
  "id": "/subscriptions/00000000-0000-0000-0000-000000000000/resourceGroups/myResourceGroup/providers/Microsoft.Document
DB/databaseAccounts/<cosmosdb-name>",
  "kind": "MongoDB",
  "location": "West Europe",
  "name": "<cosmosdb-name>",
  "readLocations": [
    {
      "documentEndpoint": "https://<cosmosdb-name>-westeurope.documents.azure.com:443/",
      "failoverPriority": 0,
      "id": "<cosmosdb-name>-westeurope",
      "locationName": "West Europe",
      "provisioningState": "Succeeded"
    }
  ],
  "resourceGroup": "myResourceGroup",
  "type": "Microsoft.DocumentDB/databaseAccounts",
  "writeLocations": [
    {
      "documentEndpoint": "https://<cosmosdb-name>-westeurope.documents.azure.com:443/",
      "failoverPriority": 0,
      "id": "<cosmosdb-name>-westeurope",
      "locationName": "West Europe",
      "provisioningState": "Succeeded"
    }
  ]
}
```

Connect your Node.js application to the database

In this step, you connect your MEAN.js sample application to the Azure Cosmos DB database account you just created.

Configure the connection string in your Node.js application

In your MEAN.js repository, open `config/env/local-development.js`.

Replace the content of this file with the following code. Be sure to also replace the two `<cosmosdb-name>` placeholders with your Cosmos account name.

```
'use strict';

module.exports = {
  db: {
    uri: 'mongodb://<cosmosdb-name>:<primary_master_key>@<cosmosdb-name>.documents.azure.com:10255/mean-dev?
ssl=true&sslverifycertificate=false'
  }
};
```

Retrieve the key

In order to connect to a Cosmos database, you need the database key. Use the az cosmosdb keys list command to retrieve the primary key.

```
az cosmosdb keys list --name <cosmosdb-name> --resource-group myResourceGroup --query "primaryMasterKey"
```

The Azure CLI outputs information similar to the following example.

```
"RUayjYjixJDWG5xTqIiXjC..."
```

Copy the value of `primaryMasterKey`. Paste this over the `<primary_master_key>` in `local-development.js`.

Save your changes.

Run the application again.

Run `npm start` again.

```
npm start
```

A console message should now tell you that the development environment is up and running.

Go to `http://localhost:3000` in a browser. Select **Sign Up** in the top menu and try to create two dummy users.

The MEAN.js sample application stores user data in the database. If you are successful and MEAN.js automatically signs into the created user, then your Azure Cosmos DB connection is working.

View data in Data Explorer

Data stored in a Cosmos database is available to view and query in the Azure portal.

To view, query, and work with the user data created in the previous step, login to the Azure portal in your web browser.

In the top Search box, enter **Azure Cosmos DB**. When your Cosmos account blade opens, select your Cosmos account. In the left navigation, select **Data Explorer**. Expand your collection in the Collections pane, and then you can view the documents in the collection, query the data, and even create and run stored procedures, triggers, and UDFs.

Deploy the Node.js application to Azure

In this step, you deploy your Node.js application to Cosmos DB.

You may have noticed that the configuration file that you changed earlier is for the development environment (

`/config/env/local-development.js`). When you deploy your application to App Service, it will run in the production environment by default. So now, you need to make the same change to the respective configuration file.

In your MEAN.js repository, open `config/env/production.js`.

In the `db` object, replace the value of `uri` as show in the following example. Be sure to replace the placeholders as before.

```
'mongodb://<cosmosdb-name>:<primary_master_key>@<cosmosdb-name>.documents.azure.com:10255/mean?
ssl=true&sslverifycertificate=false',
```

> **NOTE**
> The `ssl=true` option is important because of Cosmos DB requirements. For more information, see Connection string requirements.

In the terminal, commit all your changes into Git. You can copy both commands to run them together.

```
git add .
git commit -m "configured MongoDB connection string"
```

Clean up resources

When you're done with your app and Azure Cosmos DB account, you can delete the Azure resources you created so you don't incur more charges. To delete the resources:

1. In the Azure portal Search bar, search for and select **Resource groups**.
2. From the list, select the resource group you created for this quickstart.

3. On the resource group **Overview** page, select **Delete resource group**.

4. In the next window, enter the name of the resource group to delete, and then select **Delete**.

Next steps

In this quickstart, you learned how to create an Azure Cosmos DB MongoDB API account using the Azure Cloud Shell, and create and run a MEAN.js app to add users to the account. You can now import additional data to your Azure Cosmos DB account.

Import MongoDB data into Azure Cosmos DB

Quickstart: Build a Python app using Azure Cosmos DB's API for MongoDB

12/4/2020 • 7 minutes to read • Edit Online

APPLIES TO: Azure Cosmos DB API for MongoDB

In this quickstart, you use an Azure Cosmos DB for Mongo DB API account or the Azure Cosmos DB Emulator to run a Python Flask To-Do web app cloned from GitHub. Azure Cosmos DB is a multi-model database service that lets you quickly create and query document, table, key-value, and graph databases with global distribution and horizontal scale capabilities.

Prerequisites

- An Azure account with an active subscription. Create one for free. Or try Azure Cosmos DB for free without an Azure subscription. Or, you can use the Azure Cosmos DB Emulator.
- Python 3.6+
- Visual Studio Code with the Python Extension.

Clone the sample application

Now let's clone a Flask-MongoDB app from GitHub, set the connection string, and run it. You see how easy it is to work with data programmatically.

1. Open a command prompt, create a new folder named git-samples, then close the command prompt.

```
md "C:\git-samples"
```

2. Open a git terminal window, such as git bash, and use the `cd` command to change to the new folder to install the sample app.

```
cd "C:\git-samples"
```

3. Run the following command to clone the sample repository. This command creates a copy of the sample app on your computer.

```
git clone https://github.com/Azure-Samples/CosmosDB-Flask-Mongo-Sample.git
```

4. Run the following command to install the python modules.

```
pip install -r .\requirements.txt
```

5. Open the folder in Visual Studio Code.

Review the code

This step is optional. If you're interested in learning how the database resources are created in the code, you can review the following snippets. Otherwise, you can skip ahead to Run the web app.

The following snippets are all taken from the *app.py* file and uses the connection string for the local Azure Cosmos DB Emulator. The password needs to be split up as seen below to accommodate for the forward slashes that cannot be parsed otherwise.

- Initialize the MongoDB client, retrieve the database, and authenticate.

```
client = MongoClient("mongodb://127.0.0.1:10250/?ssl=true") #host uri
db = client.test    #Select the database
db.authenticate(name="localhost",password='C2y6yDjf5' + r'/R' +
'+ob0N8A7Cgv30VRDJIWEHLM+4QDU5DE2nQ9nDuVTqobD4b8mGGyPMbIZnqyMsEcaGQy67XIw' + r'/Jw==')
```

- Retrieve the collection or create it if it does not already exist.

```
todos = db.todo #Select the collection
```

- Create the app

```
app = Flask(__name__)
title = "TODO with Flask"
heading = "ToDo Reminder"
```

Run the web app

1. Make sure the Azure Cosmos DB Emulator is running.
2. Open a terminal window and `cd` to the directory that the app is saved in.
3. Then set the environment variable for the Flask app with `set FLASK_APP=app.py`, `$env:FLASK_APP = app.py` for PowerShell editors, or `export FLASK_APP=app.py` if you are using a Mac.
4. Run the app with `flask run` and browse to *http://127.0.0.1:5000/*.
5. Add and remove tasks and see them added and changed in the collection.

Create a database account

If you want to test the code against a live Azure Cosmos DB account, go to the Azure portal to create an account.

1. In a new browser window, sign in to the Azure portal.
2. In the left menu, select **Create a resource**.

3. On the **New** page, select **Databases** > **Azure Cosmos DB**.

4. On the **Create Azure Cosmos DB Account** page, enter the settings for the new Azure Cosmos DB account.

SETTING	VALUE	DESCRIPTION
Subscription	Your subscription	Select the Azure subscription that you want to use for this Azure Cosmos DB account.
Resource Group	Create new Then enter the same name as Account Name	Select **Create new**. Then enter a new resource group name for your account. For simplicity, use the same name as your Azure Cosmos DB account name.
Account Name	Enter a unique name	Enter a unique name to identify your Azure Cosmos DB account. Your account URI will be *mongo.cosmos.azure.com* appended to your unique account name. The account name can use only lowercase letters, numbers, and hyphens (-), and must be between 3 and 31 characters long.

SETTING	VALUE	DESCRIPTION
API	Azure Cosmos DB for Mongo DB API	The API determines the type of account to create. Azure Cosmos DB provides five APIs: Core (SQL) for document databases, Gremlin for graph databases, Azure Cosmos DB for Mongo DB API for document databases, Azure Table, and Cassandra. Currently, you must create a separate account for each API. Select **Azure Cosmos DB for Mongo DB API** because in this quickstart you are creating a collection that works with MongoDB. Learn more about Azure Cosmos DB for MongoDB API.
Location	Select the region closest to your users	Select a geographic location to host your Azure Cosmos DB account. Use the location that's closest to your users to give them the fastest access to the data.
Capacity mode	Provisioned throughput or Serverless	Select **Provisioned throughput** to create an account in provisioned throughput mode. Select **Serverless** to create an account in serverless mode. **Note**: Only MongoDB API version 3.6 is supported by serverless accounts. Choosing 3.2 as the version will force the account in provisioned throughput mode.

Select **Review+Create**. You can skip the **Network** and **Tags** section.

Home > New >

Create Azure Cosmos DB Account

Basics | Networking | Backup Policy | Encryption | Tags | Review + create

Azure Cosmos DB is a globally distributed, multi-model, fully managed database service. Try it for free, for 30 days with unlimited renewals. Go to production starting at $24/month per database, multiple containers included. Learn more

Project Details

Select the subscription to manage deployed resources and costs. Use resource groups like folders to organize and manage all your resources.

Subscription * — Visual Studio Enterprise

Resource Group * — Create new

Instance Details

Account Name * — Enter account name

API * — Azure Cosmos DB for MongoDB API

Notebooks (Preview) — On / Off

Location * — (US) West US

Capacity mode — Provisioned throughput / Serverless (preview)

Learn more about capacity mode

With Azure Cosmos DB free tier, you will get 400 RU/s and 5 GB of storage for free in an account. You can enable free tier on up to one account per subscription. Estimated $24/month discount per account.

Apply Free Tier Discount — Apply / Do Not Apply

Account Type — Production / Non-Production

Version — 3.6

Geo-Redundancy — Enable / Disable

Multi-region Writes — Enable / Disable

*Up to 33% off multi-region writes is available to qualifying new accounts only. Offer limited to accounts with both account locations and geo-redundancy, and applies only to multi-region writes in those same regions. Both Geo-Redundancy and Multi-region Writes must be enabled in account settings. Actual discount will vary based on number of qualifying regions selected.

Review + create | Previous | Next: Networking

5. The account creation takes a few minutes. Wait for the portal to display the **Congratulations! Your Azure Cosmos DB for Mongo DB API account is ready** page.

Update your connection string

To test the code against the live Azure Cosmos DB account, get your connection string information. Then copy it into the app.

1. In your Azure Cosmos DB account in the Azure portal, in the left navigation select **Connection String**, and then select **Read-write Keys**. You'll use the copy buttons on the right side of the screen to copy the username, connection string, and password.

2. Open the *app.py* file in the root directory.

3. Copy your **username** value from the portal (using the copy button) and make it the value of the **name** in your *app.py* file.

4. Then copy your **connection string** value from the portal and make it the value of the **MongoClient** in your *app.py* file.

5. Finally copy your **password** value from the portal and make it the value of the **password** in your *app.py* file.

You've now updated your app with all the info it needs to communicate with Azure Cosmos DB. You can run it the same way as before.

Deploy to Azure

To deploy this app, you can create a new web app in Azure and enable continuous deployment with a fork of this GitHub repo. Follow this tutorial to set up continuous deployment with GitHub in Azure.

When deploying to Azure, you should remove your application keys and make sure the section below is not

commented out:

```
client = MongoClient(os.getenv("MONGOURL"))
db = client.test    #Select the database
db.authenticate(name=os.getenv("MONGO_USERNAME"),password=os.getenv("MONGO_PASSWORD"))
```

You then need to add your MONGOURL, MONGO_PASSWORD, and MONGO_USERNAME to the application settings. You can follow this tutorial to learn more about Application Settings in Azure Web Apps.

If you don't want to create a fork of this repo, you can also select the **Deploy to Azure** button below. You should then go into Azure and set up the application settings with your Azure Cosmos DB account info.

Deploy to Azure

> **NOTE**
>
> If you plan to store your code in GitHub or other source control options, please be sure to remove your connection strings from the code. They can be set with application settings for the web app instead.

Review SLAs in the Azure portal

The Azure portal monitors your Cosmos DB account throughput, storage, availability, latency, and consistency. Charts for metrics associated with an Azure Cosmos DB Service Level Agreement (SLA) show the SLA value compared to actual performance. This suite of metrics makes monitoring your SLAs transparent.

To review metrics and SLAs:

1. Select **Metrics** in your Cosmos DB account's navigation menu.
2. Select a tab such as **Latency**, and select a timeframe on the right. Compare the **Actual** and **SLA** lines on the charts.

3. Review the metrics on the other tabs.

Clean up resources

When you're done with your app and Azure Cosmos DB account, you can delete the Azure resources you created so you don't incur more charges. To delete the resources:

1. In the Azure portal Search bar, search for and select **Resource groups**.
2. From the list, select the resource group you created for this quickstart.

3. On the resource group **Overview** page, select **Delete resource group**.

4. In the next window, enter the name of the resource group to delete, and then select **Delete**.

Next steps

In this quickstart, you learned how to create an Azure Cosmos DB for Mongo DB API account, and use the Azure Cosmos DB Emulator to run a Python Flask To-Do web app cloned from GitHub. You can now import additional data to your Azure Cosmos DB account.

Import MongoDB data into Azure Cosmos DB

QuickStart: Build a Xamarin.Forms app with .NET SDK and Azure Cosmos DB's API for MongoDB

12/4/2020 • 7 minutes to read • Edit Online

APPLIES TO: Azure Cosmos DB API for MongoDB

Azure Cosmos DB is Microsoft's globally distributed multi-model database service. You can quickly create and query document, key/value, and graph databases, all of which benefit from the global distribution and horizontal scale capabilities at the core of Azure Cosmos DB.

This quickstart demonstrates how to create a Cosmos account configured with Azure Cosmos DB's API for MongoDB, document database, and collection using the Azure portal. You'll then build a todo app Xamarin.Forms app by using the MongoDB .NET driver.

Prerequisites to run the sample app

To run the sample, you'll need Visual Studio or Visual Studio for Mac and a valid Azure CosmosDB account.

If you don't already have Visual Studio, download Visual Studio 2019 Community Edition with the **Mobile development with .NET** workload installed with setup.

If you prefer to work on a Mac, download Visual Studio for Mac and run the setup.

If you don't have an Azure subscription, create a free account before you begin.

Create a database account

1. In a new browser window, sign in to the Azure portal.
2. In the left menu, select **Create a resource**.

3. On the **New** page, select **Databases** > **Azure Cosmos DB**.

4. On the **Create Azure Cosmos DB Account** page, enter the settings for the new Azure Cosmos DB account.

SETTING	VALUE	DESCRIPTION
Subscription	Your subscription	Select the Azure subscription that you want to use for this Azure Cosmos DB account.
Resource Group	Create new Then enter the same name as Account Name	Select **Create new**. Then enter a new resource group name for your account. For simplicity, use the same name as your Azure Cosmos DB account name.
Account Name	Enter a unique name	Enter a unique name to identify your Azure Cosmos DB account. Your account URI will be *mongo.cosmos.azure.com* appended to your unique account name. The account name can use only lowercase letters, numbers, and hyphens (-), and must be between 3 and 31 characters long.

SETTING	VALUE	DESCRIPTION
API	Azure Cosmos DB for Mongo DB API	The API determines the type of account to create. Azure Cosmos DB provides five APIs: Core (SQL) for document databases, Gremlin for graph databases, Azure Cosmos DB for Mongo DB API for document databases, Azure Table, and Cassandra. Currently, you must create a separate account for each API. Select **Azure Cosmos DB for Mongo DB API** because in this quickstart you are creating a collection that works with MongoDB. Learn more about Azure Cosmos DB for MongoDB API.
Location	Select the region closest to your users	Select a geographic location to host your Azure Cosmos DB account. Use the location that's closest to your users to give them the fastest access to the data.
Capacity mode	Provisioned throughput or Serverless	Select **Provisioned throughput** to create an account in provisioned throughput mode. Select **Serverless** to create an account in serverless mode. **Note**: Only MongoDB API version 3.6 is supported by serverless accounts. Choosing 3.2 as the version will force the account in provisioned throughput mode.

Select **Review+Create**. You can skip the **Network** and **Tags** section.

Home > New >

Create Azure Cosmos DB Account

Basics | Networking | Backup Policy | Encryption | Tags | Review + create

Azure Cosmos DB is a globally distributed, multi-model, fully managed database service. Try it for free, for 30 days with unlimited renewals. Go to production starting at $24/month per database, multiple containers included. Learn more

Project Details

Select the subscription to manage deployed resources and costs. Use resource groups like folders to organize and manage all your resources.

Subscription * — Visual Studio Enterprise

Resource Group * — Create new

Instance Details

Account Name * — Enter account name

API * — Azure Cosmos DB for MongoDB API

Notebooks (Preview) — On / Off

Location * — (US) West US

Capacity mode — Provisioned throughput / Serverless (preview)

Learn more about capacity mode

With Azure Cosmos DB free tier, you will get 400 RU/s and 5 GB of storage for free in an account. You can enable free tier on up to one account per subscription. Estimated $24/month discount per account.

Apply Free Tier Discount — Apply / Do Not Apply

Account Type — Production / Non-Production

Version — 3.6

Geo-Redundancy — Enable / Disable

Multi-region Writes — Enable / Disable

*Up to 33% off multi-region writes is available to qualifying new accounts only. Offer limited to accounts with both account locations and geo-redundancy, and applies only to multi-region writes in those same regions. Both Geo-Redundancy and Multi-region Writes must be enabled in account settings. Actual discount will vary based on number of qualifying regions selected.

Review + create | Previous | Next: Networking

5. The account creation takes a few minutes. Wait for the portal to display the **Congratulations! Your Azure Cosmos DB for Mongo DB API account is ready** page.

The sample described in this article is compatible with MongoDB.Driver version 2.6.1.

Clone the sample app

First, download the sample app from GitHub. It implements a todo app with MongoDB's document storage model.

- Windows

1. On Windows open a command prompt or on Mac open the terminal, create a new folder named git-samples, then close the window.

```
md "C:\git-samples"
```

```
mkdir '$home\git-samples\
```

2. Open a git terminal window, such as git bash, and use the `cd` command to change to the new folder to install the sample app.

```
cd "C:\git-samples"
```

3. Run the following command to clone the sample repository. This command creates a copy of the sample app on your computer.

```
git clone https://github.com/Azure-Samples/azure-cosmos-db-mongodb-xamarin-getting-started.git
```

If you don't wish to use git, you can also download the project as a ZIP file

Review the code

This step is optional. If you're interested in learning how the database resources are created in the code, you can review the following snippets. Otherwise, you can skip ahead to Update your connection string.

The following snippets are all taken from the `MongoService` class, found at the following path: src/TaskList.Core/Services/MongoService.cs.

- Initialize the Mongo Client.

```
MongoClientSettings settings = MongoClientSettings.FromUrl(new MongoUrl(APIKeys.ConnectionString));

settings.SslSettings = new SslSettings() { EnabledSslProtocols = SslProtocols.Tls12 };

settings.RetryWrites = false;

MongoClient mongoClient = new MongoClient(settings);
```

- Retrieve a reference to the database and collection. The MongoDB .NET SDK will automatically create both the database and collection if they do not already exist.

```
string dbName = "MyTasks";
string collectionName = "TaskList";

var db = mongoClient.GetDatabase(dbName);

var collectionSettings = new MongoCollectionSettings
{
    ReadPreference = ReadPreference.Nearest
};

tasksCollection = db.GetCollection<MyTask>(collectionName, collectionSettings);
```

- Retrieve all documents as a List.

```
var allTasks = await TasksCollection
                .Find(new BsonDocument())
                .ToListAsync();
```

- Query for particular documents.

```
public async Task<List<MyTask>> GetIncompleteTasksDueBefore(DateTime date)
{
    var tasks = await TasksCollection
                    .AsQueryable()
                    .Where(t => t.Complete == false)
                    .Where(t => t.DueDate < date)
                    .ToListAsync();

    return tasks;
}
```

- Create a task and insert it into the collection.

```
public async Task CreateTask(MyTask task)
{
    await TasksCollection.InsertOneAsync(task);
}
```

- Update a task in a collection.

```
public async Task UpdateTask(MyTask task)
{
    await TasksCollection.ReplaceOneAsync(t => t.Id.Equals(task.Id), task);
}
```

- Delete a task from a collection.

```
public async Task DeleteTask(MyTask task)
{
    await TasksCollection.DeleteOneAsync(t => t.Id.Equals(task.Id));
}
```

Update your connection string

Now go back to the Azure portal to get your connection string information and copy it into the app.

1. In the Azure portal, in your Azure Cosmos DB account, in the left navigation click **Connection String**, and then click **Read-write Keys**. You'll use the copy buttons on the right side of the screen to copy the Primary Connection String in the next steps.
2. Open the **APIKeys.cs** file in the **Helpers** directory of the **TaskList.Core** project.
3. Copy your **primary connection string** value from the portal (using the copy button) and make it the value of the **ConnectionString** field in your **APIKeys.cs** file.
4. Remove `&replicaSet=globaldb` from the connection string. You will get a runtime error if you do not remove that value from the query string.

> **IMPORTANT**
>
> You must remove the `&replicaSet=globaldb` key/value pair from the connection string's query string in order to avoid a runtime error.

You've now updated your app with all the info it needs to communicate with Azure Cosmos DB.

Run the app

Visual Studio 2019

1. In Visual Studio, right-click on each project in **Solution Explorer** and then click **Manage NuGet Packages**.
2. Click **Restore all NuGet packages**.
3. Right click on the **TaskList.Android** and select **Set as startup project**.
4. Press F5 to start debugging the application.
5. If you want to run on iOS, first your machine is connected to a Mac (here are instructions on how to do so).
6. Right click on **TaskList.iOS** project and select **Set as startup project**.
7. Click F5 to start debugging the application.

Visual Studio for Mac

1. In the platform dropdown list, select either TaskList.iOS or TaskList.Android, depending which platform you want to run on.
2. Press cmd+Enter to start debugging the application.

Review SLAs in the Azure portal

The Azure portal monitors your Cosmos DB account throughput, storage, availability, latency, and consistency. Charts for metrics associated with an Azure Cosmos DB Service Level Agreement (SLA) show the SLA value compared to actual performance. This suite of metrics makes monitoring your SLAs transparent.

To review metrics and SLAs:

1. Select **Metrics** in your Cosmos DB account's navigation menu.

2. Select a tab such as **Latency**, and select a timeframe on the right. Compare the **Actual** and **SLA** lines on the charts.

3. Review the metrics on the other tabs.

Clean up resources

When you're done with your app and Azure Cosmos DB account, you can delete the Azure resources you created so you don't incur more charges. To delete the resources:

1. In the Azure portal Search bar, search for and select **Resource groups**.

2. From the list, select the resource group you created for this quickstart.

3. On the resource group **Overview** page, select **Delete resource group**.

4. In the next window, enter the name of the resource group to delete, and then select **Delete**.

Next steps

In this quickstart, you've learned how to create an Azure Cosmos DB account and run a Xamarin.Forms app using the API for MongoDB. You can now import additional data to your Cosmos DB account.

Import data into Azure Cosmos DB configured with Azure Cosmos DB's API for MongoDB

Quickstart: Connect a Go application to Azure Cosmos DB's API for MongoDB

12/4/2020 • 10 minutes to read • Edit Online

APPLIES TO: Azure Cosmos DB API for MongoDB

Azure Cosmos DB is a multi-model database service that lets you quickly create and query document, table, key-value, and graph databases with global distribution and horizontal scale capabilities. In this quickstart, you create and manage an Azure Cosmos DB account by using the Azure Cloud Shell, clone an existing sample application from GitHub and configure it to work with Azure Cosmos DB.

The sample application is a command-line based `todo` management tool written in Go. Azure Cosmos DB's API for MongoDB is compatible with the MongoDB wire protocol, making it possible for any MongoDB client driver to connect to it. This application uses the Go driver for MongoDB in a way that is transparent to the application that the data is stored in an Azure Cosmos DB database.

Prerequisites

- An Azure account with an active subscription. Create one for free. Or try Azure Cosmos DB for free without an Azure subscription. You can also use the Azure Cosmos DB Emulator with the connection string

  ```
  .mongodb://localhost:C2y6yDjf5/R+ob0N8A7Cgv30VRDJIWEHLM+4QDU5DE2nQ9nDuVTqobD4b8mGGyPMbIZnqyMsEcaGQy67XIw/Jw==@localhost:10255/admin?ssl=true
  ```

  .
- Go installed on your computer, and a working knowledge of Go.
- Git.
- Use Azure Cloud Shell using the bash environment.

  Launch Cloud Shell

- If you prefer, install the Azure CLI to run CLI reference commands.
  - If you're using a local install, sign in with Azure CLI by using the az login command. To finish the authentication process, follow the steps displayed in your terminal. See Sign in with Azure CLI for additional sign-in options.
  - When you're prompted, install Azure CLI extensions on first use. For more information about extensions, see Use extensions with Azure CLI.
  - Run az version to find the version and dependent libraries that are installed. To upgrade to the latest version, run az upgrade.

Clone the sample application

Run the following commands to clone the sample repository.

1. Open a command prompt, create a new folder named `git-samples`, then close the command prompt.

   ```
   mkdir "C:\git-samples"
   ```

2. Open a git terminal window, such as git bash, and use the `cd` command to change to the new folder to install the sample app.

   ```
   cd "C:\git-samples"
   ```

3. Run the following command to clone the sample repository. This command creates a copy of the sample app on your computer.

```
git clone https://github.com/Azure-Samples/cosmosdb-go-mongodb-quickstart
```

Review the code

This step is optional. If you're interested in learning how the application works, you can review the following snippets. Otherwise, you can skip ahead to Run the application. The application layout is as follows:

```
.
├── go.mod
├── go.sum
└── todo.go
```

The following snippets are all taken from the `todo.go` file.

Connecting the Go app to Azure Cosmos DB

`clientOptions` encapsulates the connection string for Azure Cosmos DB, which is passed in using an environment variable (details in the upcoming section). The connection is initialized using `mongo.NewClient` to which the `clientOptions` instance is passed. `Ping` function is invoked to confirm successful connectivity (it is a fail-fast strategy)

```
    ctx, cancel := context.WithTimeout(context.Background(), time.Second*10)
    defer cancel()

    clientOptions := options.Client().ApplyURI(mongoDBConnectionString).SetDirect(true)

    c, err := mongo.NewClient(clientOptions)
    err = c.Connect(ctx)
    if err != nil {
        log.Fatalf("unable to initialize connection %v", err)
    }

    err = c.Ping(ctx, nil)
    if err != nil {
        log.Fatalf("unable to connect %v", err)
    }
```

NOTE

Using the `SetDirect(true)` configuration is important, without which you will get the following connectivity error:
`unable to connect connection(cdb-ms-prod-<azure-region>-cm1.documents.azure.com:10255[-4]) connection is closed`

Create a `todo` item

To create a `todo`, we get a handle to a `mongo.Collection` and invoke the `InsertOne` function.

```
func create(desc string) {
    c := connect()
    ctx := context.Background()
    defer c.Disconnect(ctx)

    todoCollection := c.Database(database).Collection(collection)
    r, err := todoCollection.InsertOne(ctx, Todo{Description: desc, Status: statusPending})
    if err != nil {
        log.Fatalf("failed to add todo %v", err)
    }
```

We pass in a `Todo` struct that contains the description and the status (which is initially set to `pending`)

```
type Todo struct {
    ID          primitive.ObjectID `bson:"_id,omitempty"`
    Description string             `bson:"description"`
    Status      string             `bson:"status"`
}
```

List `todo` items

We can list TODOs based on criteria. A `bson.D` is created to encapsulate the filter criteria

```
func list(status string) {
    .....
    var filter interface{}
    switch status {
    case listAllCriteria:
        filter = bson.D{}
    case statusCompleted:
        filter = bson.D{{statusAttribute, statusCompleted}}
    case statusPending:
        filter = bson.D{{statusAttribute, statusPending}}
    default:
        log.Fatal("invalid criteria for listing todo(s)")
    }
```

`Find` is used to search for documents based on the filter and the result is converted into a slice of `Todo`

```
    todoCollection := c.Database(database).Collection(collection)
    rs, err := todoCollection.Find(ctx, filter)
    if err != nil {
        log.Fatalf("failed to list todo(s) %v", err)
    }
    var todos []Todo
    err = rs.All(ctx, &todos)
    if err != nil {
        log.Fatalf("failed to list todo(s) %v", err)
    }
```

Finally, the information is rendered in tabular format

```
    todoTable := [][]string{}

    for _, todo := range todos {
        s, _ := todo.ID.MarshalJSON()
        todoTable = append(todoTable, []string{string(s), todo.Description, todo.Status})
    }

    table := tablewriter.NewWriter(os.Stdout)
    table.SetHeader([]string{"ID", "Description", "Status"})

    for _, v := range todoTable {
        table.Append(v)
    }
    table.Render()
```

Update a `todo` item

A `todo` can be updated based on its `_id`. A `bson.D` filter is created based on the `_id` and another one is created for the updated information, which is a new status (`completed` or `pending`) in this case. Finally, the `UpdateOne` function is invoked with the filter and the updated document

```
func update(todoid, newStatus string) {
....
    todoCollection := c.Database(database).Collection(collection)
    oid, err := primitive.ObjectIDFromHex(todoid)
    if err != nil {
        log.Fatalf("failed to update todo %v", err)
    }
    filter := bson.D{{"_id", oid}}
    update := bson.D{{"$set", bson.D{{statusAttribute, newStatus}}}}
    _, err = todoCollection.UpdateOne(ctx, filter, update)
    if err != nil {
        log.Fatalf("failed to update todo %v", err)
    }
```

Delete a `todo`

A `todo` is deleted based on its `_id` and it is encapsulated in the form of a `bson.D` instance. `DeleteOne` is invoked to delete the document.

```
func delete(todoid string) {
....
    todoCollection := c.Database(database).Collection(collection)
    oid, err := primitive.ObjectIDFromHex(todoid)
    if err != nil {
        log.Fatalf("invalid todo ID %v", err)
    }
    filter := bson.D{{"_id", oid}}
    _, err = todoCollection.DeleteOne(ctx, filter)
    if err != nil {
        log.Fatalf("failed to delete todo %v", err)
    }
}
```

Build the application

Change into the directory where you cloned the application and build it (using `go build`).

```
cd monogdb-go-quickstart
go build -o todo
```

To confirm that the application was built properly.

```
./todo --help
```

Setup Azure Cosmos DB

Sign in to Azure

If you choose to install and use the CLI locally, this topic requires that you are running the Azure CLI version 2.0 or later. Run `az --version` to find the version. If you need to install or upgrade, see [Install Azure CLI].

If you are using an installed Azure CLI, sign in to your Azure subscription with the az login command and follow the on-screen directions. You can skip this step if you're using the Azure Cloud Shell.

```
az login
```

Add the Azure Cosmos DB module

If you are using an installed Azure CLI, check to see if the `cosmosdb` component is already installed by running the `az` command. If `cosmosdb` is in the list of base commands, proceed to the next command. You can skip this step if you're using the Azure Cloud Shell.

If `cosmosdb` is not in the list of base commands, reinstall Azure CLI.

Create a resource group

Create a resource group with the az group create. An Azure resource group is a logical container into which Azure resources like web apps, databases and storage accounts are deployed and managed.

The following example creates a resource group in the West Europe region. Choose a unique name for the resource group.

If you are using Azure Cloud Shell, select **Try It**, follow the onscreen prompts to login, then copy the command into the command prompt.

```
az group create --name myResourceGroup --location "West Europe"
```

Create an Azure Cosmos DB account

Create a Cosmos account with the az cosmosdb create command.

In the following command, please substitute your own unique Cosmos account name where you see the `<cosmosdb-name>` placeholder. This unique name will be used as part of your Cosmos DB endpoint (`https://<cosmosdb-name>.documents.azure.com/`), so the name needs to be unique across all Cosmos accounts in Azure.

```
az cosmosdb create --name <cosmosdb-name> --resource-group myResourceGroup --kind MongoDB
```

The `--kind MongoDB` parameter enables MongoDB client connections.

When the Azure Cosmos DB account is created, the Azure CLI shows information similar to the following example.

> **NOTE**
> This example uses JSON as the Azure CLI output format, which is the default. To use another output format, see Output formats for Azure CLI commands.

```
{
  "databaseAccountOfferType": "Standard",
  "documentEndpoint": "https://<cosmosdb-name>.documents.azure.com:443/",
  "id": "/subscriptions/00000000-0000-0000-0000-000000000000/resourceGroups/myResourceGroup/providers/Microsoft.Document
DB/databaseAccounts/<cosmosdb-name>",
  "kind": "MongoDB",
  "location": "West Europe",
  "name": "<cosmosdb-name>",
  "readLocations": [
    {
      "documentEndpoint": "https://<cosmosdb-name>-westeurope.documents.azure.com:443/",
      "failoverPriority": 0,
      "id": "<cosmosdb-name>-westeurope",
      "locationName": "West Europe",
      "provisioningState": "Succeeded"
    }
  ],
  "resourceGroup": "myResourceGroup",
  "type": "Microsoft.DocumentDB/databaseAccounts",
  "writeLocations": [
    {
      "documentEndpoint": "https://<cosmosdb-name>-westeurope.documents.azure.com:443/",
      "failoverPriority": 0,
      "id": "<cosmosdb-name>-westeurope",
      "locationName": "West Europe",
      "provisioningState": "Succeeded"
    }
  ]
}
```

Retrieve the database key

In order to connect to a Cosmos database, you need the database key. Use the az cosmosdb keys list command to retrieve the primary key.

```
az cosmosdb keys list --name <cosmosdb-name> --resource-group myResourceGroup --query "primaryMasterKey"
```

The Azure CLI outputs information similar to the following example.

```
"RUayjYjixJDWG5xTqIiXjC..."
```

Configure the application

Export the connection string, MongoDB database and collection names as environment variables.

```
export MONGODB_CONNECTION_STRING="mongodb://<COSMOSDB_ACCOUNT_NAME>:
<COSMOSDB_PASSWORD>@<COSMOSDB_ACCOUNT_NAME>.mongo.cosmos.azure.com:10255/?
ssl=true&replicaSet=globaldb&maxIdleTimeMS=120000&appName=@<COSMOSDB_ACCOUNT_NAME>@"
```

> **NOTE**
>
> The `ssl=true` option is important because of Cosmos DB requirements. For more information, see Connection string requirements.

For the `MONGODB_CONNECTION_STRING` environment variable, replace the placeholders for `<COSMOSDB_ACCOUNT_NAME>` and `<COSMOSDB_PASSWORD>`

1. `<COSMOSDB_ACCOUNT_NAME>`: The name of the Azure Cosmos DB account you created
2. `<COSMOSDB_PASSWORD>`: The database key extracted in the previous step

```
export MONGODB_DATABASE=todo-db
export MONGODB_COLLECTION=todos
```

You can choose your preferred values for `MONGODB_DATABASE` and `MONGODB_COLLECTION` or leave them as is.

Run the application

To create a `todo`

```
./todo --create "Create an Azure Cosmos DB database account"
```

If successful, you should see an output with the MongoDB `_id` of the newly created document:

```
added todo ObjectID("5e9fd6befd2f076d1f03bd8a")
```

Create another `todo`

```
./todo --create "Get the MongoDB connection string using the Azure CLI"
```

List all the `todo`s

```
./todo --list all
```

You should see the ones you just added in a tabular format as such

```
+---------------------------+-------------------------------+-----------+
|            ID             |          DESCRIPTION          |  STATUS   |
+---------------------------+-------------------------------+-----------+
| "5e9fd6b1bcd2fa6bd267d4c4" | Create an Azure Cosmos DB      | pending   |
|                           | database account              |           |
| "5e9fd6befd2f076d1f03bd8a" | Get the MongoDB connection     | pending   |
|                           | string using the Azure CLI    |           |
+---------------------------+-------------------------------+-----------+
```

To update the status of a `todo` (e.g. change it to `completed` status), use the `todo` ID

```
./todo --update 5e9fd6b1bcd2fa6bd267d4c4,completed
```

List only the completed `todo`s

```
./todo --list completed
```

You should see the one you just updated

```
+---------------------------+-------------------------------+-----------+
|            ID             |          DESCRIPTION          |  STATUS   |
+---------------------------+-------------------------------+-----------+
| "5e9fd6b1bcd2fa6bd267d4c4" | Create an Azure Cosmos DB     | completed |
|                           | database account              |           |
+---------------------------+-------------------------------+-----------+
```

View data in Data Explorer

Data stored in Azure Cosmos DB is available to view and query in the Azure portal.

To view, query, and work with the user data created in the previous step, login to the Azure portal in your web browser.

In the top Search box, enter **Azure Cosmos DB**. When your Cosmos account blade opens, select your Cosmos account. In the left navigation, select **Data Explorer**. Expand your collection in the Collections pane, and then you can view the documents in the collection, query the data, and even create and run stored procedures, triggers, and UDFs.

Delete a `todo` using it's ID

```
./todo --delete 5e9fd6b1bcd2fa6bd267d4c4,completed
```

List the `todo`s to confirm

```
./todo --list all
```

The `todo` you just deleted should not be present

```
+---------------------------+-------------------------------+-----------+
|            ID             |          DESCRIPTION          |  STATUS   |
+---------------------------+-------------------------------+-----------+
| "5e9fd6befd2f076d1f03bd8a" | Get the MongoDB connection    | pending   |
|                           | string using the Azure CLI    |           |
+---------------------------+-------------------------------+-----------+
```

Clean up resources

When you're done with your app and Azure Cosmos DB account, you can delete the Azure resources you created so you don't incur more charges. To delete the resources:

1. In the Azure portal Search bar, search for and select **Resource groups**.
2. From the list, select the resource group you created for this quickstart.

3. On the resource group **Overview** page, select **Delete resource group**.

4. In the next window, enter the name of the resource group to delete, and then select **Delete**.

Next steps

In this quickstart, you learned how to create an Azure Cosmos DB MongoDB API account using the Azure Cloud Shell, and create and run a Go command-line app to manage `todo`s. You can now import additional data to your Azure Cosmos DB account.

Import MongoDB data into Azure Cosmos DB

Build an app using Node.js and Azure Cosmos DB's API for MongoDB

11/2/2020 • 2 minutes to read • Edit Online

APPLIES TO: Azure Cosmos DB API for MongoDB

This example shows you how to build a console app using Node.js and Azure Cosmos DB's API for MongoDB.

To use this example, you must:

- Create a Cosmos account configured to use Azure Cosmos DB's API for MongoDB.
- Retrieve your connection string information.

Create the app

1. Create a *app.js* file and copy & paste the code below.

```
var MongoClient = require('mongodb').MongoClient;
var assert = require('assert');
var ObjectId = require('mongodb').ObjectID;
var url = 'mongodb://<username>:<password>@<endpoint>.documents.azure.com:10255/?ssl=true';

var insertDocument = function(db, callback) {
db.collection('families').insertOne( {
        "id": "AndersenFamily",
        "lastName": "Andersen",
        "parents": [
            { "firstName": "Thomas" },
            { "firstName": "Mary Kay" }
        ],
        "children": [
            { "firstName": "John", "gender": "male", "grade": 7 }
        ],
        "pets": [
            { "givenName": "Fluffy" }
        ],
        "address": { "country": "USA", "state": "WA", "city": "Seattle" }
    }, function(err, result) {
    assert.equal(err, null);
    console.log("Inserted a document into the families collection.");
    callback();
});
};

var findFamilies = function(db, callback) {
var cursor =db.collection('families').find( );
cursor.each(function(err, doc) {
    assert.equal(err, null);
    if (doc != null) {
        console.dir(doc);
    } else {
        callback();
    }
});
};

var updateFamilies = function(db, callback) {
db.collection('families').updateOne(
    { "lastName" : "Andersen" },
```

```
    {
        $set: { "pets": [
            { "givenName": "Fluffy" },
            { "givenName": "Rocky"}
        ] },
        $currentDate: { "lastModified": true }
    }, function(err, results) {
    console.log(results);
    callback();
});
};

var removeFamilies = function(db, callback) {
db.collection('families').deleteMany(
    { "lastName": "Andersen" },
    function(err, results) {
        console.log(results);
        callback();
    }
);
};

MongoClient.connect(url, function(err, client) {
assert.equal(null, err);
var db = client.db('familiesdb');
insertDocument(db, function() {
    findFamilies(db, function() {
    updateFamilies(db, function() {
        removeFamilies(db, function() {
            client.close();
        });
    });
    });
});
});
```

Optional: If you are using the **MongoDB Node.js 2.2 driver**, please replace the following code snippet:

Original:

```
MongoClient.connect(url, function(err, client) {
assert.equal(null, err);
var db = client.db('familiesdb');
insertDocument(db, function() {
    findFamilies(db, function() {
    updateFamilies(db, function() {
        removeFamilies(db, function() {
            client.close();
        });
    });
    });
});
});
```

Should be replaced with:

```
MongoClient.connect(url, function(err, db) {
assert.equal(null, err);
insertDocument(db, function() {
    findFamilies(db, function() {
    updateFamilies(db, function() {
        removeFamilies(db, function() {
            db.close();
        });
    });
    });
});
});
```

2. Modify the following variables in the *app.js* file per your account settings (Learn how to find your connection string):

> **IMPORTANT**
>
> The **MongoDB Node.js 3.0 driver** requires encoding special characters in the Cosmos DB password. Make sure to encode '=' characters as %3D
>
> Example: The password *jm1HbNdLg5zxEuyD86ajvINRFrFCUX0bIWP15ATK3BvSv==* encodes to *jm1HbNdLg5zxEuyD86ajvINRFrFCUX0bIWP15ATK3BvSv%3D%3D*
>
> The **MongoDB Node.js 2.2 driver** does not require encoding special characters in the Cosmos DB password.

```
var url = 'mongodb://<endpoint>:<password>@<endpoint>.documents.azure.com:10255/?ssl=true';
```

3. Open your favorite terminal, run **npm install mongodb --save**, then run your app with **node app.js**

Next steps

- Learn how to use Studio 3T with Azure Cosmos DB's API for MongoDB.
- Learn how to use Robo 3T with Azure Cosmos DB's API for MongoDB.
- Explore MongoDB samples with Azure Cosmos DB's API for MongoDB.

Create an Angular app with Azure Cosmos DB's API for MongoDB

11/2/2020 • 2 minutes to read • Edit Online

APPLIES TO: Azure Cosmos DB API for MongoDB

This multi-part tutorial demonstrates how to create a new app written in Node.js with Express and Angular and then connect it to your Cosmos account configured with Cosmos DB's API for MongoDB.

Azure Cosmos DB is Microsoft's fast NoSQL database with open APIs for any scale. It allows you to develop modern apps with SLA-backed speed and availability, automatic and instant scalability, and open source APIs for many NoSQL engines.

This multi-part tutorial covers the following tasks:

- Create a Node.js Express app with the Angular CLI
- Build the UI with Angular
- Create an Azure Cosmos DB account using the Azure CLI
- Use Mongoose to connect to Azure Cosmos DB
- Add Post, Put, and Delete functions to the app

Want to do build this same app with React? See the React tutorial video series.

Video walkthrough

Finished project

This tutorial walks you through the steps to build the application step-by-step. If you want to download the finished project, you can get the completed application from the angular-cosmosdb repo on GitHub.

Next steps

In this part of the tutorial, you've done the following:

- Seen an overview of the steps to create a MEAN.js app with Azure Cosmos DB.

You can proceed to the next part of the tutorial to create the Node.js Express app.

Create a Node.js Express app with the Angular CLI

Create an Angular app with Azure Cosmos DB's API for MongoDB - Create a Node.js Express app

11/2/2020 • 3 minutes to read • Edit Online

APPLIES TO: Azure Cosmos DB API for MongoDB

This multi-part tutorial demonstrates how to create a new app written in Node.js with Express and Angular and then connect it to your Cosmos account configured with Cosmos DB's API for MongoDB.

Part 2 of the tutorial builds on the introduction and covers the following tasks:

- Install the Angular CLI and TypeScript
- Create a new project using Angular
- Build out the app using the Express framework
- Test the app in Postman

Video walkthrough

Prerequisites

Before starting this part of the tutorial, ensure you've watched the introduction video.

This tutorial also requires:

- Node.js version 8.4.0 or above.
- Postman
- Visual Studio Code or your favorite code editor.

> **TIP**
> This tutorial walks you through the steps to build the application step-by-step. If you want to download the finished project, you can get the completed application from the angular-cosmosdb repo on GitHub.

Install the Angular CLI and TypeScript

1. Open a Windows Command Prompt or Mac Terminal window and install the Angular CLI.

```
npm install -g @angular/cli
```

2. Install TypeScript by entering the following command in the prompt.

```
npm install -g typescript
```

Use the Angular CLI to create a new project

1. At the command prompt, change to the folder where you want to create your new project, then run the

following command. This command creates a new folder and project named angular-cosmosdb and installs the Angular components required for a new app. It uses the minimal setup (--minimal), and specifies that the project uses Sass (a CSS-like syntax with the flag --style scss).

```
ng new angular-cosmosdb --minimal --style scss
```

2. Once the command completes, change directories into the src/client folder.

```
cd angular-cosmosdb
```

3. Then open the folder in Visual Studio Code.

```
code .
```

Build the app using the Express framework

1. In Visual Studio Code, in the **Explorer** pane, right-click the **src** folder, click **New Folder**, and name the new folder *server*.

2. In the **Explorer** pane, right-click the **server** folder, click **New File**, and name the new file *index.js*.

3. Back at the command prompt, use the following command to install the body parser. This helps our app parse the JSON data that are passed in through the APIs.

```
npm i express body-parser --save
```

4. In Visual Studio Code, copy the following code into the index.js file. This code:

   - References Express
   - Pulls in the body-parser for reading JSON data in the body of requests
   - Uses a built-in feature called path
   - Sets root variables to make it easier to find where our code is located
   - Sets up a port
   - Cranks up Express
   - Tells the app how to use the middleware that were going to be using to serve up the server
   - Serves everything that's in the dist folder, which will be the static content
   - Serves up the application, and serves index.html for any GET requests not found on the server (for deep links)
   - Starts the server with app.listen
   - Uses an arrow function to log that the port is alive

```
const express = require('express');
const bodyParser = require('body-parser');
const path = require('path');
const routes = require('./routes');

const root = './';
const port = process.env.PORT || '3000';
const app = express();

app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: false }));
app.use(express.static(path.join(root, 'dist/angular-cosmosdb')));
app.use('/api', routes);
app.get('*', (req, res) => {
  res.sendFile('dist/angular-cosmosdb/index.html', {root});
});

app.listen(port, () => console.log(`API running on localhost:${port}`));
```

5. In Visual Studio Code, in the **Explorer** pane, right-click the **server** folder, and then click **New file**. Name the new file *routes.js*.

6. Copy the following code into **routes.js**. This code:

   - References the Express router
   - Gets the heroes
   - Sends back the JSON for a defined hero

```
const express = require('express');
const router = express.Router();

router.get('/heroes', (req, res) => {
 res.send(200, [
    {"id": 10, "name": "Starlord", "saying": "oh yeah"}
 ])
});

module.exports=router;
```

7. Save all your modified files.

8. In Visual Studio Code, click the **Debug** button , click the Gear button . The new launch.json file opens in Visual Studio Code.

9. On line 11 of the launch.json file, change `"${workspaceFolder}\\server"` to `"program": "${workspaceRoot}/src/server/index.js"` and save the file.

10. Click the **Start Debugging** button to run the app.

    The app should run without errors.

Use Postman to test the app

1. Now open Postman and put `http://localhost:3000/api/heroes` in the GET box.

2. Click the **Send** button and get the json response from the app.

   This response shows the app is up and running locally.

Next steps

In this part of the tutorial, you've done the following:

- Created a Node.js project using the Angular CLI
- Tested the app using Postman

You can proceed to the next part of the tutorial to build the UI.

Build the UI with Angular

Create an Angular app with Azure Cosmos DB's API for MongoDB - Build the UI with Angular

11/2/2020 • 9 minutes to read • Edit Online

APPLIES TO: Azure Cosmos DB API for MongoDB

This multi-part tutorial demonstrates how to create a new app written in Node.js with Express and Angular and then connect it to your Cosmos account configured with Cosmos DB's API for MongoDB.

Part 3 of the tutorial builds on Part 2 and covers the following tasks:

- Build the Angular UI
- Use CSS to set the look and feel
- Test the app locally

Video walkthrough

Prerequisites

Before starting this part of the tutorial, ensure you've completed the steps in Part 2 of the tutorial.

> **TIP**
> This tutorial walks you through the steps to build the application step-by-step. If you want to download the finished project, you can get the completed application from the angular-cosmosdb repo on GitHub.

Build the UI

1. In Visual Studio Code, click the Stop button to stop the Node app.

2. In your Windows Command Prompt or Mac Terminal window, enter the following command to generate a heroes component. In this code g=generate, c=component, heroes=name of component, and it's using a flat file structure (--flat) so that a subfolder isn't created for it.

   ```
   ng g c heroes --flat
   ```

   The terminal window displays confirmation of the new components.

   ```
   CREATE src/app/heroes.component.html (25 bytes)
   CREATE src/app/heroes.component.spec.ts (628 bytes)
   CREATE src/app/heroes.component.ts (270 bytes)
   CREATE src/app/heroes.component.scss (0 bytes)
   UPDATE src/app/app.module.ts (389 bytes)
   ```

   Let's take a look at the files that were created and updated.

3. In Visual Studio Code, in the **Explorer** pane, navigate to the new **src\app** folder and open the new **heroes.component.ts** file generated within the app folder. This TypeScript component file was created by

the previous command.

> **TIP**
>
> If the app folder doesn't display in Visual Studio Code, enter CMD + SHIFT P on a Mac or Ctrl + Shift + P on Windows to open the Command Palette, and then type *Reload Window* to pick up the system change.

4. In the same folder, open the **app.module.ts** file, and notice that it added the `HeroesComponent` to the declarations on line 5 and it imported it as well on line 10.

5. Go back to the **heroes.component.html** file and copy in this code. The `<div>` is the container for the entire page. Inside of the container there is a list of heroes which we need to create so that when you click on one you can select it and edit it or delete it in the UI. Then in the HTML we've got some styling so you know which one has been selected. There's also an edit area so that you can add a new hero or edit an existing one.

```
<div>
  <ul class="heroes">
    <li *ngFor="let hero of heroes" (click)="onSelect(hero)" [class.selected]="hero === selectedHero">
      <button class="delete-button" (click)="deleteHero(hero)">Delete</button>
      <div class="hero-element">
        <div class="badge">{{hero.id}}</div>
        <div class="name">{{hero.name}}</div>
        <div class="saying">{{hero.saying}}</div>
      </div>
    </li>
  </ul>
  <div class="editarea">
    <button (click)="enableAddMode()">Add New Hero</button>
    <div *ngIf="selectedHero">
      <div class="editfields">
        <div>
          <label>id: </label>
          <input [(ngModel)]="selectedHero.id" placeholder="id" *ngIf="addingHero" />
          <label *ngIf="!addingHero" class="value">{{selectedHero.id}}</label>
        </div>
        <div>
          <label>name: </label>
          <input [(ngModel)]="selectedHero.name" placeholder="name" />
        </div>
        <div>
          <label>saying: </label>
          <input [(ngModel)]="selectedHero.saying" placeholder="saying" />
        </div>
      </div>
      <button (click)="cancel()">Cancel</button>
      <button (click)="save()">Save</button>
    </div>
  </div>
</div>
```

6. Now that we've got the HTML in place, we need to add it to the **heroes.component.ts** file so we can interact with the template. The following code adds the template to your component file. A constructor has been added that gets some heroes, and initializes the hero service component to go get all the data. This code also adds all of the necessary methods for handling events in the UI. You can copy the following code over the existing code in **heroes.component.ts**. It's expected to see errors at the Hero and HeroService areas as the corresponding components aren't imported yet, you will fix these error in the next section.

```
import { Component, OnInit } from '@angular/core';

@Component({
  selector: 'app-heroes',
  templateUrl: './heroes.component.html',
  styleUrls: ['./heroes.component.scss']
})
export class HeroesComponent implements OnInit {
  addingHero = false;
  heroes: any = [];
  selectedHero: Hero;

  constructor(private heroService: HeroService) {}

  ngOnInit() {
   this.getHeroes();
  }

  cancel() {
    this.addingHero = false;
    this.selectedHero = null;
  }

  deleteHero(hero: Hero) {
    this.heroService.deleteHero(hero).subscribe(res => {
      this.heroes = this.heroes.filter(h => h !== hero);
      if (this.selectedHero === hero) {
        this.selectedHero = null;
      }
    });
  }

  getHeroes() {
    return this.heroService.getHeroes().subscribe(heroes => {
      this.heroes = heroes;
    });
  }

  enableAddMode() {
    this.addingHero = true;
    this.selectedHero = new Hero();
  }

  onSelect(hero: Hero) {
    this.addingHero = false;
    this.selectedHero = hero;
  }

  save() {
    if (this.addingHero) {
      this.heroService.addHero(this.selectedHero).subscribe(hero => {
        this.addingHero = false;
        this.selectedHero = null;
        this.heroes.push(hero);
      });
    } else {
      this.heroService.updateHero(this.selectedHero).subscribe(hero => {
        this.addingHero = false;
        this.selectedHero = null;
      });
    }
  }
}
```

7. In **Explorer**, open the **app/app.module.ts** file and update the imports section to add an import for a `FormsModule`. The import section should now look as follows:

```
imports: [
  BrowserModule,
  FormsModule
],
```

8. In the **app/app.module.ts** file add an import for the new FormsModule module on line 3.

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { FormsModule } from '@angular/forms';
```

Use CSS to set the look and feel

1. In the Explorer pane, open the **src/styles.scss** file.

2. Copy the following code into the **styles.scss** file, replacing the existing content of the file.

```
/* You can add global styles to this file, and also import other style files */

* {
  font-family: Arial;
}
h2 {
  color: #444;
  font-weight: lighter;
}
body {
  margin: 2em;
}

body,
input[text],
button {
  color: #888;
  // font-family: Cambria, Georgia;
}
button {
  font-size: 14px;
  font-family: Arial;
  background-color: #eee;
  border: none;
  padding: 5px 10px;
  border-radius: 4px;
  cursor: pointer;
  cursor: hand;
  &:hover {
    background-color: #cfd8dc;
  }
  &.delete-button {
    float: right;
    background-color: gray !important;
    background-color: rgb(216, 59, 1) !important;
    color: white;
    padding: 4px;
    position: relative;
    font-size: 12px;
  }
}
div {
  margin: .1em;
}

.selected {
```

```
    background-color: #cfd8dc !important;
    background-color: rgb(0, 120, 215) !important;
    color: white;
  }

  .heroes {
    float: left;
    margin: 0 0 2em 0;
    list-style-type: none;
    padding: 0;
    li {
      cursor: pointer;
      position: relative;
      left: 0;
      background-color: #eee;
      margin: .5em;
      padding: .5em;
      height: 3.0em;
      border-radius: 4px;
      width: 17em;
      &:hover {
        color: #607d8b;
        color: rgb(0, 120, 215);
        background-color: #ddd;
        left: .1em;
      }
      &.selected:hover {
        /*background-color: #BBD8DC !important;*/
        color: white;
      }
    }
    .text {
      position: relative;
      top: -3px;
    }
    .saying {
      margin: 5px 0;
    }
    .name {
      font-weight: bold;
    }
    .badge {
      /* display: inline-block; */
      float: left;
      font-size: small;
      color: white;
      padding: 0.7em 0.7em 0 0.5em;
      background-color: #607d8b;
      background-color: rgb(0, 120, 215);
      background-color:rgb(134, 183, 221);
      line-height: 1em;
      position: relative;
      left: -1px;
      top: -4px;
      height: 3.0em;
      margin-right: .8em;
      border-radius: 4px 0 0 4px;
      width: 1.2em;
    }
  }

  .header-bar {
    background-color: rgb(0, 120, 215);
    height: 4px;
    margin-top: 10px;
    margin-bottom: 10px;
  }

  label {
```

```
    display: inline-block;
    width: 4em;
    margin: .5em 0;
    color: #888;
    &.value {
      margin-left: 10px;
      font-size: 14px;
    }
  }

  input {
    height: 2em;
    font-size: 1em;
    padding-left: .4em;
    &::placeholder {
        color: lightgray;
        font-weight: normal;
        font-size: 12px;
        letter-spacing: 3px;
    }
  }

  .editarea {
    float: left;
    input {
      margin: 4px;
      height: 20px;
      color: rgb(0, 120, 215);
    }
    button {
      margin: 8px;
    }
    .editfields {
      margin-left: 12px;
    }
  }
```

3. Save the file.

Display the component

Now that we have the component, how do we get it to show up on the screen? Let's modify the default components in **app.component.ts**.

1. In the Explorer pane, open **/app/app.component.ts**, change the title to Heroes, and then put the name of the component we created in **heroes.components.ts** (app-heroes) to refer to that new component. The contents of the file should now look as follows:

```
import { Component } from '@angular/core';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
  template: `
  <h1>Heroes</h1>
  <div class="header-bar"></div>
  <app-heroes></app-heroes>
`
})
export class AppComponent {
  title = 'app';
}
```

2. There are other components in **heroes.components.ts** that we're referring to, like the Hero component, so we need to go create that, too. In the Angular CLI command prompt, use the following command to create a hero model and a file named **hero.ts**, where g=generate, cl=class, and hero=name of class.

```
ng g cl hero
```

3. In the Explorer pane, open **src\app\hero.ts**. In **hero.ts**, replace the content of the file with the following code, which adds a Hero class with an ID, a name, and a saying.

```
  export class Hero {
  id: number;
  name: string;
  saying: string;
}
```

4. Go back to **heroes.components.ts** and notice that on the `selectedHero: Hero;` line (line 10), `Hero` has a red line underneath.

5. Left-click the term `Hero`, and Visual Studio displays a lightbulb icon on the left side of the code block.

6. Click the lightbulb and then click **Import Hero from "/app/hero".** or **Import Hero from "./hero".** (The message changes depending on your setup)

   A new line of code appears on line 2. If line 2 references /app/hero, modify it so that it references the hero file from the local folder (./hero). Line 2 should look like this:

```
import { Hero } from "./hero";
```

   That takes care of the model, but we still need to create the service.

Create the Service

1. In the Angular CLI command prompt, enter the following command to create a hero service in **app.module.ts**, where g=generate, s=service, hero=name of service, -m=put in app.module.

```
ng g s hero -m app.module
```

2. In Visual Studio Code, go back to **heroes.components.ts**. Notice that on the `constructor(private heroService: HeroService) {}` line (line 13), `HeroService` has a red line underneath. Click `HeroService`, and you'll get the lightbulb on the left side of code block. Click the light bulb and then click **Import HeroService from "./hero.service "**. or **Import HeroService from "/app/hero.service "**.

   Clicking the light bulb inserts a new line of code on line 2. If line 2 references the /app/hero.service folder, modify it so that it references the hero file from the local folder (./hero.service). Line 2 should look like this:

```
import { HeroService } from "./hero.service"
```

3. In Visual Studio Code, open **hero.service.ts** and copy in the following code, replacing the content of the file.

```
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';

import { Hero } from './hero';

const api = '/api';

@Injectable()
export class HeroService {
  constructor(private http: HttpClient) {}

  getHeroes() {
    return this.http.get<Array<Hero>>(`${api}/heroes`)
  }

  deleteHero(hero: Hero) {
    return this.http.delete(`${api}/hero/${hero.id}`);
  }

  addHero(hero: Hero) {
    return this.http.post<Hero>(`${api}/hero/`, hero);
  }

  updateHero(hero: Hero) {
    return this.http.put<Hero>(`${api}/hero/${hero.id}`, hero);
  }
}
```

   This code uses the newest version of the HttpClient that Angular offers, which is a module that you need to provide, so we'll do that next.

4. In Visual Studio Code, open **app.module.ts** and import the HttpClientModule by updating the import section to include HttpClientModule.

```
imports: [
  BrowserModule,
  FormsModule,
  HttpClientModule
],
```

5. In **app.module.ts**, add the HttpClientModule import statement the list of imports.

```
import { HttpClientModule } from '@angular/common/http';
```

6. Save all files in Visual Studio Code.

Build the app

1. At the command prompt, enter the following command to build the Angular application.

   ```
   ng b
   ```

   If there are any problems, the terminal window displays information about the files to fix. When the build completes, the new files go into the **dist** folder. You can review the files new in the **dist** folder if you want.

   Now let's run the app.

2. In Visual Studio Code, click the **Debug** button on the left side, then click the **Start Debugging** button .

3. Now open an internet browser and navigate to **localhost:3000** and see the app running locally.

Next steps

In this part of the tutorial, you've done the following:

- Built the Angular UI
- Tested the app locally

You can proceed to the next part of the tutorial to create an Azure Cosmos DB account.

Create an Azure Cosmos DB account using the Azure CLI

Create an Angular app with Azure Cosmos DB's API for MongoDB - Create a Cosmos account

11/2/2020 • 4 minutes to read • Edit Online

APPLIES TO: Azure Cosmos DB API for MongoDB

This multi-part tutorial demonstrates how to create a new app written in Node.js with Express and Angular and then connect it to your Cosmos account configured with Cosmos DB's API for MongoDB.

Part 4 of the tutorial builds on Part 3 and covers the following tasks:

- Create an Azure resource group using the Azure CLI
- Create a Cosmos account using the Azure CLI

Video walkthrough

Prerequisites

Before starting this part of the tutorial, ensure you've completed the steps in Part 3 of the tutorial.

In this tutorial section, you can either use the Azure Cloud Shell (in your internet browser) or the Azure CLI installed locally.

Use Azure Cloud Shell

Azure hosts Azure Cloud Shell, an interactive shell environment that you can use through your browser. You can use either Bash or PowerShell with Cloud Shell to work with Azure services. You can use the Cloud Shell preinstalled commands to run the code in this article without having to install anything on your local environment.

To start Azure Cloud Shell:

OPTION	EXAMPLE/LINK
Select **Try It** in the upper-right corner of a code block. Selecting **Try It** doesn't automatically copy the code to Cloud Shell.	
Go to https://shell.azure.com, or select the **Launch Cloud Shell** button to open Cloud Shell in your browser.	
Select the **Cloud Shell** button on the menu bar at the upper right in the Azure portal.	

To run the code in this article in Azure Cloud Shell:

1. Start Cloud Shell.
2. Select the **Copy** button on a code block to copy the code.
3. Paste the code into the Cloud Shell session by selecting **Ctrl+Shift+V** on Windows and Linux or by

selecting **Cmd+Shift+V** on macOS.

4. Select **Enter** to run the code.

Sign in to Azure

You'll use the Azure CLI to create the resources needed to host your app in Azure. If you run Azure CLI commands in the Cloud Shell, you're already signed in. To run Azure CLI commands locally, sign in to your Azure subscription with the az login command and follow the on-screen directions.

```
az login
```

Create a resource group

A resource group is a logical container into which Azure resources, such as web apps, databases, and storage accounts, are deployed and managed. For example, you can choose to delete the entire resource group in one simple step later.

In the Cloud Shell, create a resource group with the `az group create` command. The following example creates a resource group named *myResourceGroup* in the *West Europe* location. To see all supported locations for App Service in **Free** tier, run the `az appservice list-locations --sku FREE` command.

```
az group create --name myResourceGroup --location "West Europe"
```

You generally create your resource group and the resources in a region near you.

When the command finishes, a JSON output shows you the resource group properties.

> **TIP**
> This tutorial walks you through the steps to build the application step-by-step. If you want to download the finished project, you can get the completed application from the angular-cosmosdb repo on GitHub.

Create an Azure Cosmos DB account

Create an Azure Cosmos DB account with the `az cosmosdb create` command.

```
az cosmosdb create --name <cosmosdb-name> --resource-group myResourceGroup --kind MongoDB
```

- For `<cosmosdb-name>` use your own unique Azure Cosmos DB account name, the name needs to be unique across all Azure Cosmos DB account names in Azure.
- The `--kind MongoDB` setting enables the Azure Cosmos DB to have MongoDB client connections.

It may take a minute or two for the command to complete. When it's done, the terminal window displays information about the new database.

Once the Azure Cosmos DB account has been created:

1. Open a new browser window and go to https://portal.azure.com
2. Click the Azure Cosmos DB logo on the left bar, and it shows you all the Azure Cosmos DBs you have.
3. Click on the Azure Cosmos DB account you just created, select the **Overview** tab and scroll down to view

the map where the database is located.

4. Scroll down on the left navigation and click the **Replicate data globally** tab, this displays a map where you can see the different areas you can replicate into. For example, you can click Australia Southeast or Australia East and replicate your data to Australia. You can learn more about global replication in How to distribute data globally with Azure Cosmos DB. For now, let's just keep the once instance and when we want to replicate, we know how.

Next steps

In this part of the tutorial, you've done the following:

- Created an Azure resource group using the Azure CLI
- Created an Azure Cosmos DB account using the Azure CLI

You can proceed to the next part of the tutorial to connect Azure Cosmos DB to your app using Mongoose.

Use Mongoose to connect to Azure Cosmos DB

Create an Angular app with Azure Cosmos DB's API for MongoDB - Use Mongoose to connect to Cosmos DB

11/2/2020 • 6 minutes to read • Edit Online

APPLIES TO: Azure Cosmos DB API for MongoDB

This multi-part tutorial demonstrates how to create a Node.js app with Express and Angular, and connect it to it to your Cosmos account configured with Cosmos DB's API for MongoDB. This article describes Part 5 of the tutorial and builds on Part 4.

In this part of the tutorial, you will:

- Use Mongoose to connect to Cosmos DB.
- Get your Cosmos DB connection string.
- Create the Hero model.
- Create the Hero service to get Hero data.
- Run the app locally.

If you don't have an Azure subscription, create a free account before you begin.

Prerequisites

- Before you start this tutorial, complete the steps in Part 4.
- This tutorial requires that you run the Azure CLI locally. You must have the Azure CLI version 2.0 or later installed. Run `az --version` to find the version. If you need to install or upgrade the Azure CLI, see Install the Azure CLI 2.0.
- This tutorial walks you through the steps to build the application step by step. If you want to download the finished project, you can get the completed application from the angular-cosmosdb repo on GitHub.

Use Mongoose to connect

Mongoose is an object data modeling (ODM) library for MongoDB and Node.js. You can use Mongoose to connect to your Azure Cosmos DB account. Use the following steps to install Mongoose and connect to Azure Cosmos DB:

1. Install the mongoose npm module, which is an API that's used to talk to MongoDB.

   ```
   npm i mongoose --save
   ```

2. In the **server** folder, create a file named **mongo.js**. You'll add the connection details of your Azure Cosmos DB account to this file.

3. Copy the following code into the **mongo.js** file. The code provides the following functionality:

   - Requires Mongoose.
   - Overrides the Mongo promise to use the basic promise that's built into ES6/ES2015 and later versions.

- Calls on an env file that lets you set up certain things based on whether you're in staging, production, or development. You'll create that file in the next section.
- Includes the MongoDB connection string, which is set in the env file.
- Creates a connect function that calls Mongoose.

```
const mongoose = require('mongoose');
/**
* Set to Node.js native promises
* Per https://mongoosejs.com/docs/promises.html
*/
mongoose.Promise = global.Promise;

const env = require('./env/environment');

// eslint-disable-next-line max-len
const mongoUri =
`mongodb://${env.accountName}:${env.key}@${env.accountName}.documents.azure.com:${env.port}/${en
v.databaseName}?ssl=true`;

function connect() {
mongoose.set('debug', true);
return mongoose.connect(mongoUri, { useMongoClient: true });
}

module.exports = {
connect,
mongoose
};
```

4. In the Explorer pane, under **server**, create a folder named **environment**. In the **environment** folder, create a file named **environment.js**.
5. From the mongo.js file, we need to include values for the `dbName`, the `key`, and the `cosmosPort` parameters. Copy the following code into the **environment.js** file:

```
// TODO: replace if yours are different
module.exports = {
  accountName: 'your-cosmosdb-account-name-goes-here',
  databaseName: 'admin',
  key: 'your-key-goes-here',
  port: 10255
};
```

Get the connection string

To connect your application to Azure Cosmos DB, you need to update the configuration settings for the application. Use the following steps to update the settings:

1. In the Azure portal, get the port number, Azure Cosmos DB account name, and primary key values for your Azure Cosmos DB account.
2. In the **environment.js** file, change the value of `port` to 10255.

```
const port = 10255;
```

3. In the **environment.js** file, change the value of `accountName` to the name of the Azure Cosmos DB account that you created in Part 4 of the tutorial.

4. Retrieve the primary key for the Azure Cosmos DB account by using the following CLI command in the terminal window:

```
az cosmosdb list-keys --name <cosmosdb-name> -g myResourceGroup
```

   <cosmosdb-name> is the name of the Azure Cosmos DB account that you created in Part 4 of the tutorial.

5. Copy the primary key into the **environment.js** file as the `key` value.

Now your application has all the necessary information to connect to Azure Cosmos DB.

Create a Hero model

Next, you need to define the schema of the data to store in Azure Cosmos DB by defining a model file. Use the following steps to create a *Hero model* that defines the schema of the data:

1. In the Explorer pane, under the **server** folder, create a file named **hero.model.js**.

2. Copy the following code into the **hero.model.js** file. The code provides the following functionality:

   - Requires Mongoose.
   - Creates a new schema with an ID, name, and saying.
   - Creates a model by using the schema.
   - Exports the model.
   - Names the collection **Heroes** (instead of **Heros**, which is the default name of the collection based on Mongoose plural naming rules).

```
const mongoose = require('mongoose');

const Schema = mongoose.Schema;

const heroSchema = new Schema(
  {
    id: { type: Number, required: true, unique: true },
    name: String,
    saying: String
  },
  {
    collection: 'Heroes'
  }
);

const Hero = mongoose.model('Hero', heroSchema);

module.exports = Hero;
```

Create a Hero service

After you create the hero model, you need to define a service to read the data, and perform list, create, delete, and update operations. Use the following steps to create a *Hero service* that queries the data from Azure Cosmos DB:

1. In the Explorer pane, under the **server** folder, create a file named **hero.service.js**.

2. Copy the following code into the **hero.service.js** file. The code provides the following functionality:

   - Gets the model that you created.
   - Connects to the database.
   - Creates a `docquery` variable that uses the `hero.find` method to define a query that returns all heroes.

- Runs a query with the `docquery.exec` function with a promise to get a list of all heroes, where the response status is 200.
- Sends back the error message if the status is 500.
- Gets the heroes because we're using modules.

```
const Hero = require('./hero.model');

require('./mongo').connect();

function getHeroes() {
  const docquery = Hero.find({});
  docquery
    .exec()
    .then(heroes => {
      res.status(200).json(heroes);
    })
    .catch(error => {
      res.status(500).send(error);
      return;
    });
}

module.exports = {
  getHeroes
};
```

Configure routes

Next, you need to set up routes to handle the URLs for get, create, read, and delete requests. The routing methods specify callback functions (also called *handler functions*). These functions are called when the application receives a request to the specified endpoint and HTTP method. Use the following steps to add the Hero service and to define your routes:

1. In Visual Studio Code, in the **routes.js** file, comment out the `res.send` function that sends the sample hero data. Add a line to call the `heroService.getHeroes` function instead.

```
router.get('/heroes', (req, res) => {
  heroService.getHeroes(req, res);
//  res.send(200, [
//      {"id": 10, "name": "Starlord", "saying": "oh yeah"}
//  ])
});
```

2. In the **routes.js** file, `require` the hero service:

```
const heroService = require('./hero.service');
```

3. In the **hero.service.js** file, update the `getHeroes` function to take the `req` and `res` parameters as follows:

```
function getHeroes(req, res) {
```

Let's take a minute to review and walk through the previous code. First, we come into the index.js file, which sets up the node server. Notice that it sets up and defines your routes. Next, your routes.js file talks to the hero service and tells it to get your functions, like **getHeroes**, and pass the request and response. The hero.service.js file gets the model and connects to Mongo. Then it executes **getHeroes** when we call it, and returns back a response of 200.

Run the app

Next, run the app by using the following steps:

1. In Visual Studio Code, save all your changes. On the left, select the **Debug** button , and then select the **Start Debugging** button .

2. Now switch to the browser. Open the **Developer tools** and the **Network tab**. Go to `http://localhost:3000`, and there you see our application.

There are no heroes stored yet in the app. In the next part of this tutorial, we'll add put, push, and delete functionality. Then we can add, update, and delete heroes from the UI by using Mongoose connections to our Azure Cosmos database.

Clean up resources

When you no longer need the resources, you can delete the resource group, Azure Cosmos DB account, and all the related resources. Use the following steps to delete the resource group:

1. Go to the resource group where you created the Azure Cosmos DB account.
2. Select **Delete resource group**.
3. Confirm the name of the resource group to delete, and select **Delete**.

Next steps

Continue to Part 6 of the tutorial to add Post, Put, and Delete functions to the app:

Part 6: Add Post, Put, and Delete functions to the app

Create an Angular app with Azure Cosmos DB's API for MongoDB - Add CRUD functions to the app

11/2/2020 • 4 minutes to read • Edit Online

APPLIES TO: Azure Cosmos DB API for MongoDB

This multi-part tutorial demonstrates how to create a new app written in Node.js with Express and Angular and then connect it to your Cosmos account configured with Cosmos DB's API for MongoDB. Part 6 of the tutorial builds on Part 5 and covers the following tasks:

- Create Post, Put, and Delete functions for the hero service
- Run the app

Prerequisites

Before starting this part of the tutorial, ensure you've completed the steps in Part 5 of the tutorial.

> **TIP**
> This tutorial walks you through the steps to build the application step-by-step. If you want to download the finished project, you can get the completed application from the angular-cosmosdb repo on GitHub.

Add a Post function to the hero service

1. In Visual Studio Code, open **routes.js** and **hero.service.js** side by side by pressing the **Split Editor** button .

   See that routes.js line 7 is calling the `getHeroes` function on line 5 in **hero.service.js**. We need to create this same pairing for the post, put, and delete functions.

   routes.js

   ```
   1  const express = require('express');
   2  const router = express.Router();
   3
   4  const heroService = require('./hero.service');
   5
   6  router.get('/heroes', (req, res) => {
   7    heroService.getHeroes(req, res);
   8    // res.send(200, [
   9    //   { "id": 10, "name": "Starlord", "saying": "oh y
   10   // ])
   11 });
   12
   13 module.exports = router;
   ```

   hero.service.js

   ```
   1  const Hero = require('./hero.model');
   2
   3  require('./mongo').connect();
   4
   5  function getHeroes(req, res) {
   6    const docquery = Hero.find({});
   7    docquery
   8      .exec()
   9      .then(heroes => {
   10       res.status(200).json(heroes);
   11     })
   12     .catch(error => {
   13       res.status(500).send(error);
   14       return;
   15     });
   16 }
   ```

   Let's start by coding up the hero service.

2. Copy the following code into **hero.service.js** after the `getHeroes` function and before `module.exports`. This code:

- Uses the hero model to post a new hero.
- Checks the responses to see if there's an error and returns a status value of 500.

```
function postHero(req, res) {
  const originalHero = { uid: req.body.uid, name: req.body.name, saying: req.body.saying };
  const hero = new Hero(originalHero);
  hero.save(error => {
    if (checkServerError(res, error)) return;
    res.status(201).json(hero);
    console.log('Hero created successfully!');
  });
}

function checkServerError(res, error) {
  if (error) {
    res.status(500).send(error);
    return error;
  }
}
```

3. In **hero.service.js**, update the `module.exports` to include the new `postHero` function.

```
module.exports = {
  getHeroes,
  postHero
};
```

4. In **routes.js**, add a router for the `post` function after the `get` router. This router posts one hero at a time. Structuring the router file this way cleanly shows you all of the available API endpoints and leaves the real work to the **hero.service.js** file.

```
router.post('/hero', (req, res) => {
  heroService.postHero(req, res);
});
```

5. Check that everything worked by running the app. In Visual Studio Code, save all your changes, select the **Debug** button on the left side, then select the **Start Debugging** button .

6. Now go back to your internet browser and open the Developer tools Network tab by pressing F12 on most machines. Navigate to `http://localhost:3000` to watch the calls made over the network.

7. Add a new hero by selecting the **Add New Hero** button. Enter an ID of "999", name of "Fred", and saying of "Hello", then select **Save**. You should see in the Networking tab you've sent a POST request for a new hero.

Now let's go back and add the Put and Delete functions to the app.

Add the Put and Delete functions

1. In **routes.js**, add the `put` and `delete` routers after the post router.

```
router.put('/hero/:uid', (req, res) => {
  heroService.putHero(req, res);
});

router.delete('/hero/:uid', (req, res) => {
  heroService.deleteHero(req, res);
});
```

2. Copy the following code into **hero.service.js** after the `checkServerError` function. This code:

   - Creates the `put` and `delete` functions
   - Performs a check on whether the hero was found
   - Performs error handling

```
function putHero(req, res) {
  const originalHero = {
    uid: parseInt(req.params.uid, 10),
    name: req.body.name,
    saying: req.body.saying
  };
  Hero.findOne({ uid: originalHero.uid }, (error, hero) => {
    if (checkServerError(res, error)) return;
    if (!checkFound(res, hero)) return;

   hero.name = originalHero.name;
    hero.saying = originalHero.saying;
    hero.save(error => {
      if (checkServerError(res, error)) return;
      res.status(200).json(hero);
      console.log('Hero updated successfully!');
    });
  });
}

function deleteHero(req, res) {
  const uid = parseInt(req.params.uid, 10);
  Hero.findOneAndRemove({ uid: uid })
    .then(hero => {
      if (!checkFound(res, hero)) return;
      res.status(200).json(hero);
      console.log('Hero deleted successfully!');
    })
    .catch(error => {
      if (checkServerError(res, error)) return;
    });
}

function checkFound(res, hero) {
  if (!hero) {
    res.status(404).send('Hero not found.');
    return;
  }
  return hero;
}
```

3. In **hero.service.js**, export the new modules:

```
 module.exports = {
   getHeroes,
   postHero,
   putHero,
   deleteHero
 };
```

4. Now that we've updated the code, select the **Restart** button in Visual Studio Code.

5. Refresh the page in your internet browser and select the **Add New Hero** button. Add a new hero with an ID of "9", name of "Starlord", and saying "Hi". Select the **Save** button to save the new hero.

6. Now select the **Starlord** hero, and change the saying from "Hi" to "Bye", then select the **Save** button.

   You can now select the ID in the Network tab to show the payload. You can see in the payload that the saying is now set to "Bye".

You can also delete one of the heroes in the UI, and see the times it takes to complete the delete operation. Try this out by selecting the "Delete" button for the hero named "Fred".

If you refresh the page, the Network tab shows the time it takes to get the heroes. While these times are fast, a lot depends on where your data is located in the world and your ability to geo-replicate it in an area close to your users. You can find out more about geo-replication in the next, soon to be released, tutorial.

Next steps

In this part of the tutorial, you've done the following:

- Added Post, Put, and Delete functions to the app

Check back soon for additional videos in this tutorial series.

Create a MongoDB app with React and Azure Cosmos DB

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: Azure Cosmos DB API for MongoDB

This multi-part video tutorial demonstrates how to create a hero tracking app with a React front-end. The app used Node and Express for the server, connects to Cosmos database configured with the Azure Cosmos DB's API for MongoDB, and then connects the React front-end to the server portion of the app. The tutorial also demonstrates how to do point-and-click scaling of Cosmos DB in the Azure portal and how to deploy the app to the internet so everyone can track their favorite heroes.

Azure Cosmos DB supports wire protocol compatibility with MongoDB, enabling clients to use Azure Cosmos DB in place of MongoDB.

This multi-part tutorial covers the following tasks:

- Introduction
- Setup the project
- Build the UI with React
- Create an Azure Cosmos DB account using the Azure portal
- Use Mongoose to connect to Azure Cosmos DB
- Add React, Create, Update, and Delete operations to the app

Want to do build this same app with Angular? See the Angular tutorial video series.

Prerequisites

- Node.js

Finished Project

Get the completed application from GitHub.

Introduction

In this video, Burke Holland gives an introduction to Azure Cosmos DB and walks you through the app that is created in this video series.

Project setup

This video shows how set up the Express and React in the same project. Burke then provides a walkthrough of the code in the project.

Build the UI

This video shows how to create the application's user interface (UI) with React.

> **NOTE**
> The CSS referenced in this video can be found in the react-cosmosdb GitHub repo.

Connect to Azure Cosmos DB

This video shows how to create an Azure Cosmos DB account in the Azure portal, install the MongoDB and Mongoose packages, and then connect the app to the newly created account using the Azure Cosmos DB connection string.

Read and create heroes in the app

This video shows how to read heroes and create heroes in the Cosmos database, as well as how to test those methods using Postman and the React UI.

Delete and update heroes in the app

This video shows how to delete and update heroes from the app and display the updates in the UI.

Complete the app

This video shows how to complete the app and finish hooking the UI up to the backend API.

Clean up resources

If you're not going to continue to use this app, use the following steps to delete all resources created by this tutorial in the Azure portal.

1. From the left-hand menu in the Azure portal, click **Resource groups** and then click the name of the resource you created.
2. On your resource group page, click **Delete**, type the name of the resource to delete in the text box, and then click **Delete**.

Next steps

In this tutorial, you've learned how to:

- Create an app with React, Node, Express, and Azure Cosmos DB
- Create an Azure Cosmos DB account
- Connect the app to the Azure Cosmos DB account
- Test the app using Postman
- Run the application and add heroes to the database

You can proceed to the next tutorial and learn how to import MongoDB data into Azure Cosmos DB.

Query data by using Azure Cosmos DB's API for MongoDB

11/2/2020 • 3 minutes to read • Edit Online

APPLIES TO: Azure Cosmos DB API for MongoDB

The Azure Cosmos DB's API for MongoDB supports MongoDB queries.

This article covers the following tasks:

- Querying data stored in your Cosmos database using MongoDB shell

You can get started by using the examples in this document and watch the Query Azure Cosmos DB with MongoDB shell video .

Sample document

The queries in this article use the following sample document.

```
{
  "id": "WakefieldFamily",
  "parents": [
      { "familyName": "Wakefield", "givenName": "Robin" },
      { "familyName": "Miller", "givenName": "Ben" }
  ],
  "children": [
      {
        "familyName": "Merriam",
        "givenName": "Jesse",
        "gender": "female", "grade": 1,
        "pets": [
            { "givenName": "Goofy" },
            { "givenName": "Shadow" }
        ]
      },
      {
        "familyName": "Miller",
         "givenName": "Lisa",
         "gender": "female",
         "grade": 8 }
  ],
  "address": { "state": "NY", "county": "Manhattan", "city": "NY" },
  "creationDate": 1431620462,
  "isRegistered": false
}
```

Example query 1

Given the sample family document above, the following query returns the documents where the id field matches `WakefieldFamily`.

Query

```
db.families.find({ id: "WakefieldFamily"})
```

Results

```
{
    "_id": "ObjectId(\"58f65e1198f3a12c7090e68c\")",
    "id": "WakefieldFamily",
    "parents": [
      {
        "familyName": "Wakefield",
        "givenName": "Robin"
      },
      {
        "familyName": "Miller",
        "givenName": "Ben"
      }
    ],
    "children": [
      {
        "familyName": "Merriam",
        "givenName": "Jesse",
        "gender": "female",
        "grade": 1,
        "pets": [
          { "givenName": "Goofy" },
          { "givenName": "Shadow" }
        ]
      },
      {
        "familyName": "Miller",
        "givenName": "Lisa",
        "gender": "female",
        "grade": 8
      }
    ],
    "address": {
      "state": "NY",
      "county": "Manhattan",
      "city": "NY"
    },
    "creationDate": 1431620462,
    "isRegistered": false
}
```

Example query 2

The next query returns all the children in the family.

Query

```
db.families.find( { id: "WakefieldFamily" }, { children: true } )
```

Results

```
{
    "_id": "ObjectId("58f65e1198f3a12c7090e68c")",
    "children": [
      {
        "familyName": "Merriam",
        "givenName": "Jesse",
        "gender": "female",
        "grade": 1,
        "pets": [
          { "givenName": "Goofy" },
          { "givenName": "Shadow" }
        ]
      },
      {
        "familyName": "Miller",
        "givenName": "Lisa",
        "gender": "female",
        "grade": 8
      }
    ]
}
```

Example query 3

The next query returns all the families that are registered.

Query

```
db.families.find( { "isRegistered" : true })
```

Results

No document will be returned.

Example query 4

The next query returns all the families that are not registered.

Query

```
db.families.find( { "isRegistered" : false })
```

Results

```
{
 "_id": ObjectId("58f65e1198f3a12c7090e68c"),
 "id": "WakefieldFamily",
 "parents": [{
  "familyName": "Wakefield",
  "givenName": "Robin"
 }, {
  "familyName": "Miller",
  "givenName": "Ben"
 }],
 "children": [{
  "familyName": "Merriam",
  "givenName": "Jesse",
  "gender": "female",
  "grade": 1,
  "pets": [{
   "givenName": "Goofy"
  }, {
   "givenName": "Shadow"
  }]
 }, {
  "familyName": "Miller",
  "givenName": "Lisa",
  "gender": "female",
  "grade": 8
 }],
 "address": {
  "state": "NY",
  "county": "Manhattan",
  "city": "NY"
 },
 "creationDate": 1431620462,
 "isRegistered": false
}
```

Example query 5

The next query returns all the families that are not registered and state is NY.

Query

```
db.families.find( { "isRegistered" : false, "address.state" : "NY" })
```

Results

```
{
 "_id": ObjectId("58f65e1198f3a12c7090e68c"),
 "id": "WakefieldFamily",
 "parents": [{
  "familyName": "Wakefield",
  "givenName": "Robin"
 }, {
  "familyName": "Miller",
  "givenName": "Ben"
 }],
 "children": [{
  "familyName": "Merriam",
  "givenName": "Jesse",
  "gender": "female",
  "grade": 1,
  "pets": [{
   "givenName": "Goofy"
  }, {
   "givenName": "Shadow"
  }]
 }, {
  "familyName": "Miller",
  "givenName": "Lisa",
  "gender": "female",
  "grade": 8
 }],
 "address": {
  "state": "NY",
  "county": "Manhattan",
  "city": "NY"
 },
 "creationDate": 1431620462,
 "isRegistered": false
}
```

Example query 6

The next query returns all the families where children grades are 8.

Query

```
db.families.find( { children : { $elemMatch: { grade : 8 }} } )
```

Results

```
{
 "_id": ObjectId("58f65e1198f3a12c7090e68c"),
 "id": "WakefieldFamily",
 "parents": [{
  "familyName": "Wakefield",
  "givenName": "Robin"
 }, {
  "familyName": "Miller",
  "givenName": "Ben"
 }],
 "children": [{
  "familyName": "Merriam",
  "givenName": "Jesse",
  "gender": "female",
  "grade": 1,
  "pets": [{
   "givenName": "Goofy"
  }, {
   "givenName": "Shadow"
  }]
 }, {
  "familyName": "Miller",
  "givenName": "Lisa",
  "gender": "female",
  "grade": 8
 }],
 "address": {
  "state": "NY",
  "county": "Manhattan",
  "city": "NY"
 },
 "creationDate": 1431620462,
 "isRegistered": false
}
```

Example query 7

The next query returns all the families where size of children array is 3.

Query

```
db.Family.find( {children: { $size:3} } )
```

Results

No results will be returned as there are no families with more than two children. Only when parameter is 2 this query will succeed and return the full document.

Next steps

In this tutorial, you've done the following:

- Learned how to query using Cosmos DB's API for MongoDB

You can now proceed to the next tutorial to learn how to distribute your data globally.

Distribute your data globally

Set up global distributed database using Azure Cosmos DB's API for MongoDB

11/2/2020 • 3 minutes to read • Edit Online

APPLIES TO: Azure Cosmos DB API for MongoDB

In this article, we show how to use the Azure portal to setup a global distributed database and connect to it using Azure Cosmos DB's API for MongoDB.

This article covers the following tasks:

- Configure global distribution using the Azure portal
- Configure global distribution using the Azure Cosmos DB's API for MongoDB

Add global database regions using the Azure portal

Azure Cosmos DB is available in all Azure regions worldwide. After selecting the default consistency level for your database account, you can associate one or more regions (depending on your choice of default consistency level and global distribution needs).

1. In the Azure portal, in the left bar, click **Azure Cosmos DB**.
2. In the **Azure Cosmos DB** page, select the database account to modify.
3. In the account page, click **Replicate data globally** from the menu.
4. In the **Replicate data globally** page, select the regions to add or remove by clicking regions in the map, and then click **Save**. There is a cost to adding regions, see the pricing page or the Distribute data globally with Azure Cosmos DB article for more information.

Once you add a second region, the **Manual Failover** option is enabled on the **Replicate data globally** page in the portal. You can use this option to test the failover process or change the primary write region. Once you add a third region, the **Failover Priorities** option is enabled on the same page so that you can change the failover order for reads.

Selecting global database regions

There are two common scenarios for configuring two or more regions:

1. Delivering low-latency access to data to end users no matter where they are located around the globe
2. Adding regional resiliency for business continuity and disaster recovery (BCDR)

For delivering low-latency to end users, it is recommended that you deploy both the application and Azure Cosmos DB in the regions that correspond to where the application's users are located.

For BCDR, it is recommended to add regions based on the region pairs described in the Business continuity and disaster recovery (BCDR): Azure Paired Regions article.

Verifying your regional setup

A simple way to check your global configuration with Cosmos DB's API for MongoDB is to run the *isMaster()* command from the Mongo Shell.

From your Mongo Shell:

```
db.isMaster()
```

Example results:

```
{
   "_t": "IsMasterResponse",
   "ok": 1,
   "ismaster": true,
   "maxMessageSizeBytes": 4194304,
   "maxWriteBatchSize": 1000,
   "minWireVersion": 0,
   "maxWireVersion": 2,
   "tags": {
      "region": "South India"
   },
   "hosts": [
      "vishi-api-for-mongodb-southcentralus.documents.azure.com:10255",
      "vishi-api-for-mongodb-westeurope.documents.azure.com:10255",
      "vishi-api-for-mongodb-southindia.documents.azure.com:10255"
   ],
   "setName": "globaldb",
   "setVersion": 1,
   "primary": "vishi-api-for-mongodb-southindia.documents.azure.com:10255",
   "me": "vishi-api-for-mongodb-southindia.documents.azure.com:10255"
}
```

Connecting to a preferred region

The Azure Cosmos DB's API for MongoDB enables you to specify your collection's read preference for a globally distributed database. For both low latency reads and global high availability, we recommend setting your collection's read preference to *nearest*. A read preference of *nearest* is configured to read from the closest region.

```
var collection = database.GetCollection<BsonDocument>(collectionName);
collection = collection.WithReadPreference(new ReadPreference(ReadPreferenceMode.Nearest));
```

For applications with a primary read/write region and a secondary region for disaster recovery (DR) scenarios, we recommend setting your collection's read preference to *secondary preferred*. A read preference of *secondary preferred* is configured to read from the secondary region when the primary region is unavailable.

```
var collection = database.GetCollection<BsonDocument>(collectionName);
collection = collection.WithReadPreference(new ReadPreference(ReadPreferenceMode.SecondaryPreferred));
```

Lastly, if you would like to manually specify your read regions. You can set the region Tag within your read preference.

```
var collection = database.GetCollection<BsonDocument>(collectionName);
var tag = new Tag("region", "Southeast Asia");
collection = collection.WithReadPreference(new ReadPreference(ReadPreferenceMode.Secondary, new[] { new
TagSet(new[] { tag }) }));
```

That's it, that completes this tutorial. You can learn how to manage the consistency of your globally replicated account by reading Consistency levels in Azure Cosmos DB. And for more information about how global database replication works in Azure Cosmos DB, see Distribute data globally with Azure Cosmos DB.

Next steps

In this tutorial, you've done the following:

- Configure global distribution using the Azure portal
- Configure global distribution using the Cosmos DB's API for MongoDB

You can now proceed to the next tutorial to learn how to develop locally using the Azure Cosmos DB local emulator.

Develop locally with the Azure Cosmos DB emulator

Azure Cosmos DB's API for MongoDB (3.6 version): supported features and syntax

12/4/2020 • 8 minutes to read • Edit Online

APPLIES TO: Azure Cosmos DB API for MongoDB

Azure Cosmos DB is Microsoft's globally distributed multi-model database service. You can communicate with the Azure Cosmos DB's API for MongoDB using any of the open-source MongoDB client drivers. The Azure Cosmos DB's API for MongoDB enables the use of existing client drivers by adhering to the MongoDB wire protocol.

By using the Azure Cosmos DB's API for MongoDB, you can enjoy the benefits of the MongoDB you're used to, with all of the enterprise capabilities that Cosmos DB provides: global distribution, automatic sharding, availability and latency guarantees, encryption at rest, backups, and much more.

Protocol Support

The Azure Cosmos DB's API for MongoDB is compatible with MongoDB server version **3.6** by default for new accounts. The supported operators and any limitations or exceptions are listed below. Any client driver that understands these protocols should be able to connect to Azure Cosmos DB's API for MongoDB. Note that when using Azure Cosmos DB's API for MongoDB accounts, the 3.6 version of accounts have the endpoint in the format `*.mongo.cosmos.azure.com` whereas the 3.2 version of accounts have the endpoint in the format `*.documents.azure.com`.

Query language support

Azure Cosmos DB's API for MongoDB provides comprehensive support for MongoDB query language constructs. Below you can find the detailed list of currently supported operations, operators, stages, commands, and options.

Database commands

Azure Cosmos DB's API for MongoDB supports the following database commands:

Query and write operation commands

COMMAND	SUPPORTED
delete	Yes
find	Yes
findAndModify	Yes
getLastError	Yes
getMore	Yes
getPrevError	No
insert	Yes

COMMAND	SUPPORTED
parallelCollectionScan	Yes
resetError	No
update	Yes
Change streams	Yes
GridFS	Yes

Authentication commands

COMMAND	SUPPORTED
authenticate	Yes
logout	Yes
getnonce	Yes

Administration commands

COMMAND	SUPPORTED
Capped Collections	No
cloneCollectionAsCapped	No
collMod	No
collMod: expireAfterSeconds	No
convertToCapped	No
copydb	No
create	Yes
createIndexes	Yes
currentOp	Yes
drop	Yes
dropDatabase	Yes
dropIndexes	Yes
filemd5	Yes
killCursors	Yes

COMMAND	SUPPORTED
killOp	No
listCollections	Yes
listDatabases	Yes
listIndexes	Yes
reIndex	Yes
renameCollection	No
connectionStatus	No

Diagnostics commands

COMMAND	SUPPORTED
buildInfo	Yes
collStats	Yes
connPoolStats	No
connectionStatus	No
dataSize	No
dbHash	No
dbStats	Yes
explain	Yes
explain: executionStats	Yes
features	No
hostInfo	No
listDatabases	Yes
listCommands	No
profiler	No
serverStatus	No
top	No

COMMAND	SUPPORTED
whatsmyuri	Yes

Aggregation pipeline

Aggregation commands

COMMAND	SUPPORTED
aggregate	Yes
count	Yes
distinct	Yes
mapReduce	No

Aggregation stages

COMMAND	SUPPORTED
$collStats	No
$project	Yes
$match	Yes
$redact	Yes
$limit	Yes
$skip	Yes
$unwind	Yes
$group	Yes
$sample	Yes
$sort	Yes
$geoNear	No
$lookup	Yes
$out	Yes
$indexStats	No
$facet	Yes

COMMAND	SUPPORTED
$bucket	No
$bucketAuto	No
$sortByCount	Yes
$addFields	Yes
$replaceRoot	Yes
$count	Yes
$currentOp	No
$listLocalSessions	No
$listSessions	No
$graphLookup	Yes

Boolean expressions

COMMAND	SUPPORTED
$and	Yes
$or	Yes
$not	Yes

Set expressions

COMMAND	SUPPORTED
$setEquals	Yes
$setIntersection	Yes
$setUnion	Yes
$setDifference	Yes
$setIsSubset	Yes
$anyElementTrue	Yes
$allElementsTrue	Yes

Comparison expressions

COMMAND	SUPPORTED
$cmp	Yes
$eq	Yes
$gt	Yes
$gte	Yes
$lt	Yes
$lte	Yes
$ne	Yes
$in	Yes
$nin	Yes

Arithmetic expressions

COMMAND	SUPPORTED
$abs	Yes
$add	Yes
$ceil	Yes
$divide	Yes
$exp	Yes
$floor	Yes
$ln	Yes
$log	Yes
$log10	Yes
$mod	Yes
$multiply	Yes
$pow	Yes
$sqrt	Yes
$subtract	Yes

COMMAND	SUPPORTED
$trunc	Yes

String expressions

COMMAND	SUPPORTED
$concat	Yes
$indexOfBytes	Yes
$indexOfCP	Yes
$split	Yes
$strLenBytes	Yes
$strLenCP	Yes
$strcasecmp	Yes
$substr	Yes
$substrBytes	Yes
$substrCP	Yes
$toLower	Yes
$toUpper	Yes

Text search operator

COMMAND	SUPPORTED
$meta	No

Array expressions

COMMAND	SUPPORTED
$arrayElemAt	Yes
$arrayToObject	Yes
$concatArrays	Yes
$filter	Yes
$indexOfArray	Yes
$isArray	Yes

COMMAND	SUPPORTED
$objectToArray	Yes
$range	Yes
$reverseArray	Yes
$reduce	Yes
$size	Yes
$slice	Yes
$zip	Yes
$in	Yes

Variable operators

COMMAND	SUPPORTED
$map	No
$let	Yes

System variables

COMMAND	SUPPORTED
$$CURRENT	Yes
$$DESCEND	Yes
$$KEEP	Yes
$$PRUNE	Yes
$$REMOVE	Yes
$$ROOT	Yes

Literal operator

COMMAND	SUPPORTED
$literal	Yes

Date expressions

COMMAND	SUPPORTED
$dayOfYear	Yes

COMMAND	SUPPORTED
$dayOfMonth	Yes
$dayOfWeek	Yes
$year	Yes
$month	Yes
$week	Yes
$hour	Yes
$minute	Yes
$second	Yes
$millisecond	Yes
$dateToString	Yes
$isoDayOfWeek	Yes
$isoWeek	Yes
$dateFromParts	No
$dateToParts	No
$dateFromString	No
$isoWeekYear	Yes

Conditional expressions

COMMAND	SUPPORTED
$cond	Yes
$ifNull	Yes
$switch	Yes

Data type operator

COMMAND	SUPPORTED
$type	Yes

Accumulator expressions

COMMAND	SUPPORTED
$sum	Yes
$avg	Yes
$first	Yes
$last	Yes
$max	Yes
$min	Yes
$push	Yes
$addToSet	Yes
$stdDevPop	No
$stdDevSamp	No

Merge operator

COMMAND	SUPPORTED
$mergeObjects	Yes

Data types

COMMAND	SUPPORTED
Double	Yes
String	Yes
Object	Yes
Array	Yes
Binary Data	Yes
ObjectId	Yes
Boolean	Yes
Date	Yes
Null	Yes
32-bit Integer (int)	Yes

COMMAND	SUPPORTED
Timestamp	Yes
64-bit Integer (long)	Yes
MinKey	Yes
MaxKey	Yes
Decimal128	Yes
Regular Expression	Yes
JavaScript	Yes
JavaScript (with scope)	Yes
Undefined	Yes

Indexes and index properties

Indexes

COMMAND	SUPPORTED
Single Field Index	Yes
Compound Index	Yes
Multikey Index	Yes
Text Index	No
2dsphere	Yes
2d Index	No
Hashed Index	Yes

Index properties

COMMAND	SUPPORTED
TTL	Yes
Unique	Yes
Partial	No
Case Insensitive	No

COMMAND	SUPPORTED
Sparse	No
Background	Yes

Operators

Logical operators

COMMAND	SUPPORTED
$or	Yes
$and	Yes
$not	Yes
$nor	Yes

Element operators

COMMAND	SUPPORTED
$exists	Yes
$type	Yes

Evaluation query operators

COMMAND	SUPPORTED
$expr	No
$jsonSchema	No
$mod	Yes
$regex	Yes
$text	No (Not supported. Use $regex instead.)
$where	No

In the $regex queries, left-anchored expressions allow index search. However, using 'i' modifier (case-insensitivity) and 'm' modifier (multiline) causes the collection scan in all expressions.

When there's a need to include '$' or '|', it is best to create two (or more) regex queries. For example, given the following original query: `find({x:{$regex: /^abc$/})`, it has to be modified as follows:

`find({x:{$regex: /^abc/, x:{$regex:/^abc$/}})`.

The first part will use the index to restrict the search to those documents beginning with ^abc and the second part will match the exact entries. The bar operator '|' acts as an "or" function - the query

`find({x:{$regex: /^abc|^def/})` matches the documents in which field 'x' has values that begin with "abc" or "def". To utilize the index, it's recommended to break the query into two different queries joined by the $or operator: `find( {$or : [{x: $regex: /^abc/}, {$regex: /^def/}] })`.

Array operators

COMMAND	SUPPORTED
$all	Yes
$elemMatch	Yes
$size	Yes

Comment operator

COMMAND	SUPPORTED
$comment	Yes

Projection operators

COMMAND	SUPPORTED
$elemMatch	Yes
$meta	No
$slice	Yes

Update operators

Field update operators

COMMAND	SUPPORTED
$inc	Yes
$mul	Yes
$rename	Yes
$setOnInsert	Yes
$set	Yes
$unset	Yes
$min	Yes
$max	Yes
$currentDate	Yes

Array update operators

COMMAND	SUPPORTED
$	Yes
$[]	Yes
$[]	Yes
$addToSet	Yes
$pop	Yes
$pullAll	Yes
$pull	Yes
$push	Yes
$pushAll	Yes

Update modifiers

COMMAND	SUPPORTED
$each	Yes
$slice	Yes
$sort	Yes
$position	Yes

Bitwise update operator

COMMAND	SUPPORTED
$bit	Yes
$bitsAllSet	No
$bitsAnySet	No
$bitsAllClear	No
$bitsAnyClear	No

Geospatial operators

OPERATOR	SUPPORTED
$geoWithin	Yes
$geoIntersects	Yes

OPERATOR	SUPPORTED
$near	Yes
$nearSphere	Yes
$geometry	Yes
$minDistance	Yes
$maxDistance	Yes
$center	No
$centerSphere	No
$box	No
$polygon	No

Cursor methods

COMMAND	SUPPORTED
cursor.batchSize()	Yes
cursor.close()	Yes
cursor.isClosed()	Yes
cursor.collation()	No
cursor.comment()	Yes
cursor.count()	Yes
cursor.explain()	No
cursor.forEach()	Yes
cursor.hasNext()	Yes
cursor.hint()	Yes
cursor.isExhausted()	Yes
cursor.itcount()	Yes
cursor.limit()	Yes
cursor.map()	Yes

COMMAND	SUPPORTED
cursor.maxScan()	Yes
cursor.maxTimeMS()	Yes
cursor.max()	Yes
cursor.min()	Yes
cursor.next()	Yes
cursor.noCursorTimeout()	No
cursor.objsLeftInBatch()	Yes
cursor.pretty()	Yes
cursor.readConcern()	Yes
cursor.readPref()	Yes
cursor.returnKey()	No
cursor.showRecordId()	No
cursor.size()	Yes
cursor.skip()	Yes
cursor.sort()	Yes
cursor.tailable()	No
cursor.toArray()	Yes

Sort operations

When using the `findOneAndUpdate` operation, sort operations on a single field are supported but sort operations on multiple fields are not supported.

Unique indexes

Unique indexes ensure that a specific field doesn't have duplicate values across all documents in a collection, similar to the way uniqueness is preserved on the default "_id" key. You can create unique indexes in Cosmos DB by using the `createIndex` command with the `unique` constraint parameter:

```
globaldb:PRIMARY> db.coll.createIndex( { "amount" : 1 }, {unique:true} )
{
        "_t" : "CreateIndexesResponse",
        "ok" : 1,
        "createdCollectionAutomatically" : false,
        "numIndexesBefore" : 1,
        "numIndexesAfter" : 4
}
```

Compound indexes

Compound indexes provide a way to create an index for groups of fields for up to 8 fields. This type of index differs from the native MongoDB compound indexes. In Azure Cosmos DB, compound indexes are used for sorting operations that are applied to multiple fields. To create a compound index you need to specify more than one property as the parameter:

```
globaldb:PRIMARY> db.coll.createIndex({"amount": 1, "other":1})
{
        "createdCollectionAutomatically" : false,
        "numIndexesBefore" : 1,
        "numIndexesAfter" : 2,
        "ok" : 1
}
```

Time-to-live (TTL)

Cosmos DB supports a time-to-live (TTL) based on the timestamp of the document. TTL can be enabled for collections by going to the Azure portal.

User and role management

Cosmos DB does not yet support users and roles. However, Cosmos DB supports Azure role-based access control (Azure RBAC) and read-write and read-only passwords/keys that can be obtained through the Azure portal (Connection String page).

Replication

Cosmos DB supports automatic, native replication at the lowest layers. This logic is extended out to achieve low-latency, global replication as well. Cosmos DB does not support manual replication commands.

Write Concern

Some applications rely on a Write Concern which specifies the number of responses required during a write operation. Due to how Cosmos DB handles replication in the background all writes are all automatically Quorum by default. Any write concern specified by the client code is ignored. Learn more in Using consistency levels to maximize availability and performance.

Sharding

Azure Cosmos DB supports automatic, server-side sharding. It manages shard creation, placement, and balancing automatically. Azure Cosmos DB does not support manual sharding commands, which means you don't have to invoke commands such as addShard, balancerStart, moveChunk etc. You only need to specify the shard key while creating the containers or querying the data.

Sessions

Azure Cosmos DB does not yet support server side sessions commands.

Next steps

- For further information check Mongo 3.6 version features
- Learn how to use Studio 3T with Azure Cosmos DB's API for MongoDB.
- Learn how to use Robo 3T with Azure Cosmos DB's API for MongoDB.
- Explore MongoDB samples with Azure Cosmos DB's API for MongoDB.

Note: This article describes a feature of Azure Cosmos DB that provides wire protocol compatibility with MongoDB databases. Microsoft does not run MongoDB databases to provide this service. Azure Cosmos DB is not affiliated with MongoDB, Inc.

Azure Cosmos DB's API for MongoDB (3.2 version): supported features and syntax

11/2/2020 • 7 minutes to read • Edit Online

APPLIES TO: Azure Cosmos DB API for MongoDB

Azure Cosmos DB is Microsoft's globally distributed multi-model database service. You can communicate with the Azure Cosmos DB's API for MongoDB using any of the open source MongoDB client drivers. The Azure Cosmos DB's API for MongoDB enables the use of existing client drivers by adhering to the MongoDB wire protocol.

By using the Azure Cosmos DB's API for MongoDB, you can enjoy the benefits of the MongoDB you're used to, with all of the enterprise capabilities that Cosmos DB provides: global distribution, automatic sharding, availability and latency guarantees, automatic indexing of every field, encryption at rest, backups, and much more.

> **NOTE**
>
> This article is for Azure Cosmos DB's API for MongoDB 3.2. For MongoDB 3.6 version, see MongoDB 3.6 supported features and syntax.

Protocol Support

All new accounts for Azure Cosmos DB's API for MongoDB are compatible with MongoDB server version **3.6**. This article covers MongoDB version 3.2. The supported operators and any limitations or exceptions are listed below. Any client driver that understands these protocols should be able to connect to Azure Cosmos DB's API for MongoDB.

Azure Cosmos DB's API for MongoDB also offers a seamless upgrade experience for qualifying accounts. Learn more on the MongoDB version upgrade guide.

Query language support

Azure Cosmos DB's API for MongoDB provides comprehensive support for MongoDB query language constructs. Below you can find the detailed list of currently supported operations, operators, stages, commands and options.

Database commands

Azure Cosmos DB's API for MongoDB supports the following database commands:

Query and write operation commands

- delete
- find
- findAndModify
- getLastError
- getMore
- insert
- update

Authentication commands

- logout

- authenticate
- getnonce

Administration commands

- dropDatabase
- listCollections
- drop
- create
- filemd5
- createIndexes
- listIndexes
- dropIndexes
- connectionStatus
- reIndex

Diagnostics commands

- buildInfo
- collStats
- dbStats
- hostInfo
- listDatabases
- whatsmyuri

Aggregation pipeline

Cosmos DB supports aggregation pipeline for MongoDB 3.2 in public preview. See the Azure blog for instructions on how to onboard to the public preview.

Aggregation commands

- aggregate
- count
- distinct

Aggregation stages

- $project
- $match
- $limit
- $skip
- $unwind
- $group
- $sample
- $sort
- $lookup
- $out
- $count
- $addFields

Aggregation expressions

Boolean expressions

- $and

- $or
- $not

Set expressions

- $setEquals
- $setIntersection
- $setUnion
- $setDifference
- $setIsSubset
- $anyElementTrue
- $allElementsTrue

Comparison expressions

- $cmp
- $eq
- $gt
- $gte
- $lt
- $lte
- $ne

Arithmetic expressions

- $abs
- $add
- $ceil
- $divide
- $exp
- $floor
- $ln
- $log
- $log10
- $mod
- $multiply
- $pow
- $sqrt
- $subtract
- $trunc

String expressions

- $concat
- $indexOfBytes
- $indexOfCP
- $split
- $strLenBytes
- $strLenCP
- $strcasecmp
- $substr
- $substrBytes
- $substrCP

- $toLower
- $toUpper

Array expressions

- $arrayElemAt
- $concatArrays
- $filter
- $indexOfArray
- $isArray
- $range
- $reverseArray
- $size
- $slice
- $in

Date expressions

- $dayOfYear
- $dayOfMonth
- $dayOfWeek
- $year
- $month
- $week
- $hour
- $minute
- $second
- $millisecond
- $isoDayOfWeek
- $isoWeek

Conditional expressions

- $cond
- $ifNull

Aggregation accumulators

- $sum
- $avg
- $first
- $last
- $max
- $min
- $push
- $addToSet

Operators

Following operators are supported with corresponding examples of their use. Consider this sample document used in the queries below:

```
{
  "Volcano Name": "Rainier",
  "Country": "United States",
  "Region": "US-Washington",
  "Location": {
    "type": "Point",
    "coordinates": [
      -121.758,
      46.87
    ]
  },
  "Elevation": 4392,
  "Type": "Stratovolcano",
  "Status": "Dendrochronology",
  "Last Known Eruption": "Last known eruption from 1800-1899, inclusive"
}
```

OPERATOR	EXAMPLE		
$eq	`{ "Volcano Name": { $eq: "Rainier" } }`		-
$gt	`{ "Elevation": { $gt: 4000 } }`		-
$gte	`{ "Elevation": { $gte: 4392 } }`		-
$lt	`{ "Elevation": { $lt: 5000 } }`		-
$lte	`{ "Elevation": { $lte: 5000 } }`		-
$ne	`{ "Elevation": { $ne: 1 } }`		-
$in	`{ "Volcano Name": { $in: ["St. Helens", "Rainier", "Glacier Peak"] } }`		-
$nin	`{ "Volcano Name": { $nin: ["Lassen Peak", "Hood", "Baker"] } }`		-
$or	`{ $or: [ { Elevation: { $lt: 4000 } }, { "Volcano Name": "Rainier" } ] }`		-
$and	`{ $and: [ { Elevation: { $gt: 4000 } }, { "Volcano Name": "Rainier" } ] }`		-
$not	`{ "Elevation": { $not: { $gt: 5000 } } }`		-

OPERATOR	EXAMPLE		
$nor	`{ $nor: [ { "Elevation": { $lt: 4000 } }, { "Volcano Name": "Baker" } ] }`		-
$exists	`{ "Status": { $exists: true } }`		-
$type	`{ "Status": { $type: "string" } }`		-
$mod	`{ "Elevation": { $mod: [ 4, 0 ] } }`		-
$regex	`{ "Volcano Name": { $regex: "^Rain"} }`		-

Notes

In $regex queries, Left-anchored expressions allow index search. However, using 'i' modifier (case-insensitivity) and 'm' modifier (multiline) causes the collection scan in all expressions. When there's a need to include '$' or '|', it is best to create two (or more) regex queries. For example, given the following original query: `find({x:{$regex: /^abc$/})`, it has to be modified as follows: `find({x:{$regex: /^abc/, x:{$regex:/^abc$/}})`. The first part will use the index to restrict the search to those documents beginning with ^abc and the second part will match the exact entries. The bar operator '|' acts as an "or" function - the query `find({x:{$regex: /^abc|^def/})` matches the documents in which field 'x' has values that begin with "abc" or "def". To utilize the index, it's recommended to break the query into two different queries joined by the $or operator: `find( {$or : [{x: $regex: /^abc/}, {$regex: /^def/}] })`.

Update operators

Field update operators

- $inc
- $mul
- $rename
- $setOnInsert
- $set
- $unset
- $min
- $max
- $currentDate

Array update operators

- $addToSet
- $pop
- $pullAll
- $pull (Note: $pull with condition is not supported)
- $pushAll
- $push
- $each
- $slice
- $sort
- $position

Bitwise update operator

- $bit

Geospatial operators

OPERATOR	EXAMPLE	SUPPORTED
$geoWithin	`{ "Location.coordinates": { $geoWithin: { $centerSphere: [ [ -121, 46 ], 5 ] } } }`	Yes
$geoIntersects	`{ "Location.coordinates": { $geoIntersects: { $geometry: { type: "Polygon", coordinates: [ [ [ -121.9, 46.7 ], [ -121.5, 46.7 ], [ -121.5, 46.9 ], [ -121.9, 46.9 ], [ -121.9, 46.7 ] ] ] } } } }`	Yes
$near	`{ "Location.coordinates": { $near: { $geometry: { type: "Polygon", coordinates: [ [ [ -121.9, 46.7 ], [ -121.5, 46.7 ], [ -121.5, 46.9 ], [ -121.9, 46.9 ], [ -121.9, 46.7 ] ] ] } } } }`	Yes
$nearSphere	`{ "Location.coordinates": { $nearSphere : [ -121, 46 ], $maxDistance: 0.50 } }`	Yes
$geometry	`{ "Location.coordinates": { $geoWithin: { $geometry: { type: "Polygon", coordinates: [ [ [ -121.9, 46.7 ], [ -121.5, 46.7 ], [ -121.5, 46.9 ], [ -121.9, 46.9 ], [ -121.9, 46.7 ] ] ] } } } }`	Yes
$minDistance	`{ "Location.coordinates": { $nearSphere : { $geometry: {type: "Point", coordinates: [ -121, 46 ]}, $minDistance: 1000, $maxDistance: 1000000 } } }`	Yes
$maxDistance	`{ "Location.coordinates": { $nearSphere : [ -121, 46 ], $maxDistance: 0.50 } }`	Yes
$center	`{ "Location.coordinates": { $geoWithin: { $center: [ [-121, 46], 1 ] } } }`	Yes
$centerSphere	`{ "Location.coordinates": { $geoWithin: { $centerSphere: [ [ -121, 46 ], 5 ] } } }`	Yes
$box	`{ "Location.coordinates": { $geoWithin: { $box: [ [ 0, 0 ], [ -122, 47 ] ] } } }`	Yes
$polygon	`{ "Location.coordinates": { $near: { $geometry: { type: "Polygon", coordinates: [ [ [ -121.9, 46.7 ], [ -121.5, 46.7 ], [ -121.5, 46.9 ], [ -121.9, 46.9 ], [ -121.9, 46.7 ] ] ] } } } }`	Yes

Sort Operations

When using the `findOneAndUpdate` operation, sort operations on a single field are supported but sort operations on multiple fields are not supported.

Additional operators

OPERATOR	EXAMPLE	NOTES
$all	`{ "Location.coordinates": { $all: [-121.758, 46.87] } }`	
$elemMatch	`{ "Location.coordinates": { $elemMatch: { $lt: 0 } } }`	
$size	`{ "Location.coordinates": { $size: 2 } }`	
$comment	`{ "Location.coordinates": { $elemMatch: { $lt: 0 } }, $comment: "Negative values"}`	
$text		Not supported. Use $regex instead.

Unsupported operators

The `$where` and the `$eval` operators are not supported by Azure Cosmos DB.

Methods

Following methods are supported:

Cursor methods

METHOD	EXAMPLE	NOTES
cursor.sort()	`cursor.sort({ "Elevation": -1 })`	Documents without sort key do not get returned

Unique indexes

Cosmos DB indexes every field in documents that are written to the database by default. Unique indexes ensure that a specific field doesn't have duplicate values across all documents in a collection, similar to the way uniqueness is preserved on the default `_id` key. You can create custom indexes in Cosmos DB by using the createIndex command, including the 'unique' constraint.

Unique indexes are available for all Cosmos accounts using Azure Cosmos DB's API for MongoDB.

Time-to-live (TTL)

Cosmos DB supports a time-to-live (TTL) based on the timestamp of the document. TTL can be enabled for collections by going to the Azure portal.

User and role management

Cosmos DB does not yet support users and roles. However, Cosmos DB supports Azure role-based access control (Azure RBAC) and read-write and read-only passwords/keys that can be obtained through the Azure portal

(Connection String page).

Replication

Cosmos DB supports automatic, native replication at the lowest layers. This logic is extended out to achieve low-latency, global replication as well. Cosmos DB does not support manual replication commands.

Write Concern

Some applications rely on a Write Concern which specifies the number of responses required during a write operation. Due to how Cosmos DB handles replication in the background all writes are all automatically Quorum by default. Any write concern specified by the client code is ignored. Learn more in Using consistency levels to maximize availability and performance.

Sharding

Azure Cosmos DB supports automatic, server-side sharding. It manages shard creation, placement, and balancing automatically. Azure Cosmos DB does not support manual sharding commands, which means you don't have to invoke commands such as shardCollection, addShard, balancerStart, moveChunk etc. You only need to specify the shard key while creating the containers or querying the data.

Next steps

- Learn how to use Studio 3T with Azure Cosmos DB's API for MongoDB.
- Learn how to use Robo 3T with Azure Cosmos DB's API for MongoDB.
- Explore MongoDB samples with Azure Cosmos DB's API for MongoDB.

Note: This article describes a feature of Azure Cosmos DB that provides wire protocol compatibility with MongoDB databases. Microsoft does not run MongoDB databases to provide this service. Azure Cosmos DB is not affiliated with MongoDB, Inc.

Frequently asked questions about the Azure Cosmos DB's API for MongoDB

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: Azure Cosmos DB API for MongoDB

The Azure Cosmos DB's API for MongoDB is a wire-protocol compatibility layer that allows applications to easily and transparently communicate with the native Azure Cosmos database engine by using existing, community-supported SDKs and drivers for MongoDB. Developers can now use existing MongoDB toolchains and skills to build applications that take advantage of Azure Cosmos DB. Developers benefit from the unique capabilities of Azure Cosmos DB, which include global distribution with multi-region write replication, auto-indexing, backup maintenance, financially backed service level agreements (SLAs) etc.

How do I connect to my database?

The quickest way to connect to a Cosmos database with Azure Cosmos DB's API for MongoDB is to head over to the Azure portal. Go to your account and then, on the left navigation menu, click **Quick Start**. Quickstart is the best way to get code snippets to connect to your database.

Azure Cosmos DB enforces strict security requirements and standards. Azure Cosmos DB accounts require authentication and secure communication via TLS, so be sure to use TLSv1.2.

For more information, see Connect to your Cosmos database with Azure Cosmos DB's API for MongoDB.

Error codes while using Azure Cosmos DB's API for MongoDB?

Along with the common MongoDB error codes, the Azure Cosmos DB's API for MongoDB has its own specific error codes:

ERROR	CODE	DESCRIPTION	SOLUTION
TooManyRequests	16500	The total number of request units consumed is more than the provisioned request-unit rate for the container and has been throttled.	Consider scaling the throughput assigned to a container or a set of containers from the Azure portal or retrying again.

ExceededMemoryLimit	16501	As a multi-tenant service, the operation has gone over the client's memory allotment.	Reduce the scope of the operation through more restrictive query criteria or contact support from the Azure portal.
			
			Example:
			`db.getCollection('users').aggregate(`
			`{name: "Andy"}}, {$sort: {age: -1}}]`

Supported drivers

Is the Simba driver for MongoDB supported for use with Azure Cosmos DB's API for MongoDB?

Yes, you can use Simba's Mongo ODBC driver with Azure Cosmos DB's API for MongoDB

Next steps

- Build a .NET web app using Azure Cosmos DB's API for MongoDB
- Create a console app with Java and the MongoDB API in Azure Cosmos DB

Consistency levels for Azure Cosmos DB and the API for MongoDB

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: Azure Cosmos DB API for MongoDB

Unlike Azure Cosmos DB, the native MongoDB does not provide precisely defined consistency guarantees. Instead, native MongoDB allows users to configure the following consistency guarantees: a write concern, a read concern, and the isMaster directive - to direct the read operations to either primary or secondary replicas to achieve the desired consistency level.

When using Azure Cosmos DB's API for MongoDB, the MongoDB driver treats your write region as the primary replica and all other regions are read replica. You can choose which region associated with your Azure Cosmos account as a primary replica.

> **NOTE**
>
> The default consistency model for Azure Cosmos DB is Session. Session is a client-centric consistency model which is not natively supported by either Cassandra or MongoDB. For more information on which consistency model to choose see, Consistency levels in Azure Cosmos DB

While using Azure Cosmos DB's API for MongoDB:

- The write concern is mapped to the default consistency level configured on your Azure Cosmos account.
- Azure Cosmos DB will dynamically map the read concern specified by the MongoDB client driver to one of the Azure Cosmos DB consistency levels that is configured dynamically on a read request.
- You can annotate a specific region associated with your Azure Cosmos account as "Primary" by making the region as the first writable region.

Mapping consistency levels

The following table illustrates how the native MongoDB write/read concerns are mapped to the Azure Cosmos consistency levels when using Azure Cosmos DB's API for MongoDB:

MongoDB Write Concern	Default Consistency (set on the Cosmos account)	Region serving the reads	MongoDB Read Concern	Cosmos Consistency Level A(dynamically set on a read request)	Guarantees offered by native MongoDB	Guarantees offered by Cosmos MongoDB API
MAJORITY (W)	Strong	*IsMaster* = true	LOCAL, AVAILABLE	Bounded Staleness	Dirty Reads (Read Uncommitted) in steady state; Stale reads under split brain (network partitioning)	Linearizable reads in the region where the write is performed Bounded staleness in all other regions
			MAJORITY	Bounded Staleness	Linearizability in steady state; Stale reads under split brain (network partitioning)	Linearizable reads in the region where the write is performed Bounded staleness (*time-bounded* linearizability) in all other regions
			LINEARIZABILITY	Strong	Linearizability	Linearizability
			SNAPSHOT	Strong	Atomic reads across documents	Linearizability
		IsMaster = false	LOCAL, AVAILABLE	Consistent Prefix	Stale Reads; Dirty Reads	Consistent Prefix
			MAJORITY	Strong	Stale Reads	Linearizability
			LINEARIZABILITY	Not Allowed	Not Allowed	Not Allowed
			SNAPSHOT	Not Allowed	Not Allowed	Not Allowed
MINORITY (1 < W < Majority)	Bounded Staleness	*IsMaster* = true	LOCAL, AVAILABLE	Bounded Staleness	Dirty Reads in steady state; Stale reads under network partitioning	Linearizable reads in the region where the write is performed Bounded staleness (*time-bounded* linearizability) in all other regions
			MAJORITY	Bounded Staleness	Linearizability in steady state; Stale reads under network partitioning	Linearizable reads in the region where the write is performed Bounded staleness (*time-bounded* linearizability) in all other regions
			LINEARIZABILITY	Bounded Staleness	Linearizability	Linearizable reads in the region where the write is performed Bounded staleness (*time-bounded* linearizability) in all other regions
			SNAPSHOT	Bounded Staleness	Linearizability	Linearizable reads in the region where the write is performed Bounded staleness (*time-bounded* linearizability) in all other regions
		IsMaster = false	LOCAL, AVAILABLE	Consistent Prefix	Stale Reads; Dirty Reads	Consistent Prefix
			MAJORITY	Bounded Staleness	Stale Reads; Monotonic Reads	Bounded staleness configured via *K* and *T* along with monotonicity
			LINEARIZABILITY / SNAPSHOT	Not Allowed	Not Allowed	Not Allowed
None (W=0)	Consistent Prefix	*IsMaster* = true/false	ANY	Consistent Prefix	Stale Reads; Dirty Reads	Consistent Prefix

If your Azure Cosmos account is configured with a consistency level other than the strong consistency, you can find out the probability that your clients may get strong and consistent reads for your workloads by looking at the *Probabilistically Bounded Staleness* (PBS) metric. This metric is exposed in the Azure portal, to learn more, see Monitor Probabilistically Bounded Staleness (PBS) metric.

Probabilistic bounded staleness shows how eventual is your eventual consistency. This metric provides an insight into how often you can get a stronger consistency than the consistency level that you have currently configured on your Azure Cosmos account. In other words, you can see the probability (measured in milliseconds) of getting strongly consistent reads for a combination of write and read regions.

Next steps

Learn more about global distribution and consistency levels for Azure Cosmos DB:

- Global distribution overview
- Consistency Level overview
- High availability

Pre-migration steps for data migrations from MongoDB to Azure Cosmos DB's API for MongoDB

12/4/2020 • 5 minutes to read • Edit Online

APPLIES TO: Azure Cosmos DB API for MongoDB

Before you migrate your data from MongoDB (either on-premises or in the cloud) to Azure Cosmos DB's API for MongoDB, you should:

1. Read the key considerations about using Azure Cosmos DB's API for MongoDB
2. Choose an option to migrate your data
3. Estimate the throughput needed for your workloads
4. Pick an optimal partition key for your data
5. Understand the indexing policy that you can set on your data

If you have already completed the above pre-requisites for migration, you can Migrate MongoDB data to Azure Cosmos DB's API for MongoDB using the Azure Database Migration Service. Additionally, if you haven't created an account, you can browse any of the Quickstarts that show the steps to create an account.

Considerations when using Azure Cosmos DB's API for MongoDB

The following are specific characteristics about Azure Cosmos DB's API for MongoDB:

- **Capacity model**: Database capacity on Azure Cosmos DB is based on a throughput-based model. This model is based on Request Units per second, which is a unit that represents the number of database operations that can be executed against a collection on a per-second basis. This capacity can be allocated at a database or collection level, and it can be provisioned on an allocation model, or using the autoscale provisioned throughput.
- **Request Units**: Every database operation has an associated Request Units (RUs) cost in Azure Cosmos DB. When executed, this is subtracted from the available request units level on a given second. If a request requires more RUs than the currently allocated RU/s there are two options to solve the issue - increase the amount of RUs, or wait until the next second starts and then retry the operation.
- **Elastic capacity**: The capacity for a given collection or database can change at any time. This allows for the database to elastically adapt to the throughput requirements of your workload.
- **Automatic sharding**: Azure Cosmos DB provides an automatic partitioning system that only requires a shard (or a partition key). The automatic partitioning mechanism is shared across all the Azure Cosmos DB APIs and it allows for seamless data and throughout scaling through horizontal distribution.

Migration options for Azure Cosmos DB's API for MongoDB

The Azure Database Migration Service for Azure Cosmos DB's API for MongoDB provides a mechanism that simplifies data migration by providing a fully managed hosting platform, migration monitoring options and automatic throttling handling. The full list of options are the following:

MIGRATION TYPE	SOLUTION	CONSIDERATIONS

MIGRATION TYPE	SOLUTION	CONSIDERATIONS
Online	Azure Database Migration Service	• Makes use of the Azure Cosmos DB bulk executor library • Suitable for large datasets and takes care of replicating live changes • Works only with other MongoDB sources
Offline	Azure Database Migration Service	• Makes use of the Azure Cosmos DB bulk executor library • Suitable for large datasets and takes care of replicating live changes • Works only with other MongoDB sources
Offline	Azure Data Factory	• Easy to set up and supports multiple sources • Makes use of the Azure Cosmos DB bulk executor library • Suitable for large datasets • Lack of checkpointing means that any issue during the course of migration would require a restart of the whole migration process • Lack of a dead letter queue would mean that a few erroneous files could stop the entire migration process • Needs custom code to increase read throughput for certain data sources
Offline	Existing Mongo Tools (mongodump, mongorestore, Studio3T)	• Easy to set up and integration • Needs custom handling for throttles

Estimate the throughput need for your workloads

In Azure Cosmos DB, the throughput is provisioned in advance and is measured in Request Units (RU's) per second. Unlike VMs or on-premises servers, RUs are easy to scale up and down at any time. You can change the number of provisioned RUs instantly. For more information, see Request units in Azure Cosmos DB.

You can use the Azure Cosmos DB Capacity Calculator to determine the amount of Request Units based on your database account configuration, amount of data, document size, and required reads and writes per second.

The following are key factors that affect the number of required RUs:

- **Document size**: As the size of an item/document increases, the number of RUs consumed to read or write the item/document also increases.
- **Document property count**:The number of RUs consumed to create or update a document is related to the number, complexity and length of its properties. You can reduce the request unit consumption for write operations by limiting the number of indexed properties.
- **Query patterns**: The complexity of a query affects how many request units are consumed by the query.

The best way to understand the cost of queries is to use sample data in Azure Cosmos DB, and run sample queries from the MongoDB Shell using the `getLastRequestStastistics` command to get the request charge, which will output the number of RUs consumed:

```
db.runCommand({getLastRequestStatistics: 1})
```

This command will output a JSON document similar to the following:

```
{ "_t": "GetRequestStatisticsResponse", "ok": 1, "CommandName": "find", "RequestCharge": 10.1,
"RequestDurationInMilliSeconds": 7.2}
```

You can also use the diagnostic settings to understand the frequency and patterns of the queries executed against Azure Cosmos DB. The results from the diagnostic logs can be sent to a storage account, an EventHub instance or Azure Log Analytics.

Choose your partition key

Partitioning, also known as Sharding, is a key point of consideration before migrating data. Azure Cosmos DB uses fully-managed partitioning to increase the capacity in a database to meet the storage and throughput requirements. This feature doesn't need the hosting or configuration of routing servers.

In a similar way, the partitioning capability automatically adds capacity and re-balances the data accordingly. For details and recommendations on choosing the right partition key for your data, please see the Choosing a Partition Key article.

Index your data

The Azure Cosmos DB's API for MongoDB server version 3.6 automatically indexes the `_id` field only. This field can't be dropped. It automatically enforces the uniqueness of the `_id` field per shard key. To index additional fields, you apply the MongoDB index-management commands. This default indexing policy differs from the Azure Cosmos DB SQL API, which indexes all fields by default.

The indexing capabilities provided by Azure Cosmos DB include adding compound indices, unique indices and time-to-live (TTL) indices. The index management interface is mapped to the `createIndex()` command. Learn more at Indexing in Azure Cosmos DB's API for MongoDBarticle.

Azure Database Migration Service automatically migrates MongoDB collections with unique indexes. However, the unique indexes must be created before the migration. Azure Cosmos DB does not support the creation of unique indexes, when there is already data in your collections. For more information, see Unique keys in Azure Cosmos DB.

Next steps

- Migrate your MongoDB data to Cosmos DB using the Database Migration Service.
- Provision throughput on Azure Cosmos containers and databases
- Partitioning in Azure Cosmos DB
- Global Distribution in Azure Cosmos DB
- Indexing in Azure Cosmos DB
- Request Units in Azure Cosmos DB

Tutorial: Migrate MongoDB to Azure Cosmos DB's API for MongoDB online using DMS

12/4/2020 • 9 minutes to read • Edit Online

You can use Azure Database Migration Service to perform an online (minimal downtime) migration of databases from an on-premises or cloud instance of MongoDB to Azure Cosmos DB's API for MongoDB.

This tutorial demonstrates the steps associated with using Azure Database Migration Service to migrate MongoDB data to Azure Cosmos DB:

- Create an instance of Azure Database Migration Service.
- Create a migration project.
- Specify the source.
- Specify the target.
- Map to target databases.
- Run the migration.
- Monitor the migration.
- Verify data in Azure Cosmos DB.
- Complete the migration when you are ready.

In this tutorial, you migrate a dataset in MongoDB hosted in an Azure Virtual Machine to Azure Cosmos DB's API for MongoDB with minimal downtime by using Azure Database Migration Service. If you don't have a MongoDB source set up already, see the article Install and configure MongoDB on a Windows VM in Azure.

> **NOTE**
> Using Azure Database Migration Service to perform an online migration requires creating an instance based on the Premium pricing tier.

> **IMPORTANT**
> For an optimal migration experience, Microsoft recommends creating an instance of Azure Database Migration Service in the same Azure region as the target database. Moving data across regions or geographies can slow down the migration process.

> **TIP**
> When you migrate databases to Azure by using Azure Database Migration Service, you can do an *offline* or an *online* migration. With an offline migration, application downtime starts when the migration starts. With an online migration, downtime is limited to the time to cut over at the end of migration. We suggest that you test an offline migration to determine whether the downtime is acceptable; if not, do an online migration.

This article describes an online migration from MongoDB to Azure Cosmos DB's API for MongoDB. For an offline migration, see Migrate MongoDB to Azure Cosmos DB's API for MongoDB offline using DMS.

Prerequisites

To complete this tutorial, you need to:

- Complete the pre-migration steps such as estimating throughput, choosing a partition key, and the indexing policy.
- Create an Azure Cosmos DB's API for MongoDB account.
- Create a Microsoft Azure Virtual Network for Azure Database Migration Service by using Azure Resource Manager deployment model, which provides site-to-site connectivity to your on-premises source servers by using either ExpressRoute or VPN.

> **NOTE**
>
> During virtual network setup, if you use ExpressRoute with network peering to Microsoft, add the following service endpoints to the subnet in which the service will be provisioned:
>
> - Target database endpoint (for example, SQL endpoint, Cosmos DB endpoint, and so on)
> - Storage endpoint
> - Service bus endpoint
>
> This configuration is necessary because Azure Database Migration Service lacks internet connectivity.

- Ensure that your virtual network Network Security Group (NSG) rules don't block the following communication ports: 53, 443, 445, 9354, and 10000-20000. For more detail on virtual network NSG traffic filtering, see the article Filter network traffic with network security groups.
- Open your Windows firewall to allow Azure Database Migration Service to access the source MongoDB server, which by default is TCP port 27017.
- When using a firewall appliance in front of your source database(s), you may need to add firewall rules to allow Azure Database Migration Service to access the source database(s) for migration.

Register the Microsoft.DataMigration resource provider

1. Sign in to the Azure portal, select **All services**, and then select **Subscriptions**.

2. Select the subscription in which you want to create the instance of Azure Database Migration Service, and then select **Resource providers**.

3. Search for migration, and then to the right of **Microsoft.DataMigration**, select **Register**.

Create an instance

1. In the Azure portal, select + **Create a resource**, search for Azure Database Migration Service, and then select **Azure Database Migration Service** from the drop-down list.

2. On the **Azure Database Migration Service** screen, select **Create**.

3. On the **Create Migration Service** screen, specify a name for the service, the subscription, and a new or existing resource group.

4. Select the location in which you want to create the instance of Azure Database Migration Service.

5. Select an existing virtual network, or create a new one.

   The virtual network provides Azure Database Migration Service with access to the source MongoDB instance and the target Azure Cosmos DB account.

   For more information about how to create a virtual network in the Azure portal, see the article Create a virtual network using the Azure portal.

6. Select a SKU from the Premium pricing tier.

   > **NOTE**
   > Online migrations are supported only when using the Premium tier. For more information on costs and pricing tiers, see the pricing page.

7. Select **Create** to create the service.

Create a migration project

After the service is created, locate it within the Azure portal, open it, and then create a new migration project.

1. In the Azure portal, select **All services**, search for Azure Database Migration Service, and then select **Azure Database Migration Services**.

2. On the **Azure Database Migration Services** screen, search for the name of Azure Database Migration Service instance that you created, and then select the instance.

   Alternately, you can discover Azure Database Migration service instance from the search pane in Azure portal.

3. Select + **New Migration Project**.

4. On the **New migration project** screen, specify a name for the project, in the **Source server type** text box, select **MongoDB**, in the **Target server type** text box, select **CosmosDB (MongoDB API)**, and then for **Choose type of activity**, select **Online data migration [preview]**.

5. Select **Save**, and then select **Create and run activity** to create the project and run the migration activity.

Specify source details

1. On the **Source details** screen, specify the connection details for the source MongoDB server.

> **IMPORTANT**
>
> Azure Database Migration Service does not support Azure Cosmos DB as a source.

There are three modes to connect to a source:

- **Standard mode**, which accepts a fully qualified domain name or an IP address, Port number, and connection credentials.
- **Connection string mode**, which accepts a MongoDB Connection string as described in the article Connection String URI Format.
- **Data from Azure storage**, which accepts a blob container SAS URL. Select **Blob contains BSON dumps** if the blob container has BSON dumps produced by the MongoDB bsondump tool, and de-select it if the container contains JSON files.

  If you select this option, be sure that the storage account connection string appears in the format:

  ```
  https://blobnameurl/container?SASKEY
  ```

  Also, based on the type dump information in Azure Storage, keep the following detail in mind.

  - For BSON dumps, the data within the blob container must be in bsondump format, such that data files are placed into folders named after the containing databases in the format collection.bson. Metadata files (if any) should be named using the format *collection*.metadata.json.
  - For JSON dumps, the files in the blob container must be placed into folders named after the containing databases. Within each database folder, data files must be placed in a subfolder

called "data" and named using the format *collection*.json. Metadata files (if any) must be placed in a subfolder called "metadata" and named using the same format, *collection*.json. The metadata files must be in the same format as produced by the MongoDB bsondump tool.

> **IMPORTANT**
>
> It is discouraged to use a self-signed certificate on the mongo server. However, if one is used, please connect to the server using **connection string mode** and ensure that your connection string has ""
>
> ```
> &sslVerifyCertificate=false
> ```

You can use the IP Address for situations in which DNS name resolution isn't possible.

2. Select **Save**.

> **NOTE**
>
> The Source server address should be the address of the primary if the source is a replica set, and the router if the source is a sharded MongoDB cluster. For a sharded MongoDB cluster, Azure Database Migration Service must be able to connect to the individual shards in the cluster, which may require opening the firewall on more machines.

Specify target details

1. On the **Migration target details** screen, specify the connection details for the target Azure Cosmos DB account, which is the pre-provisioned Azure Cosmos DB's API for MongoDB account to which you're

migrating your MongoDB data.

2. Select **Save**.

Map to target databases

1. On the **Map to target databases** screen, map the source and the target database for migration.

   If the target database contains the same database name as the source database, Azure Database Migration Service selects the target database by default.

   If the string **Create** appears next to the database name, it indicates that Azure Database Migration Service didn't find the target database, and the service will create the database for you.

   At this point in the migration, if you want share throughput on the database, specify a throughput RU. In Cosmos DB, you can provision throughput either at the database-level or individually for each collection. Throughput is measured in Request Units (RUs). Learn more about Azure Cosmos DB pricing.

2. Select **Save**.

3. On the **Collection setting** screen, expand the collections listing, and then review the list of collections that will be migrated.

   Azure Database Migration Service auto selects all the collections that exist on the source MongoDB instance that don't exist on the target Azure Cosmos DB account. If you want to remigrate collections that already include data, you need to explicitly select the collections on this screen.

   You can specify the number of RUs that you want the collections to use. In most cases, a value between 500 (1000 minimum for sharded collections) and 4000 should suffice. Azure Database Migration Service suggests smart defaults based on the collection size.

   > **NOTE**
   > Perform the database migration and collection in parallel using multiple instances of Azure Database Migration Service, if necessary, to speed up the run.

   You can also specify a shard key to take advantage of partitioning in Azure Cosmos DB for optimal scalability. Be sure to review the best practices for selecting a shard/partition key. If you don't have a partition key, you can always use **_id** as the shard key for better throughput.

4. Select **Save**.

5. On the **Migration summary** screen, in the **Activity name** text box, specify a name for the migration activity.

Run the migration

- Select **Run migration**.

  The migration activity window appears, and the **Status** of the activity is displayed.

Monitor the migration

- On the migration activity screen, select **Refresh** to update the display until the **Status** of the migration shows as **Replaying**.

> **NOTE**
>
> You can select the Activity to get details of database- and collection-level migration metrics.

Verify data in Cosmos DB

1. Make changes to your source MongoDB database.

2. Connect to COSMOS DB to verify if the data is replicated from the source MongoDB server.

Complete the migration

- After all documents from the source are available on the COSMOS DB target, select **Finish** from the migration activity's context menu to complete the migration.

  This action will finish replaying all the pending changes and complete the migration.

Post-migration optimization

After you migrate the data stored in MongoDB database to Azure Cosmos DB's API for MongoDB, you can connect to Azure Cosmos DB and manage the data. You can also perform other post-migration optimization steps such as optimizing the indexing policy, update the default consistency level, or configure global distribution for your Azure

Cosmos DB account. For more information, see the Post-migration optimization article.

Additional resources

- Cosmos DB service information

Next steps

- Review migration guidance for additional scenarios in the Microsoft Database Migration Guide.

Tutorial: Migrate MongoDB to Azure Cosmos DB's API for MongoDB offline using DMS

12/4/2020 • 7 minutes to read • Edit Online

You can use Azure Database Migration Service to perform an offline (one-time) migration of databases from an on-premises or cloud instance of MongoDB to Azure Cosmos DB's API for MongoDB.

In this tutorial, you learn how to:

- Create an instance of Azure Database Migration Service.
- Create a migration project by using Azure Database Migration Service.
- Run the migration.
- Monitor the migration.

In this tutorial, you migrate a dataset in MongoDB hosted in an Azure Virtual Machine to Azure Cosmos DB's API for MongoDB by using Azure Database Migration Service. If you don't have a MongoDB source set up already, see the article Install and configure MongoDB on a Windows VM in Azure.

Prerequisites

To complete this tutorial, you need to:

- Complete the pre-migration steps such as estimating throughput, choosing a partition key, and the indexing policy.
- Create an Azure Cosmos DB's API for MongoDB account.
- Create a Microsoft Azure Virtual Network for Azure Database Migration Service by using Azure Resource Manager deployment model, which provides site-to-site connectivity to your on-premises source servers by using either ExpressRoute or VPN. For more information about creating a virtual network, see the Virtual Network Documentation, and especially the quickstart articles with step-by-step details.

  > **NOTE**
  >
  > During virtual network setup, if you use ExpressRoute with network peering to Microsoft, add the following service endpoints to the subnet in which the service will be provisioned:
  >
  > - Target database endpoint (for example, SQL endpoint, Cosmos DB endpoint, and so on)
  > - Storage endpoint
  > - Service bus endpoint
  >
  > This configuration is necessary because Azure Database Migration Service lacks internet connectivity.

- Ensure that your virtual network Network Security Group (NSG) rules don't block the following communication ports: 53, 443, 445, 9354, and 10000-20000. For more detail on virtual network NSG traffic filtering, see the article Filter network traffic with network security groups.
- Open your Windows firewall to allow Azure Database Migration Service to access the source MongoDB server, which by default is TCP port 27017.
- When using a firewall appliance in front of your source database(s), you may need to add firewall rules

to allow Azure Database Migration Service to access the source database(s) for migration.

Register the Microsoft.DataMigration resource provider

1. Sign in to the Azure portal, select **All services**, and then select **Subscriptions**.

2. Select the subscription in which you want to create the instance of the Azure Database Migration Service, and then select **Resource providers**.

3. Search for migration, and then to the right of **Microsoft.DataMigration**, select **Register**.

Create an instance

1. In the Azure portal, select + **Create a resource**, search for Azure Database Migration Service, and then select **Azure Database Migration Service** from the drop-down list.

2. On the **Azure Database Migration Service** screen, select **Create**.

3. On the **Create Migration Service** screen, specify a name for the service, the subscription, and a new or existing resource group.

4. Select the location in which you want to create the instance of Azure Database Migration Service.

5. Select an existing virtual network or create a new one.

   The virtual network provides Azure Database Migration Service with access to the source MongoDB instance and the target Azure Cosmos DB account.

   For more information about how to create a virtual network in the Azure portal, see the article Create a virtual network using the Azure portal.

6. Select a pricing tier.

   For more information on costs and pricing tiers, see the pricing page.

7. Select **Create** to create the service.

Create a migration project

After the service is created, locate it within the Azure portal, open it, and then create a new migration project.

1. In the Azure portal, select **All services**, search for Azure Database Migration Service, and then select **Azure Database Migration Services**.

2. On the **Azure Database Migration Services** screen, search for the name of Azure Database Migration Service instance that you created, and then select the instance.

3. Select + **New Migration Project**.

4. On the **New migration project** screen, specify a name for the project, in the **Source server type** text box, select **MongoDB**, in the **Target server type** text box, select **CosmosDB (MongoDB API)**, and then for **Choose type of activity**, select **Offline data migration**.

5. Select **Create and run activity** to create the project and run the migration activity.

Specify source details

1. On the **Source details** screen, specify the connection details for the source MongoDB server.

> **IMPORTANT**
>
> Azure Database Migration Service does not support Azure Cosmos DB as a source.

There are three modes to connect to a source:

- **Standard mode**, which accepts a fully qualified domain name or an IP address, Port number, and connection credentials.
- **Connection string mode**, which accepts a MongoDB Connection string as described in the article Connection String URI Format.

- **Data from Azure storage**, which accepts a blob container SAS URL. Select **Blob contains BSON dumps** if the blob container has BSON dumps produced by the MongoDB bsondump tool, and de-select it if the container contains JSON files.

  If you select this option, be sure that the storage account connection string appears in the format:

  ```
  https://blobnameurl/container?SASKEY
  ```

  This blob container SAS connection string can be found in Azure Storage explorer. Creating the SAS for the concerned container will provide you the URL in above requested format.

  Also, based on the type dump information in Azure Storage, keep the following detail in mind.

  - For BSON dumps, the data within the blob container must be in bsondump format, such that data files are placed into folders named after the containing databases in the format collection.bson. Metadata files (if any) should be named using the format *collection*.metadata.json.
  - For JSON dumps, the files in the blob container must be placed into folders named after the containing databases. Within each database folder, data files must be placed in a subfolder called "data" and named using the format *collection*.json. Metadata files (if any) must be placed in a subfolder called "metadata" and named using the same format, *collection*.json. The metadata files must be in the same format as produced by the MongoDB bsondump tool.

> **IMPORTANT**
>
> It is discouraged to use a self-signed certificate on the mongo server. However, if one is used, please connect to the server using **connection string mode** and ensure that your connection string has “”
>
> ```
> &sslVerifyCertificate=false
> ```

You can also use the IP Address for situations in which DNS name resolution isn't possible.

2. Select **Save**.

Specify target details

1. On the **Migration target details** screen, specify the connection details for the target Azure Cosmos DB account, which is the pre-provisioned Azure Cosmos DB's API for MongoDB account to which you're migrating your MongoDB data.

2. Select **Save**.

Map to target databases

1. On the **Map to target databases** screen, map the source and the target database for migration.

   If the target database contains the same database name as the source database, Azure Database Migration Service selects the target database by default.

   If the string **Create** appears next to the database name, it indicates that Azure Database Migration Service didn't find the target database, and the service will create the database for you.

   At this point in the migration, you can provision throughput. In Cosmos DB, you can provision throughput either at the database-level or individually for each collection. Throughput is measured in Request Units (RUs). Learn more about Azure Cosmos DB pricing.

2. Select **Save**.

3. On the **Collection setting** screen, expand the collections listing, and then review the list of collections that will be migrated.

   Azure Database Migration Service auto selects all the collections that exist on the source MongoDB instance that don't exist on the target Azure Cosmos DB account. If you want to remigrate collections that already include data, you need to explicitly select the collections on this blade.

   You can specify the amount of RUs that you want the collections to use. Azure Database Migration Service suggests smart defaults based on the collection size.

   > **NOTE**
   > Perform the database migration and collection in parallel using multiple instances of Azure Database Migration Service, if necessary, to speed up the run.

   You can also specify a shard key to take advantage of partitioning in Azure Cosmos DB for optimal scalability. Be sure to review the best practices for selecting a shard/partition key.

4. Select **Save**.

5. On the **Migration summary** screen, in the **Activity name** text box, specify a name for the migration activity.

Run the migration

- Select **Run migration**.

  The migration activity window appears, and the **Status** of the activity is **Not started**.

Monitor the migration

- On the migration activity screen, select **Refresh** to update the display until the **Status** of the migration shows as **Completed**.

> **NOTE**
>
> You can select the Activity to get details of database- and collection-level migration metrics.

Verify data in Cosmos DB

- After the migration completes, you can check your Azure Cosmos DB account to verify that all the collections were migrated successfully.

Post-migration optimization

After you migrate the data stored in MongoDB database to Azure Cosmos DB's API for MongoDB, you can connect to Azure Cosmos DB and manage the data. You can also perform other post-migration optimization steps such as optimizing the indexing policy, update the default consistency level, or configure global distribution for your Azure Cosmos DB account. For more information, see the Post-migration optimization article.

Additional resources

- Cosmos DB service information

Next steps

- Review migration guidance for additional scenarios in the Microsoft Database Migration Guide.

Post-migration optimization steps when using Azure Cosmos DB's API for MongoDB

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: Azure Cosmos DB API for MongoDB

After you migrate the data stored in MongoDB database to Azure Cosmos DB's API for MongoDB, you can connect to Azure Cosmos DB and manage the data. This guide provides the steps you should consider after the migration. See the Migrate MongoDB to Azure Cosmos DB's API for MongoDB tutorial for the migration steps.

In this guide, you will learn how to:

- Connect your application
- Optimize the indexing policy
- Configure global distribution for Azure Cosmos DB's API for MongoDB
- Set consistency level

> **NOTE**
>
> The only mandatory post-migration step on your application level is changing the connection string in your application to point to your new Azure Cosmos DB account. All other migration steps are recommended optimizations.

Connect your application

1. In a new window sign into the Azure portal
2. From the Azure portal, in the left pane open the **All resources** menu and find the Azure Cosmos DB account to which you have migrated your data.
3. Open the **Connection String** blade. The right pane contains all the information that you need to successfully connect to your account.
4. Use the connection information in your application's configuration (or other relevant places) to reflect the Azure Cosmos DB's API for MongoDB connection in your app.

For more details, please see the Connect a MongoDB application to Azure Cosmos DB page.

Optimize the indexing policy

All data fields are automatically indexed, by default, during the migration of data to Azure Cosmos DB. In many cases, this default indexing policy is acceptable. In general, removing indexes optimizes write requests and having the default indexing policy (i.e., automatic indexing) optimizes read requests.

For more information on indexing, see Data indexing in Azure Cosmos DB's API for MongoDB as well as the Indexing in Azure Cosmos DB articles.

Globally distribute your data

Azure Cosmos DB is available in all Azure regions worldwide. After selecting the default consistency level for your Azure Cosmos DB account, you can associate one or more Azure regions (depending on your global distribution needs). For high availability and business continuity, we always recommend running in at least 2 regions. You can review the tips for optimizing cost of multi-region deployments in Azure Cosmos DB.

To globally distribute your data, please see Distribute data globally on Azure Cosmos DB's API for MongoDB.

Set consistency level

Azure Cosmos DB offers 5 well-defined consistency levels. To read about the mapping between MongoDB and Azure Cosmos DB consistency levels, read Consistency levels and Azure Cosmos DB APIs. The default consistency level is the session consistency level. Changing the consistency level is optional and you can optimize it for your app. To change consistency level using Azure portal:

1. Go to the **Default Consistency** blade under Settings.
2. Select your consistency level

Most users leave their consistency level at the default session consistency setting. However, there are availability and performance tradeoffs for various consistency levels.

Next steps

- Connect a MongoDB application to Azure Cosmos DB
- Connect to Azure Cosmos DB account using Studio 3T
- How to globally distribute reads using Azure Cosmos DB's API for MongoDB
- Expire data with Azure Cosmos DB's API for MongoDB
- Consistency Levels in Azure Cosmos DB
- Indexing in Azure Cosmos DB
- Request Units in Azure Cosmos DB

Upgrade the MongoDB wire protocol version of your Azure Cosmos DB's API for MongoDB account

12/4/2020 • 3 minutes to read • Edit Online

APPLIES TO: Azure Cosmos DB API for MongoDB

This article describes how to upgrade the wire protocol version of your Azure Cosmos DB's API for MongoDB account. After you upgrade the wire protocol version, you can use the latest functionality in Azure Cosmos DB's API for MongoDB. The upgrade process doesn't interrupt the availability of your account and it doesn't consume RU/s or decrease the capacity of the database at any point. No existing data or indexes will be affected by this process.

> **NOTE**
> At this moment, only qualifying accounts using the server version 3.2 can be upgraded to version 3.6. If your account doesn't show the upgrade option, please file a support ticket.

Upgrading from version 3.2 to 3.6

Benefits of upgrading to version 3.6

The following are the new features included in version 3.6:

- Enhanced performance and stability
- Support for new database commands
- Support for aggregation pipeline by default and new aggregation stages
- Support for Change Streams
- Support for compound Indexes
- Cross-partition support for the following operations: update, delete, count and sort
- Improved performance for the following aggregate operations: $count, $skip, $limit and $group
- Wildcard indexing is now supported

Changes from version 3.2

- **RequestRateIsLarge errors have been removed**. Requests from the client application will not return 16500 errors anymore. Instead requests will resume until they complete or fulfill the timeout.
- Per request timeout is set to 60 seconds.
- MongoDB collections created on the new wire protocol version will only have the `_id` property indexed by default.

Action required

For the upgrade to version 3.6, the database account endpoint suffix will be updated to the following format:

```
<your_database_account_name>.mongo.cosmos.azure.com
```

You need to replace the existing endpoint in your applications and drivers that connect with this database account. **Only connections that are using the new endpoint will have access to the features in the MongoDB version 3.6**. The previous endpoint should have the suffix `.documents.azure.com`.

> **NOTE**
>
> This endpoint might have slight differences if your account was created in a Sovereign, Government or Restricted Azure Cloud.

How to upgrade

1. First, go to the Azure portal and navigate to your Azure Cosmos DB API for MongoDB account overview blade. Verify that your server version is `3.2`.

2. From the options on the left, select the `Features` blade. This will reveal the Account level features that are available for your database account.

3. Click on the `Upgrade to Mongo server version 3.6` row. If you don't see this option, your account might not be eligible for this upgrade. Please file a support ticket if that is the case.

4. Review the information displayed about this specific upgrade. Note that the upgrade will only be completed until your applications use the updated endpoint, as highlighted in this section. Click on `Enable` as soon as you are ready to start the process.

5. After starting the process, the `Features` menu will show the status of the upgrade. The status will go from `Pending`, to `In Progress`, to `Upgraded`. This process will not affect the existing functionality or operations of the database account.

6. Once the upgrade is completed, the status will show as `Upgraded`. Click on it to learn more about the next steps and actions you need to take to finalize the process. Please contact support if there was an issue processing your request.

7. **To start using the upgraded version of your database account**, go back to the `Overview` blade, and copy the new connection string to use in your application. The applications will start using the upgraded version as soon as they connect to the new endpoint. Existing connections will not be interrupted and can be updated at your convenience. To ensure a consistent experience, all your applications must use the new endpoint.

Next steps

- Learn about the supported and unsupported features of MongoDB version 3.6.
- For further information check Mongo 3.6 version features

Connect a MongoDB application to Azure Cosmos DB

11/2/2020 • 2 minutes to read • Edit Online

APPLIES TO: Azure Cosmos DB API for MongoDB

Learn how to connect your MongoDB app to an Azure Cosmos DB by using a MongoDB connection string. You can then use an Azure Cosmos database as the data store for your MongoDB app.

This tutorial provides two ways to retrieve connection string information:

- The quickstart method, for use with .NET, Node.js, MongoDB Shell, Java, and Python drivers
- The custom connection string method, for use with other drivers

Prerequisites

- An Azure account. If you don't have an Azure account, create a free Azure account now.
- A Cosmos account. For instructions, see Build a web app using Azure Cosmos DB's API for MongoDB and .NET SDK.

Get the MongoDB connection string by using the quick start

1. In an Internet browser, sign in to the Azure portal.
2. In the **Azure Cosmos DB** blade, select the API.
3. In the left pane of the account blade, click **Quick start**.
4. Choose your platform (**.NET**, **Node.js**, **MongoDB Shell**, **Java**, **Python**). If you don't see your driver or tool listed, don't worry--we continuously document more connection code snippets. Please comment below on what you'd like to see. To learn how to craft your own connection, read Get the account's connection string information.
5. Copy and paste the code snippet into your MongoDB app.

Get the MongoDB connection string to customize

1. In an Internet browser, sign in to the Azure portal.
2. In the **Azure Cosmos DB** blade, select the API.
3. In the left pane of the account blade, click **Connection String**.
4. The **Connection String** blade opens. It has all the information necessary to connect to the account by using a driver for MongoDB, including a

preconstructed connection string.

Connection string requirements

> **IMPORTANT**
>
> Azure Cosmos DB has strict security requirements and standards. Azure Cosmos DB accounts require authentication and secure communication via *TLS*.

Azure Cosmos DB supports the standard MongoDB connection string URI format, with a couple of specific requirements: Azure Cosmos DB accounts require authentication and secure communication via TLS. So, the connection string format is:

```
mongodb://username:password@host:port/[database]?ssl=true
```

The values of this string are available in the **Connection String** blade shown earlier:

- Username (required): Cosmos account name.
- Password (required): Cosmos account password.
- Host (required): FQDN of the Cosmos account.
- Port (required): 10255.
- Database (optional): The database that the connection uses. If no database is provided, the default database is "test."
- ssl=true (required)

For example, consider the account shown in the **Connection String** blade. A valid connection string is:

```
mongodb://contoso123:0Fc3IolnL12312asdfawejunASDF@asdfYXX2t8a97kghVcUzcDv98hawelufhawefafnoQRGwNj2nMPL1Y9qsIr9Srdw==@contoso123.documents.azure.com:10255/mydatabase
ssl=true
```

Next steps

- Learn how to use Studio 3T with Azure Cosmos DB's API for MongoDB.
- Learn how to use Robo 3T with Azure Cosmos DB's API for MongoDB.
- Explore MongoDB samples with Azure Cosmos DB's API for MongoDB.

Connect to an Azure Cosmos account using Studio 3T

11/2/2020 • 2 minutes to read • Edit Online

APPLIES TO: Azure Cosmos DB API for MongoDB

To connect to an Azure Cosmos DB's API for MongoDB using Studio 3T, you must:

- Download and install Studio 3T.
- Have your Azure Cosmos account's connection string information.

Create the connection in Studio 3T

To add your Azure Cosmos account to the Studio 3T connection manager, use the following steps:

1. Retrieve the connection information for your Azure Cosmos DB's API for MongoDB account using the instructions in the Connect a MongoDB application to Azure Cosmos DB article.

2. Click **Connect** to open the Connection Manager, then click **New Connection**

3. In the **New Connection** window, on the **Server** tab, enter the HOST (FQDN) of the Azure Cosmos account and the PORT.

4. In the **New Connection** window, on the **Authentication** tab, choose Authentication Mode **Basic (MONGODB-CR or SCARM-SHA-1)** and enter the USERNAME and PASSWORD. Accept the default authentication db (admin) or provide your own value.

5. In the **New Connection** window, on the **SSL** tab, check the **Use SSL protocol to connect** check box and the **Accept server self-signed SSL certificates** radio button.

6. Click the **Test Connection** button to validate the connection information, click **OK** to return to the New

Connection window, and then click **Save**.

Use Studio 3T to create a database, collection, and documents

To create a database, collection, and documents using Studio 3T, perform the following steps:

1. In **Connection Manager**, highlight the connection and click **Connect**.

2. Right-click the host and choose **Add Database**. Provide a database name and click **OK**.

3. Right-click the database and choose **Add Collection**. Provide a collection name and click **Create**.

4. Click the **Collection** menu item, then click **Add Document**.

5. In the Add Document dialog, paste the following and then click **Add Document**.

```
{
    "_id": "AndersenFamily",
    "lastName": "Andersen",
    "parents": [
        { "firstName": "Thomas" },
        { "firstName": "Mary Kay"}
    ],
    "children": [
        {
            "firstName": "Henriette Thaulow", "gender": "female", "grade": 5,
            "pets": [{ "givenName": "Fluffy" }]
        }
    ],
    "address": { "state": "WA", "county": "King", "city": "seattle" },
    "isRegistered": true
}
```

6. Add another document, this time with the following content:

```
{
    "_id": "WakefieldFamily",
    "parents": [
        { "familyName": "Wakefield", "givenName": "Robin" },
        { "familyName": "Miller", "givenName": "Ben" }
    ],
    "children": [
        {
            "familyName": "Merriam",
            "givenName": "Jesse",
            "gender": "female", "grade": 1,
            "pets": [
                { "givenName": "Goofy" },
                { "givenName": "Shadow" }
            ]
        },
        {
            "familyName": "Miller",
            "givenName": "Lisa",
            "gender": "female",
            "grade": 8 }
    ],
    "address": { "state": "NY", "county": "Manhattan", "city": "NY" },
    "isRegistered": false
}
```

7. Execute a sample query. For example, search for families with the last name 'Andersen' and return the parents and state fields.

Next steps

- Learn how to use Robo 3T with Azure Cosmos DB's API for MongoDB.
- Explore MongoDB samples with Azure Cosmos DB's API for MongoDB.

Use MongoDB Compass to connect to Azure Cosmos DB's API for MongoDB

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: Azure Cosmos DB API for MongoDB

This tutorial demonstrates how to use MongoDB Compass when storing and/or managing data in Cosmos DB. We use the Azure Cosmos DB's API for MongoDB for this walk-through. For those of you unfamiliar, Compass is a GUI for MongoDB. It is commonly used to visualize your data, run ad-hoc queries, along with managing your data.

Cosmos DB is Microsoft's globally distributed multi-model database service. You can quickly create and query document, key/value, and graph databases, all of which benefit from the global distribution and horizontal scale capabilities at the core of Cosmos DB.

Pre-requisites

To connect to your Cosmos DB account using MongoDB Compass, you must:

- Download and install Compass
- Have your Cosmos DB connection string information

Connect to Cosmos DB's API for MongoDB

To connect your Cosmos DB account to Compass, you can follow the below steps:

1. Retrieve the connection information for your Cosmos account configured with Azure Cosmos DB's API MongoDB using the instructions here.

2. Click on the button that says **Copy to clipboard** next to your **Primary/Secondary connection string** in Cosmos DB. Clicking this button will copy your entire connection string to your clipboard.

3. Open Compass on your desktop/machine and click on **Connect** and then **Connect to....**

4. Compass will automatically detect a connection string in the clipboard, and will prompt to ask whether you wish to use that to connect. Click on **Yes** as shown in the screenshot below.

5. Upon clicking **Yes** in the above step, your details from the connection string will be automatically populated. Remove the value automatically populated in the **Replica Set Name** field to ensure that is left blank.

6. Click on **Connect** at the bottom of the page. Your Cosmos DB account and databases should now be visible within MongoDB Compass.

Next steps

- Learn how to use Studio 3T with Azure Cosmos DB's API for MongoDB.
- Explore MongoDB samples with Azure Cosmos DB's API for MongoDB.

Use Robo 3T with Azure Cosmos DB's API for MongoDB

11/2/2020 • 2 minutes to read • Edit Online

APPLIES TO: Azure Cosmos DB API for MongoDB

To connect to Cosmos account using Robo 3T, you must:

- Download and install Robo 3T
- Have your Cosmos DB connection string information

> **NOTE**
>
> Currently, Robo 3T v1.2 and lower versions are supported with Cosmos DB's API for MongoDB.

Connect using Robo 3T

To add your Cosmos account to the Robo 3T connection manager, perform the following steps:

1. Retrieve the connection information for your Cosmos account configured with Azure Cosmos DB's API MongoDB using the instructions here.

2. Run the *Robomongo* application.

3. Click the connection button under **File** to manage your connections. Then, click **Create** in the **MongoDB Connections** window, which will open up the **Connection Settings** window.

4. In the **Connection Settings** window, choose a name. Then, find the **Host** and **Port** from your connection information in Step 1 and enter them into **Address** and **Port**, respectively.

5. On the **Authentication** tab, click **Perform authentication**. Then, enter your Database (default is *Admin*), **User Name** and **Password**. Both **User Name** and **Password** can be found in your connection information in Step 1.

6. On the **SSL** tab, check **Use SSL protocol**, then change the **Authentication Method** to **Self-signed Certificate**.

7. Finally, click **Test** to verify that you are able to connect, then **Save**.

Next steps

- Learn how to use Studio 3T with Azure Cosmos DB's API for MongoDB.
- Explore MongoDB samples with Azure Cosmos DB's API for MongoDB.

Connect a Node.js Mongoose application to Azure Cosmos DB

12/4/2020 • 10 minutes to read • Edit Online

APPLIES TO: Azure Cosmos DB API for MongoDB

This tutorial demonstrates how to use the Mongoose Framework when storing data in Cosmos DB. We use the Azure Cosmos DB's API for MongoDB for this walkthrough. For those of you unfamiliar, Mongoose is an object modeling framework for MongoDB in Node.js and provides a straight-forward, schema-based solution to model your application data.

Cosmos DB is Microsoft's globally distributed multi-model database service. You can quickly create and query document, key/value, and graph databases, all of which benefit from the global distribution and horizontal scale capabilities at the core of Cosmos DB.

Prerequisites

If you don't have an Azure subscription, create a free account before you begin.

You can Try Azure Cosmos DB for free without an Azure subscription, free of charge and commitments, or create an Azure Cosmos DB free tier account, with the first 400 RU/s and 5 GB of storage for free. You can also use the Azure Cosmos DB Emulator with a URI of `https://localhost:8081`. For the key to use with the emulator, see Authenticating requests.

Node.js version v0.10.29 or higher.

Create a Cosmos account

Let's create a Cosmos account. If you already have an account you want to use, you can skip ahead to Set up your Node.js application. If you are using the Azure Cosmos DB Emulator, follow the steps at Azure Cosmos DB Emulator to set up the emulator and skip ahead to Set up your Node.js application.

1. In a new browser window, sign in to the Azure portal.

2. In the left menu, select **Create a resource**.

3. On the **New** page, select **Databases** > **Azure Cosmos DB**.

4. On the **Create Azure Cosmos DB Account** page, enter the settings for the new Azure Cosmos DB account.

SETTING	VALUE	DESCRIPTION
Subscription	Your subscription	Select the Azure subscription that you want to use for this Azure Cosmos DB account.
Resource Group	Create new Then enter the same name as Account Name	Select **Create new**. Then enter a new resource group name for your account. For simplicity, use the same name as your Azure Cosmos DB account name.
Account Name	Enter a unique name	Enter a unique name to identify your Azure Cosmos DB account. Your account URI will be *mongo.cosmos.azure.com* appended to your unique account name. The account name can use only lowercase letters, numbers, and hyphens (-), and must be between 3 and 31 characters long.

SETTING	VALUE	DESCRIPTION
API	Azure Cosmos DB for Mongo DB API	The API determines the type of account to create. Azure Cosmos DB provides five APIs: Core (SQL) for document databases, Gremlin for graph databases, Azure Cosmos DB for Mongo DB API for document databases, Azure Table, and Cassandra. Currently, you must create a separate account for each API. Select **Azure Cosmos DB for Mongo DB API** because in this quickstart you are creating a collection that works with MongoDB. Learn more about Azure Cosmos DB for MongoDB API.
Location	Select the region closest to your users	Select a geographic location to host your Azure Cosmos DB account. Use the location that's closest to your users to give them the fastest access to the data.
Capacity mode	Provisioned throughput or Serverless	Select **Provisioned throughput** to create an account in provisioned throughput mode. Select **Serverless** to create an account in serverless mode. **Note**: Only MongoDB API version 3.6 is supported by serverless accounts. Choosing 3.2 as the version will force the account in provisioned throughput mode.

Select **Review+Create**. You can skip the **Network** and **Tags** section.

Home > New >

Create Azure Cosmos DB Account

Basics | Networking | Backup Policy | Encryption | Tags | Review + create

Azure Cosmos DB is a globally distributed, multi-model, fully managed database service. Try it for free, for 30 days with unlimited renewals. Go to production starting at $24/month per database, multiple containers included. Learn more

Project Details

Select the subscription to manage deployed resources and costs. Use resource groups like folders to organize and manage all your resources.

Subscription * : Visual Studio Enterprise

Resource Group * :

Create new

Instance Details

Account Name * : Enter account name

API * : Azure Cosmos DB for MongoDB API

Notebooks (Preview) : On | Off

Location * : (US) West US

Capacity mode : Provisioned throughput | Serverless (preview)

Learn more about capacity mode

With Azure Cosmos DB free tier, you will get 400 RU/s and 5 GB of storage for free in an account. You can enable free tier on up to one account per subscription. Estimated $24/month discount per account.

Apply Free Tier Discount : Apply | Do Not Apply

Account Type : Production | Non-Production

Version : 3.6

Geo-Redundancy : Enable | Disable

Multi-region Writes : Enable | Disable

*Up to 33% off multi-region writes is available to qualifying new accounts only. Offer limited to accounts with both account locations and geo-redundancy, and applies only to multi-region writes in those same regions. Both Geo-Redundancy and Multi-region Writes must be enabled in account settings. Actual discount will vary based on number of qualifying regions selected.

Review + create | Previous | Next: Networking

5. The account creation takes a few minutes. Wait for the portal to display the **Congratulations! Your Azure Cosmos DB for Mongo DB API account is ready** page.

Create a database

In this application we will cover two ways of creating collections in Azure Cosmos DB:

- **Storing each object model in a separate collection**: We recommend creating a database with dedicated throughput. Using this capacity model will give you better cost efficiency.

- **Storing all object models in a single Cosmos DB collection**: If you'd prefer to store all models in a single collection, you can just create a new database without selecting the Provision Throughput option. Using this capacity model will create each collection with its own throughput capacity for every object model.

After you create the database, you'll use the name in the `COSMOSDB_DBNAME` environment variable below.

Set up your Node.js application

> **NOTE**
> If you'd like to just walkthrough the sample code instead of setup the application itself, clone the sample used for this tutorial and build your Node.js Mongoose application on Azure Cosmos DB.

1. To create a Node.js application in the folder of your choice, run the following command in a node command prompt.

   `npm init`

   Answer the questions and your project will be ready to go.

2. Add a new file to the folder and name it `index.js`.

3. Install the necessary packages using one of the `npm install` options:

   - Mongoose: `npm install mongoose@5 --save`

     > **NOTE**
     > The Mongoose example connection below is based on Mongoose 5+, which has changed since earlier versions.

   - Dotenv (if you'd like to load your secrets from an .env file): `npm install dotenv --save`

     > **NOTE**
     > The `--save` flag adds the dependency to the package.json file.

4. Import the dependencies in your index.js file.

```
var mongoose = require('mongoose');
var env = require('dotenv').config();   //Use the .env file to load the variables
```

5. Add your Cosmos DB connection string and Cosmos DB Name to the `.env` file. Replace the placeholders {cosmos-account-name} and {dbname} with your own Cosmos account name and database name, without the brace symbols.

```
# You can get the following connection details from the Azure portal. You can find the details on the
Connection string pane of your Azure Cosmos account.

COSMOSDB_USER = "<Azure Cosmos account's user name, usually the database account name>"
COSMOSDB_PASSWORD = "<Azure Cosmos account password, this is one of the keys specified in your account>"
COSMOSDB_DBNAME = "<Azure Cosmos database name>"
COSMOSDB_HOST= "<Azure Cosmos Host name>"
COSMOSDB_PORT=10255
```

6. Connect to Cosmos DB using the Mongoose framework by adding the following code to the end of index.js.

```
mongoose.connect("mongodb://"+process.env.COSMOSDB_HOST+":"+process.env.COSMOSDB_PORT+"/"+process.env.CO
SMOSDB_DBNAME+"?ssl=true&replicaSet=globaldb", {
  auth: {
    user: process.env.COSMOSDB_USER,
    password: process.env.COSMOSDB_PASSWORD
  },
useNewUrlParser: true,
useUnifiedTopology: true,
retryWrites: false
})
.then(() => console.log('Connection to CosmosDB successful'))
.catch((err) => console.error(err));
```

> **NOTE**
>
> Here, the environment variables are loaded using process.env.{variableName} using the 'dotenv' npm package.

Once you are connected to Azure Cosmos DB, you can now start setting up object models in Mongoose.

Best practices for using Mongoose with Cosmos DB

For every model you create, Mongoose creates a new collection. This is best addressed using the Database Level Throughput option, which was previously discussed. To use a single collection, you need to use Mongoose Discriminators. Discriminators are a schema inheritance mechanism. They enable you to have multiple models with overlapping schemas on top of the same underlying MongoDB collection.

You can store the various data models in the same collection and then use a filter clause at query time to pull down only the data needed. Let's walk through each of the models.

One collection per object model

This section explores how to achieve this with Azure Cosmos DB's API for MongoDB. This method is our recommended approach since it allows you to control cost and capacity. As a result, the amount of Request Units on the database does not depend on the number of object models. This is the default operating model for Mongoose, so, you might be familiar with this.

1. Open your `index.js` again.
2. Create the schema definition for 'Family'.

```
const Family = mongoose.model('Family', new mongoose.Schema({
    lastName: String,
    parents: [{
        familyName: String,
        firstName: String,
        gender: String
    }],
    children: [{
        familyName: String,
        firstName: String,
        gender: String,
        grade: Number
    }],
    pets:[{
        givenName: String
    }],
    address: {
        country: String,
        state: String,
        city: String
    }
}));
```

3. Create an object for 'Family'.

```
const family = new Family({
    lastName: "Volum",
    parents: [
        { firstName: "Thomas" },
        { firstName: "Mary Kay" }
    ],
    children: [
        { firstName: "Ryan", gender: "male", grade: 8 },
        { firstName: "Patrick", gender: "male", grade: 7 }
    ],
    pets: [
        { givenName: "Buddy" }
    ],
    address: { country: "USA", state: "WA", city: "Seattle" }
});
```

4. Finally, let's save the object to Cosmos DB. This creates a collection underneath the covers.

```
family.save((err, saveFamily) => {
    console.log(JSON.stringify(saveFamily));
});
```

5. Now, let's create another schema and object. This time, let's create one for 'Vacation Destinations' that the families might be interested in.

   a. Just like last time, let's create the scheme

```
const VacationDestinations = mongoose.model('VacationDestinations', new mongoose.Schema({
 name: String,
 country: String
}));
```

   b. Create a sample object (You can add multiple objects to this schema) and save it.

```
const vacaySpot = new VacationDestinations({
 name: "Honolulu",
 country: "USA"
});

vacaySpot.save((err, saveVacay) => {
 console.log(JSON.stringify(saveVacay));
});
```

6. Now, going into the Azure portal, you notice two collections created in Cosmos DB.

7. Finally, let's read the data from Cosmos DB. Since we're using the default Mongoose operating model, the reads are the same as any other reads with Mongoose.

```
Family.find({ 'children.gender' : "male"}, function(err, foundFamily){
    foundFamily.forEach(fam => console.log("Found Family: " + JSON.stringify(fam)));
});
```

Using Mongoose discriminators to store data in a single collection

In this method, we use Mongoose Discriminators to help optimize for the costs of each collection. Discriminators allow you to define a differentiating 'Key', which allows you to store, differentiate and filter on different object models.

Here, we create a base object model, define a differentiating key and add 'Family' and 'VacationDestinations' as an extension to the base model.

1. Let's set up the base config and define the discriminator key.

```
const baseConfig = {
    discriminatorKey: "_type", //If you've got a lot of different data types, you could also consider
setting up a secondary index here.
    collection: "alldata"   //Name of the Common Collection
};
```

2. Next, let's define the common object model

```
const commonModel = mongoose.model('Common', new mongoose.Schema({}, baseConfig));
```

3. We now define the 'Family' model. Notice here that we're using `commonModel.discriminator` instead of `mongoose.model`. Additionally, we're also adding the base config to the mongoose schema. So, here, the discriminatorKey is `FamilyType`.

```
const Family_common = commonModel.discriminator('FamilyType', new     mongoose.Schema({
    lastName: String,
    parents: [{
        familyName: String,
        firstName: String,
        gender: String
    }],
    children: [{
        familyName: String,
        firstName: String,
       gender: String,
        grade: Number
    }],
    pets:[{
        givenName: String
    }],
    address: {
        country: String,
        state: String,
        city: String
    }
}, baseConfig));
```

4. Similarly, let's add another schema, this time for the 'VacationDestinations'. Here, the DiscriminatorKey is `VacationDestinationsType`.

```
const Vacation_common = commonModel.discriminator('VacationDestinationsType', new mongoose.Schema({
    name: String,
    country: String
}, baseConfig));
```

5. Finally, let's create objects for the model and save it.

   a. Let's add object(s) to the 'Family' model.

```
  const family_common = new Family_common({
   lastName: "Volum",
   parents: [
       { firstName: "Thomas" },
       { firstName: "Mary Kay" }
   ],
   children: [
       { firstName: "Ryan", gender: "male", grade: 8 },
       { firstName: "Patrick", gender: "male", grade: 7 }
   ],
   pets: [
       { givenName: "Buddy" }
   ],
   address: { country: "USA", state: "WA", city: "Seattle" }
  });

  family_common.save((err, saveFamily) => {
   console.log("Saved: " + JSON.stringify(saveFamily));
  });
```

   b. Next, let's add object(s) to the 'VacationDestinations' model and save it.

```
const vacay_common = new Vacation_common({
 name: "Honolulu",
 country: "USA"
});

vacay_common.save((err, saveVacay) => {
 console.log("Saved: " + JSON.stringify(saveVacay));
});
```

6. Now, if you go back to the Azure portal, you notice that you have only one collection called `alldata` with both 'Family' and 'VacationDestinations' data.

7. Also, notice that each object has another attribute called as `__type`, which help you differentiate between the two different object models.

8. Finally, let's read the data that is stored in Azure Cosmos DB. Mongoose takes care of filtering data based on the model. So, you have to do nothing different when reading data. Just specify your model (in this case, `Family_common`) and Mongoose handles filtering on the 'DiscriminatorKey'.

```
Family_common.find({ 'children.gender' : "male"}, function(err, foundFamily){
    foundFamily.forEach(fam => console.log("Found Family (using discriminator): " +
JSON.stringify(fam)));
});
```

As you can see, it is easy to work with Mongoose discriminators. So, if you have an app that uses the Mongoose framework, this tutorial is a way for you to get your application up and running using Azure Cosmos's API for MongoDB without requiring too many changes.

Clean up resources

When you're done with your app and Azure Cosmos DB account, you can delete the Azure resources you created so you don't incur more charges. To delete the resources:

1. In the Azure portal Search bar, search for and select **Resource groups**.

2. From the list, select the resource group you created for this quickstart.

3. On the resource group **Overview** page, select **Delete resource group**.

4. In the next window, enter the name of the resource group to delete, and then select **Delete**.

Next steps

- Learn how to use Studio 3T with Azure Cosmos DB's API for MongoDB.
- Learn how to use Robo 3T with Azure Cosmos DB's API for MongoDB.
- Explore MongoDB samples with Azure Cosmos DB's API for MongoDB.

How to globally distribute reads using Azure Cosmos DB's API for MongoDB

12/4/2020 • 4 minutes to read • Edit Online

APPLIES TO: Azure Cosmos DB API for MongoDB

This article shows how to globally distribute read operations with MongoDB Read Preference settings using Azure Cosmos DB's API for MongoDB.

Prerequisites

If you don't have an Azure subscription, create a free account before you begin.

Alternatively, you can Try Azure Cosmos DB for free without an Azure subscription, free of charge and commitments. Or you can use the Azure Cosmos DB Emulator for this tutorial with a connection string of:

```
mongodb://localhost:C2y6yDjf5/R+ob0N8A7Cgv30VRDJIWEHLM+4QDU5DE2nQ9nDuVTqobD4b8mGGyPMbIZnqyMsEcaGQy67XIw/Jw==@localhost:10255/admin?ssl=true
```

Refer to this Quickstart article for instructions on using the Azure portal to set up a Cosmos account with global distribution and then connect to it.

Clone the sample application

Open a git terminal window, such as git bash, and `cd` to a working directory.

Run the following commands to clone the sample repository. Based on your platform of interest, use one of the following sample repositories:

1. .NET sample application
2. NodeJS sample application
3. Mongoose sample application
4. Java sample application
5. SpringBoot sample application

```
git clone <sample repo url>
```

Run the application

Depending on the platform used, install the required packages and start the application. To install dependencies, follow the README included in the sample application repository. For instance, in the NodeJS sample application, use the following commands to install the required packages and start the application.

```
cd mean
npm install
node index.js
```

The application tries to connect to a MongoDB source and fails because the connection string is invalid. Follow the steps in the README to update the connection string `url`. Also, update the `readFromRegion` to a read region in your Cosmos account. The following instructions are from the NodeJS sample:

```
* Next, substitute the `url`, `readFromRegion` in App.Config with your Cosmos account's values.
```

After following these steps, the sample application runs and produces the following output:

```
connected!
readDefaultfunc query completed!
readFromNearestfunc query completed!
readFromRegionfunc query completed!
readDefaultfunc query completed!
readFromNearestfunc query completed!
readFromRegionfunc query completed!
readDefaultfunc query completed!
readFromSecondaryfunc query completed!
```

Read using Read Preference mode

MongoDB protocol provides the following Read Preference modes for clients to use:

1. PRIMARY
2. PRIMARY_PREFERRED
3. SECONDARY
4. SECONDARY_PREFERRED
5. NEAREST

Refer to the detailed MongoDB Read Preference behavior documentation for details on the behavior of each of these read preference modes. In Cosmos DB, primary maps to WRITE region and secondary maps to READ region.

Based on common scenarios, we recommend using the following settings:

1. If **low latency reads** are required, use the **NEAREST** read preference mode. This setting directs the read operations to the nearest available region. Note that if the nearest region is the WRITE region, then these operations are directed to that region.
2. If **high availability and geo distribution of reads** are required (latency is not a constraint), then use the **PRIMARY PREFERRED** or **SECONDARY PREFERRED** read preference mode. This setting directs read operations to an available WRITE or READ region respectively. If the region is not available, then requests are directed to the next available region as per the read preference behavior.

The following snippet from the sample application shows how to configure NEAREST Read Preference in NodeJS:

```
var query = {};
var readcoll = client.db('regionDB').collection('regionTest', {readPreference: ReadPreference.NEAREST});
readcoll.find(query).toArray(function(err, data) {
  assert.equal(null, err);
  console.log("readFromNearestfunc query completed!");
});
```

Similarly, the snippet below shows how to configure the SECONDARY_PREFERRED Read Preference in NodeJS:

```
var query = {};
var readcoll = client.db('regionDB').collection('regionTest', {readPreference: ReadPreference.SECONDARY_PREFERRED});
readcoll.find(query).toArray(function(err, data) {
  assert.equal(null, err);
  console.log("readFromSecondaryPreferredfunc query completed!");
});
```

The Read Preference can also be set by passing `readPreference` as a parameter in the connection string URI options:

```
const MongoClient = require('mongodb').MongoClient;
const assert = require('assert');

// Connection URL
const url = 'mongodb://localhost:27017?ssl=true&replicaSet=globaldb&readPreference=nearest';

// Database Name
const dbName = 'myproject';

// Use connect method to connect to the Server
MongoClient.connect(url, function(err, client) {
  console.log("Connected correctly to server");

  const db = client.db(dbName);

  client.close();
});
```

Refer to the corresponding sample application repos for other platforms, such as .NET and Java.

Read using tags

In addition to the Read Preference mode, MongoDB protocol allows the use of tags to direct read operations. In Cosmos DB's API for MongoDB, the `region` tag is included by default as a part of the `isMaster` response:

```
"tags": {
         "region": "West US"
      }
```

Hence, MongoClient can use the `region` tag along with the region name to direct read operations to specific regions. For Cosmos accounts, region names can be found in Azure portal on the left under **Settings->Replica data globally**. This setting is useful for achieving **read isolation** - cases in which client application want to direct read operations to a specific region only. This setting is ideal for non-production/analytics type scenarios, which run in the background and are not production critical services.

The following snippet from the sample application shows how to configure the Read Preference with tags in NodeJS:

```
 var query = {};
  var readcoll = client.db('regionDB').collection('regionTest',{readPreference: new
ReadPreference(ReadPreference.SECONDARY_PREFERRED, {"region": "West US"})});
  readcoll.find(query).toArray(function(err, data) {
    assert.equal(null, err);
    console.log("readFromRegionfunc query completed!");
  });
```

Refer to the corresponding sample application repos for other platforms, such as .NET and Java.

In this article, you've learned how to globally distribute read operations using Read Preference with Azure Cosmos DB's API for MongoDB.

Clean up resources

If you're not going to continue to use this app, delete all resources created by this article in the Azure portal with the following steps:

1. From the left-hand menu in the Azure portal, click **Resource groups** and then click the name of the resource you created.
2. On your resource group page, click **Delete**, type the name of the resource to delete in the text box, and then click **Delete**.

Next steps

- Import MongoDB data into Azure Cosmos DB

- Setup a globally distributed database with Azure Cosmos DB's API for MongoDB
- Develop locally with the Azure Cosmos DB emulator

Expire data with Azure Cosmos DB's API for MongoDB

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: Azure Cosmos DB API for MongoDB

Time-to-live (TTL) functionality allows the database to automatically expire data. Azure Cosmos DB's API for MongoDB utilizes Cosmos DB's core TTL capabilities. Two modes are supported: setting a default TTL value on the whole collection, and setting individual TTL values for each document. The logic governing TTL indexes and per-document TTL values in Cosmos DB's API for MongoDB is the same as in Cosmos DB.

TTL indexes

To enable TTL universally on a collection, a "TTL index" (time-to-live index) needs to be created. The TTL index is an index on the _ts field with an "expireAfterSeconds" value.

Example:

```
globaldb:PRIMARY> db.coll.createIndex({"_ts":1}, {expireAfterSeconds: 10})
{
        "_t" : "CreateIndexesResponse",
        "ok" : 1,
        "createdCollectionAutomatically" : true,
        "numIndexesBefore" : 1,
        "numIndexesAfter" : 4
}
```

The command in the above example will create an index with TTL functionality. Once the index is created, the database will automatically delete any documents in that collection that have not been modified in the last 10 seconds.

> **NOTE**
>
> **_ts** is a Cosmos DB-specific field and is not accessible from MongoDB clients. It is a reserved (system) property that contains the timestamp of the document's last modification.

Additionally, a C# example:

```
var options = new CreateIndexOptions {ExpireAfter = TimeSpan.FromSeconds(10)};
var field = new StringFieldDefinition<BsonDocument>("_ts");
var indexDefinition = new IndexKeysDefinitionBuilder<BsonDocument>().Ascending(field);
await collection.Indexes.CreateOneAsync(indexDefinition, options);
```

Set time to live value for a document

Per-document TTL values are also supported. The document(s) must contain a root-level property "ttl" (lower-case), and a TTL index as described above must have been created for that collection. TTL values set on a document will override the collection's TTL value.

The TTL value must be an int32. Alternatively, an int64 that fits in an int32, or a double with no decimal part that

fits in an int32. Values for the TTL property that do not conform to these specifications are allowed but not treated as a meaningful document TTL value.

The TTL value for the document is optional; documents without a TTL value can be inserted into the collection. In this case, the collection's TTL value will be honored.

The following documents have valid TTL values. Once the documents are inserted, the document TTL values override the collection's TTL values. So, the documents will be removed after 20 seconds.

```
globaldb:PRIMARY> db.coll.insert({id:1, location: "Paris", ttl: 20.0})
globaldb:PRIMARY> db.coll.insert({id:1, location: "Paris", ttl: NumberInt(20)})
globaldb:PRIMARY> db.coll.insert({id:1, location: "Paris", ttl: NumberLong(20)})
```

The following documents have invalid TTL values. The documents will be inserted, but the document TTL value will not be honored. So, the documents will be removed after 10 seconds because of the collection's TTL value.

```
globaldb:PRIMARY> db.coll.insert({id:1, location: "Paris", ttl: 20.5}) //TTL value contains non-zero decimal
part.
globaldb:PRIMARY> db.coll.insert({id:1, location: "Paris", ttl: NumberLong(2147483649)}) //TTL value is
greater than Int32.MaxValue (2,147,483,648).
```

How to activate the per-document TTL feature

The per-document TTL feature can be activated with Azure Cosmos DB's API for MongoDB.

Next steps

- Expire data in Azure Cosmos DB automatically with time to live
- Indexing your Cosmos database configured with Azure Cosmos DB's API for MongoDB

Change streams in Azure Cosmos DB's API for MongoDB

11/2/2020 • 4 minutes to read • Edit Online

APPLIES TO: Azure Cosmos DB API for MongoDB

Change feed support in Azure Cosmos DB's API for MongoDB is available by using the change streams API. By using the change streams API, your applications can get the changes made to the collection or to the items in a single shard. Later you can take further actions based on the results. Changes to the items in the collection are captured in the order of their modification time and the sort order is guaranteed per shard key.

> **NOTE**
>
> To use change streams, create the account with version 3.6 of Azure Cosmos DB's API for MongoDB, or a later version. If you run the change stream examples against an earlier version, you might see the `Unrecognized pipeline stage name: $changeStream` error.

Examples

The following example shows how to get change streams on all the items in the collection. This example creates a cursor to watch items when they are inserted, updated, or replaced. The `$match` stage, `$project` stage, and `fullDocument` option are required to get the change streams. Watching for delete operations using change streams is currently not supported. As a workaround, you can add a soft marker on the items that are being deleted. For example, you can add an attribute in the item called "deleted." When you'd like to delete the item, you can set "deleted" to `true` and set a TTL on the item. Since updating "deleted" to `true` is an update, this change will be visible in the change stream.

- JavaScript
- C#
- Java

```
var cursor = db.coll.watch(
    [
        { $match: { "operationType": { $in: ["insert", "update", "replace"] } } },
        { $project: { "_id": 1, "fullDocument": 1, "ns": 1, "documentKey": 1 } }
    ],
    { fullDocument: "updateLookup" });

while (!cursor.isExhausted()) {
    if (cursor.hasNext()) {
        printjson(cursor.next());
    }
}
```

Changes within a single shard

The following example shows how to get changes to the items within a single shard. This example gets the changes of items that have shard key equal to "a" and the shard key value equal to "1". It is possible to have different clients reading changes from different shards in parallel.

```
var cursor = db.coll.watch(
    [
        {
            $match: {
                $and: [
                    { "fullDocument.a": 1 },
                    { "operationType": { $in: ["insert", "update", "replace"] } }
                ]
            }
        },
        { $project: { "_id": 1, "fullDocument": 1, "ns": 1, "documentKey": 1} }
    ],
    { fullDocument: "updateLookup" });
```

Current limitations

The following limitations are applicable when using change streams:

- The `operationType` and `updateDescription` properties are not yet supported in the output document.
- The `insert`, `update`, and `replace` operations types are currently supported. However, the delete operation or other events are not yet supported.

Due to these limitations, the $match stage, $project stage, and fullDocument options are required as shown in the previous examples.

Unlike the change feed in Azure Cosmos DB's SQL API, there is not a separate Change Feed Processor Library to consume change streams or a need for a leases container. There is not currently support for Azure Functions triggers to process change streams.

Error handling

The following error codes and messages are supported when using change streams:

- **HTTP error code 16500** - When the change stream is throttled, it returns an empty page.
- **NamespaceNotFound (OperationType Invalidate)** - If you run change stream on the collection that does not exist or if the collection is dropped, then a `NamespaceNotFound` error is returned. Because the `operationType` property can't be returned in the output document, instead of the `operationType Invalidate` error, the `NamespaceNotFound` error is returned.

Next steps

- Use time to live to expire data automatically in Azure Cosmos DB's API for MongoDB
- Indexing in Azure Cosmos DB's API for MongoDB

Manage indexing in Azure Cosmos DB's API for MongoDB

12/4/2020 • 10 minutes to read • Edit Online

APPLIES TO: Azure Cosmos DB API for MongoDB

Azure Cosmos DB's API for MongoDB takes advantage of the core index-management capabilities of Azure Cosmos DB. This article focuses on how to add indexes using Azure Cosmos DB's API for MongoDB. You can also read an overview of indexing in Azure Cosmos DB that's relevant across all APIs.

Indexing for MongoDB server version 3.6

Azure Cosmos DB's API for MongoDB server version 3.6 automatically indexes the `_id` field, which can't be dropped. It automatically enforces the uniqueness of the `_id` field per shard key. In Azure Cosmos DB's API for MongoDB, sharding and indexing are separate concepts. You don't have to index your shard key. However, as with any other property in your document, if this property is a common filter in your queries, we recommend indexing the shard key.

To index additional fields, you apply the MongoDB index-management commands. As in MongoDB, Azure Cosmos DB's API for MongoDB automatically indexes the `_id` field only. This default indexing policy differs from the Azure Cosmos DB SQL API, which indexes all fields by default.

To apply a sort to a query, you must create an index on the fields used in the sort operation.

Index types

Single field

You can create indexes on any single field. The sort order of the single field index does not matter. The following command creates an index on the field `name`:

```
db.coll.createIndex({name:1})
```

One query uses multiple single field indexes where available. You can create up to 500 single field indexes per container.

Compound indexes (MongoDB server version 3.6)

Azure Cosmos DB's API for MongoDB supports compound indexes for accounts that use the version 3.6 wire protocol. You can include up to eight fields in a compound index. Unlike in MongoDB, you should create a compound index only if your query needs to sort efficiently on multiple fields at once. For queries with multiple filters that don't need to sort, create multiple single field indexes instead of a single compound index.

> **NOTE**
>
> You can't create compound indexes on nested properties or arrays.

The following command creates a compound index on the fields `name` and `age`:

```
db.coll.createIndex({name:1,age:1})
```

You can use compound indexes to sort efficiently on multiple fields at once, as shown in the following example:

```
db.coll.find().sort({name:1,age:1})
```

You can also use the preceding compound index to efficiently sort on a query with the opposite sort order on all fields. Here's an example:

```
db.coll.find().sort({name:-1,age:-1})
```

However, the sequence of the paths in the compound index must exactly match the query. Here's an example of a query that would require an additional compound index:

```
db.coll.find().sort({age:1,name:1})
```

> **NOTE**
> Compound indexes are only used in queries that sort results. For queries that have multiple filters that don't need to sort, create multipe single field indexes.

Multikey indexes

Azure Cosmos DB creates multikey indexes to index content stored in arrays. If you index a field with an array value, Azure Cosmos DB automatically indexes every element in the array.

Geospatial indexes

Many geospatial operators will benefit from geospatial indexes. Currently, Azure Cosmos DB's API for MongoDB supports `2dsphere` indexes. The API does not yet support `2d` indexes.

Here's an example of creating a geospatial index on the `location` field:

```
db.coll.createIndex({ location : "2dsphere" })
```

Text indexes

Azure Cosmos DB's API for MongoDB does not currently support text indexes. For text search queries on strings, you should use Azure Cognitive Search integration with Azure Cosmos DB.

Wildcard indexes

You can use wildcard indexes to support queries against unknown fields. Let's imagine you have a collection that holds data about families.

Here is part of an example document in that collection:

```
"children": [
   {
       "firstName": "Henriette Thaulow",
       "grade": "5"
   }
]
```

Here's another example, this time with a slightly different set of properties in `children`:

```
"children": [
    {
      "familyName": "Merriam",
      "givenName": "Jesse",
      "pets": [
          { "givenName": "Goofy" },
          { "givenName": "Shadow" }
    },
    {
      "familyName": "Merriam",
      "givenName": "John",
    }
]
```

In this collection, documents can have many different possible properties. If you wanted to index all the data in the `children` array, you have two options: create separate indexes for each individual property or create one wildcard index for the entire `children` array.

Create a wildcard index

The following command creates a wildcard index on any properties within `children`:

```
db.coll.createIndex({"children.$**" : 1})
```

Unlike in MongoDB, wildcard indexes can support multiple fields in query predicates. There will not be a difference in query performance if you use one single wildcard index instead of creating a separate index for each property.

You can create the following index types using wildcard syntax:

- Single field
- Geospatial

Indexing all properties

Here's how you can create a wildcard index on all fields:

```
db.coll.createIndex( { "$**" : 1 } )
```

> **NOTE**
> If you are just starting development, we **strongly** recommend starting off with a wildcard index on all fields. This can simplify development and make it easier to optimize queries.

Documents with many fields may have a high Request Unit (RU) charge for writes and updates. Therefore, if you have a write-heavy workload, you should opt to individually index paths as opposed to using wildcard indexes.

Limitations

Wildcard indexes do not support any of the following index types or properties:

- Compound
- TTL
- Unique

Unlike in MongoDB, in Azure Cosmos DB's API for MongoDB you **can't** use wildcard indexes for:

- Creating a wildcard index that includes multiple specific fields

```
db.coll.createIndex( { "$**" : 1 }, { "wildcardProjection " : { "children.givenName" : 1, "children.grade" :
1 } } )
```

- Creating a wildcard index that excludes multiple specific fields

```
db.coll.createIndex( { "$**" : 1 }, { "wildcardProjection" : { "children.givenName" : 0, "children.grade" : 0
} } )
```

As an alternative, you could create multiple wildcard indexes.

Index properties

The following operations are common for accounts serving wire protocol version 3.6 and accounts serving earlier versions. You can learn more about supported indexes and indexed properties.

Unique indexes

Unique indexes are useful for enforcing that two or more documents do not contain the same value for indexed fields.

> **IMPORTANT**
>
> Unique indexes can be created only when the collection is empty (contains no documents).

The following command creates a unique index on the field `student_id`:

```
globaldb:PRIMARY> db.coll.createIndex( { "student_id" : 1 }, {unique:true} )
{
        "_t" : "CreateIndexesResponse",
        "ok" : 1,
        "createdCollectionAutomatically" : false,
        "numIndexesBefore" : 1,
        "numIndexesAfter" : 4
}
```

For sharded collections, you must provide the shard (partition) key to create a unique index. In other words, all unique indexes on a sharded collection are compound indexes where one of the fields is the partition key.

The following commands create a sharded collection `coll` (the shard key is `university`) with a unique index on the fields `student_id` and `university`:

```
globaldb:PRIMARY> db.runCommand({shardCollection: db.coll._fullName, key: { university: "hashed"}});
{
        "_t" : "ShardCollectionResponse",
        "ok" : 1,
        "collectionsharded" : "test.coll"
}
globaldb:PRIMARY> db.coll.createIndex( { "university" : 1, "student_id" : 1 }, {unique:true});
{
        "_t" : "CreateIndexesResponse",
        "ok" : 1,
        "createdCollectionAutomatically" : false,
        "numIndexesBefore" : 3,
        "numIndexesAfter" : 4
}
```

In the preceding example, omitting the `"university":1` clause returns an error with the following message:

```
"cannot create unique index over {student_id : 1.0} with shard key pattern { university : 1.0 }"
```

TTL indexes

To enable document expiration in a particular collection, you need to create a time-to-live (TTL) index. A TTL index

is an index on the `_ts` field with an `expireAfterSeconds` value.

Example:

```
globaldb:PRIMARY> db.coll.createIndex({"_ts":1}, {expireAfterSeconds: 10})
```

The preceding command deletes any documents in the `db.coll` collection that have not been modified in the last 10 seconds.

> **NOTE**
> The **_ts** field is specific to Azure Cosmos DB and is not accessible from MongoDB clients. It is a reserved (system) property that contains the time stamp of the document's last modification.

Track index progress

Version 3.6 of Azure Cosmos DB's API for MongoDB supports the `currentOp()` command to track index progress on a database instance. This command returns a document that contains information about in-progress operations on a database instance. You use the `currentOp` command to track all in-progress operations in native MongoDB. In Azure Cosmos DB's API for MongoDB, this command only supports tracking the index operation.

Here are some examples that show how to use the `currentOp` command to track index progress:

- Get the index progress for a collection:

  ```
  db.currentOp({"command.createIndexes": <collectionName>, "command.$db": <databaseName>})
  ```

- Get the index progress for all collections in a database:

  ```
  db.currentOp({"command.$db": <databaseName>})
  ```

- Get the index progress for all databases and collections in an Azure Cosmos account:

  ```
  db.currentOp({"command.createIndexes": { $exists : true } })
  ```

Examples of index progress output

The index progress details show the percentage of progress for the current index operation. Here's an example that shows the output document format for different stages of index progress:

- An index operation on a "foo" collection and "bar" database that is 60 percent complete will have the following output document. The `Inprog[0].progress.total` field shows 100 as the target completion percentage.

```
{
      "inprog" : [
      {
              ……................
              "command" : {
                      "createIndexes" : foo
                      "indexes" :[ ],
                      "$db" : bar
              },
              "msg" : "Index Build (background) Index Build (background): 60 %",
              "progress" : {
                      "done" : 60,
                      "total" : 100
              },
              …….....……..
      }
      ],
      "ok" : 1
}
```

- If an index operation has just started on a "foo" collection and "bar" database, the output document might show 0 percent progress until it reaches a measurable level.

```
{
      "inprog" : [
      {
              ……................
              "command" : {
                      "createIndexes" : foo
                      "indexes" :[ ],
                      "$db" : bar
              },
              "msg" : "Index Build (background) Index Build (background): 0 %",
              "progress" : {
                      "done" : 0,
                      "total" : 100
              },
              …….....……..
      }
      ],
     "ok" : 1
}
```

- When the in-progress index operation finishes, the output document shows empty `inprog` operations.

```
{
    "inprog" : [],
    "ok" : 1
}
```

Background index updates

Regardless of the value specified for the **Background** index property, index updates are always done in the background. Because index updates consume Request Units (RUs) at a lower priority than other database operations, index changes won't result in any downtime for writes, updates, or deletes.

There is no impact to read availability when adding a new index. Queries will only utilize new indexes once the index transformation is complete. During the index transformation, the query engine will continue to use existing indexes, so you'll observe similar read performance during the indexing transformation to what you had observed before initiating the indexing change. When adding new indexes, there is also no risk of incomplete or

inconsistent query results.

When removing indexes and immediately running queries the have filters on the dropped indexes, results might be inconsistent and incomplete until the index transformation finishes. If you remove indexes, the query engine does not provide consistent or complete results when queries filter on these newly removed indexes. Most developers do not drop indexes and then immediately try to query them so, in practice, this situation is unlikely.

> **NOTE**
>
> You can track index progress.

ReIndex command

The `reIndex` command will recreate all indexes on a collection. In most cases, this is unnecessary. However, in some rare cases, query performance may improve after running the `reIndex` command.

You can run the `reIndex` command using the following syntax:

```
db.runCommand({ reIndex: <collection> })
```

You can use the below syntax to check if you need to run the `reIndex` command:

```
db.runCommand({"customAction":"GetCollection",collection:<collection>, showIndexes:true})
```

Sample output:

```
{
        "database" : "myDB",
        "collection" : "myCollection",
        "provisionedThroughput" : 400,
        "indexes" : [
                {
                        "v" : 1,
                        "key" : {
                                "_id" : 1
                        },
                        "name" : "_id_",
                        "ns" : "myDB.myCollection",
                        "requiresReIndex" : true
                },
                {
                        "v" : 1,
                        "key" : {
                                "b.$**" : 1
                        },
                        "name" : "b.$**_1",
                        "ns" : "myDB.myCollection",
                        "requiresReIndex" : true
                }
        ],
        "ok" : 1
}
```

If `reIndex` is necessary, **requiresReIndex** will be true. If `reIndex` isn't necessary, this property will be omitted.

Migrate collections with indexes

Currently, you can only create unique indexes when the collection contains no documents. Popular MongoDB migration tools try to create the unique indexes after importing the data. To circumvent this issue, you can manually create the corresponding collections and unique indexes instead of allowing the migration tool to try.

(You can achieve this behavior for `mongorestore` by using the `--noIndexRestore` flag in the command line.)

Indexing for MongoDB version 3.2

Available indexing features and defaults are different for Azure Cosmos accounts that are compatible with version 3.2 of the MongoDB wire protocol. You can check your account's version and upgrade to version 3.6.

If you're using version 3.2, this section outlines key differences with version 3.6.

Dropping default indexes (version 3.2)

Unlike the 3.6 version of Azure Cosmos DB's API for MongoDB, version 3.2 indexes every property by default. You can use the following command to drop these default indexes for a collection (`coll`):

```
> db.coll.dropIndexes()
{ "_t" : "DropIndexesResponse", "ok" : 1, "nIndexesWas" : 3 }
```

After dropping the default indexes, you can add more indexes as you would in version 3.6.

Compound indexes (version 3.2)

Compound indexes hold references to multiple fields of a document. If you want to create a compound index, upgrade to version 3.6.

Wildcard indexes (version 3.2)

If you want to create a wildcard index, upgrade to version 3.6.

Next steps

- Indexing in Azure Cosmos DB
- Expire data in Azure Cosmos DB automatically with time to live
- To learn about the relationship between partitioning and indexing, see how to Query an Azure Cosmos container article.

Create a container in Azure Cosmos DB API for MongoDB

11/2/2020 • 2 minutes to read • Edit Online

APPLIES TO: Azure Cosmos DB API for MongoDB

This article explains the different ways to create a container in Azure Cosmos DB API for MongoDB. It shows how to create a container using Azure portal, Azure CLI, PowerShell, or supported SDKs. This article demonstrates how to create a container, specify the partition key, and provision throughput.

This article explains the different ways to create a container in Azure Cosmos DB API for MongoDB. If you are using a different API, see SQL API, Cassandra API, Gremlin API, and Table API articles to create the container.

> **NOTE**
> When creating containers, make sure you don't create two containers with the same name but different casing. That's because some parts of the Azure platform are not case-sensitive, and this can result in confusion/collision of telemetry and actions on containers with such names.

Create using Azure portal

1. Sign in to the Azure portal.
2. Create a new Azure Cosmos account, or selectan existing account.
3. Open the **Data Explorer** pane, and select **New Container**. Next, provide the following details:
   - Indicate whether you are creating a new database or using an existing one.
   - Enter a container ID.
   - Enter a shard key.
   - Enter a throughput to be provisioned (for example, 1000 RUs).
   - Select **OK**.

Create using .NET SDK

```
// Create a collection with a partition key by using Mongo Shell:
db.runCommand( { shardCollection: "myDatabase.myCollection", key: { myShardKey: "hashed" } } )
```

> **NOTE**
> MongoDB wire protocol does not understand the concept of Request Units. To create a new collection with provisioned throughput on it, use the Azure portal or Cosmos DB SDKs for SQL API.

If you encounter timeout exception when creating a collection, do a read operation to validate if the collection was created successfully. The read operation throws an exception until the collection create operation is successful. For the list of status codes supported by the create operation see the HTTP Status Codes for Azure Cosmos DB article.

Create using Azure CLI

Create a collection for Azure Cosmos DB for MongoDB API with Azure CLI. For a listing of all Azure CLI samples across all Azure Cosmos DB APIs see, Azure CLI samples for Azure Cosmos DB.

Create using PowerShell

Create a collection for Azure Cosmos DB for MongoDB API with PowerShell. For a listing of all PowerShell samples across all Azure Cosmos DB APIs see, PowerShell Samples

Create a container using Azure Resource Manager templates

Create a collection for Azure Cosmos DB for MongoDB API with Resource Manager template.

Next steps

- Partitioning in Azure Cosmos DB
- Request Units in Azure Cosmos DB
- Provision throughput on containers and databases
- Work with Azure Cosmos account

Provision database, container or autoscale throughput on Azure Cosmos DB API for MongoDB resources

11/2/2020 • 2 minutes to read • Edit Online

APPLIES TO: Azure Cosmos DB API for MongoDB

This article explains how to provision throughput in Azure Cosmos DB API for MongoDB. You can provision standard(manual) or autoscale throughput on a container, or a database and share it among the containers within the database. You can provision throughput using Azure portal, Azure CLI, or Azure Cosmos DB SDKs.

If you are using a different API, see SQL API, Cassandra API, Gremlin API articles to provision the throughput.

Azure portal

1. Sign in to the Azure portal.
2. Create a new Azure Cosmos account, or select an existing Azure Cosmos account.
3. Open the **Data Explorer** pane, and select **New Collection**. Next, provide the following details:
   - Indicate whether you are creating a new database or using an existing one. Select the **Provision database throughput** option if you want to provision throughput at the database level.
   - Enter a collection ID.
   - Enter a partition key value (for example, `/ItemID`).
   - Enter a throughput that you want to provision (for example, 1000 RUs).
   - Select **OK**.

NOTE

If you are provisioning throughput on a container in an Azure Cosmos account configured with the Azure Cosmos DB API for MongoDB, use `/myShardKey` for the partition key path.

.NET SDK

```
// refer to MongoDB .NET Driver
// https://docs.mongodb.com/drivers/csharp

// Create a new Client
String mongoConnectionString = "mongodb://DBAccountName:Password@DBAccountName.documents.azure.com:10255/?
ssl=true&replicaSet=globaldb";
mongoUrl = new MongoUrl(mongoConnectionString);
mongoClientSettings = MongoClientSettings.FromUrl(mongoUrl);
mongoClient = new MongoClient(mongoClientSettings);

// Change the database name
mongoDatabase = mongoClient.GetDatabase("testdb");

// Change the collection name, throughput value then update via MongoDB extension commands
// https://docs.microsoft.com/en-us/azure/cosmos-db/mongodb-custom-commands#update-collection

var result = mongoDatabase.RunCommand<BsonDocument>(@"{customAction: ""UpdateCollection"", collection:
""testcollection"", offerThroughput: 400}");
```

Azure Resource Manager

Azure Resource Manager templates can be used to provision autoscale throughput on database or container-level resources for all Azure Cosmos DB APIs. See Azure Resource Manager templates for Azure Cosmos DB for samples.

Azure CLI

Azure CLI can be used to provision autoscale throughput on a database or container-level resources for all Azure Cosmos DB APIs. For samples see Azure CLI Samples for Azure Cosmos DB.

Azure PowerShell

Azure PowerShell can be used to provision autoscale throughput on a database or container-level resources for all Azure Cosmos DB APIs. For samples see Azure PowerShell samples for Azure Cosmos DB.

Next steps

See the following articles to learn about throughput provisioning in Azure Cosmos DB:

- Request units and throughput in Azure Cosmos DB

Find the request unit charge for operations executed in Azure Cosmos DB API for MongoDB

11/2/2020 • 2 minutes to read • Edit Online

APPLIES TO: Azure Cosmos DB API for MongoDB

Azure Cosmos DB supports many APIs, such as SQL, MongoDB, Cassandra, Gremlin, and Table. Each API has its own set of database operations. These operations range from simple point reads and writes to complex queries. Each database operation consumes system resources based on the complexity of the operation.

The cost of all database operations is normalized by Azure Cosmos DB and is expressed by Request Units (or RUs, for short). You can think of RUs as a performance currency abstracting the system resources such as CPU, IOPS, and memory that are required to perform the database operations supported by Azure Cosmos DB. No matter which API you use to interact with your Azure Cosmos container, costs are always measured by RUs. Whether the database operation is a write, point read, or query, costs are always measured in RUs. To learn more, see the request units and it's considerations article.

This article presents the different ways you can find the request unit (RU) consumption for any operation executed against a container in Azure Cosmos DB API for MongoDB. If you are using a different API, see SQL API, Cassandra API, Gremlin API, and Table API articles to find the RU/s charge.

The RU charge is exposed by a custom database command named `getLastRequestStatistics`. The command returns a document that contains the name of the last operation executed, its request charge, and its duration. If you use the Azure Cosmos DB API for MongoDB, you have multiple options for retrieving the RU charge.

Use the Azure portal

1. Sign in to the Azure portal.
2. Create a new Azure Cosmos account and feed it with data, or select an existing account that already contains data.
3. Go to the **Data Explorer** pane, and then select the container you want to work on.
4. Select **New Query**.
5. Enter a valid query, and then select **Execute Query**.
6. Select **Query Stats** to display the actual request charge for the request you executed.

Use the MongoDB .NET driver

When you use the official MongoDB .NET driver, you can execute commands by calling the `RunCommand` method on a `IMongoDatabase` object. This method requires an implementation of the `Command<>` abstract class:

```
class GetLastRequestStatisticsCommand : Command<Dictionary<string, object>>
{
    public override RenderedCommand<Dictionary<string, object>> Render(IBsonSerializerRegistry
serializerRegistry)
    {
        return new RenderedCommand<Dictionary<string, object>>(new BsonDocument("getLastRequestStatistics",
1), serializerRegistry.GetSerializer<Dictionary<string, object>>());
    }
}

Dictionary<string, object> stats = database.RunCommand(new GetLastRequestStatisticsCommand());
double requestCharge = (double)stats["RequestCharge"];
```

For more information, see Quickstart: Build a .NET web app by using an Azure Cosmos DB API for MongoDB.

Use the MongoDB Java driver

When you use the official MongoDB Java driver, you can execute commands by calling the `runCommand` method on a `MongoDatabase` object:

```
Document stats = database.runCommand(new Document("getLastRequestStatistics", 1));
Double requestCharge = stats.getDouble("RequestCharge");
```

For more information, see Quickstart: Build a web app by using the Azure Cosmos DB API for MongoDB and the Java SDK.

Use the MongoDB Node.js driver

When you use the official MongoDB Node.js driver, you can execute commands by calling the `command` method on a `db` object:

```
db.command({ getLastRequestStatistics: 1 }, function(err, result) {
    assert.equal(err, null);
    const requestCharge = result['RequestCharge'];
});
```

For more information, see Quickstart: Migrate an existing MongoDB Node.js web app to Azure Cosmos DB.

Next steps

To learn about optimizing your RU consumption, see these articles:

- Request units and throughput in Azure Cosmos DB
- Optimize provisioned throughput cost in Azure Cosmos DB
- Optimize query cost in Azure Cosmos DB

Troubleshoot query issues when using the Azure Cosmos DB API for MongoDB

12/4/2020 • 8 minutes to read • Edit Online

APPLIES TO: Azure Cosmos DB API for MongoDB

This article walks through a general recommended approach for troubleshooting queries in Azure Cosmos DB. Although you shouldn't consider the steps outlined in this article a complete defense against potential query issues, we've included the most common performance tips here. You should use this article as a starting place for troubleshooting slow or expensive queries in Azure Cosmos DB's API for MongoDB. If you are using the Azure Cosmos DB core (SQL) API, see the SQL API query troubleshooting guide article.

Query optimizations in Azure Cosmos DB are broadly categorized as follows:

- Optimizations that reduce the Request Unit (RU) charge of the query
- Optimizations that just reduce latency

If you reduce the RU charge of a query, you'll typically decrease latency as well.

This article provides examples that you can re-create by using the nutrition dataset.

> **NOTE**
> This article assumes you are using version 3.6 of Azure Cosmos DB"s API for MongoDB. Some queries that perform poorly in version 3.2 have significant improvements in version 3.6. Upgrade to version 3.6 by filing a support request.

Use $explain command to get metrics

When you optimize a query in Azure Cosmos DB, the first step is always to obtain the RU charge for your query. As a rough guideline, you should explore ways to lower the RU charge for queries with charges greater than 50 RUs.

In addition to obtaining the RU charge, you should use the `$explain` command to obtain the query and index usage metrics. Here is an example that runs a query and uses the `$explain` command to show query and index usage metrics:

$explain command:

```
db.coll.find({foodGroup: "Baby Foods"}).explain({"executionStatistics": true })
```

Output:

```
{
    "stages" : [
        {
            "stage" : "$query",
            "timeInclusiveMS" : 905.2888,
            "timeExclusiveMS" : 905.2888,
            "in" : 362,
            "out" : 362,
            "details" : {
                "database" : "db-test",
                "collection" : "collection-test",
                "query" : {
                    "foodGroup" : {
                        "$eq" : "Baby Foods"
                    }
                },
                "pathsIndexed" : [],
                "pathsNotIndexed" : [
                    "foodGroup"
                ],
                "shardInformation" : [
                    {
                        "activityId" : "e68e6bdd-5e89-4ec5-b053-3dbbc2428140",
                        "shardKeyRangeId" : "0",
                        "durationMS" : 788.5867,
                        "preemptions" : 1,
                        "outputDocumentCount" : 362,
                        "retrievedDocumentCount" : 8618
                    }
                ],
                "queryMetrics" : {
                    "retrievedDocumentCount" : 8618,
                    "retrievedDocumentSizeBytes" : 104963042,
                    "outputDocumentCount" : 362,
                    "outputDocumentSizeBytes" : 2553535,
                    "indexHitRatio" : 0.0016802042237178,
                    "totalQueryExecutionTimeMS" : 777.72,
                    "queryPreparationTimes" : {
                        "queryCompilationTimeMS" : 0.19,
                        "logicalPlanBuildTimeMS" : 0.14,
                        "physicalPlanBuildTimeMS" : 0.09,
                        "queryOptimizationTimeMS" : 0.03
                    },
                    "indexLookupTimeMS" : 0,
                    "documentLoadTimeMS" : 687.22,
                    "vmExecutionTimeMS" : 774.09,
                    "runtimeExecutionTimes" : {
                        "queryEngineExecutionTimeMS" : 37.45,
                        "systemFunctionExecutionTimeMS" : 10.82,
                        "userDefinedFunctionExecutionTimeMS" : 0
                    },
                    "documentWriteTimeMS" : 49.42
                }
            }
        }
    ],
    "estimatedDelayFromRateLimitingInMilliseconds" : 0.0,
    "continuation" : {
        "hasMore" : false
    },
    "ok" : 1.0
}
```


The `$explain` command output is lengthy and has detailed information about query execution. In general, though, there are a few sections where you should focus when optimizing query performance:

METRIC	DESCRIPTION
`timeInclusiveMS`	Backend query latency
`pathsIndexed`	Shows indexes that the query used
`pathsNotIndexed`	Shows indexes that the query could have used, if available
`shardInformation`	Summary of query performance for a particular physical partition
`retrievedDocumentCount`	Number of documents loaded by the query engine
`outputDocumentCount`	Number of documents returned in the query results
`estimatedDelayFromRateLimitingInMilliseconds`	Estimated additional query latency due to rate limiting

After you get the query metrics, compare the `retrievedDocumentCount` with the `outputDocumentCount` for your query. Use this comparison to identify the relevant sections to review in this article. The `retrievedDocumentCount` is the number of documents that the query engine needs to load. The `outputDocumentCount` is the number of documents that were needed for the results of the query. If the `retrievedDocumentCount` is significantly higher than the `outputDocumentCount`, there was at least one part of your query that was unable to use an index and needed to do a scan.

Refer to the following sections to understand the relevant query optimizations for your scenario.

Query's RU charge is too high

Retrieved Document Count is significantly higher than Output Document Count

- Include necessary indexes.
- Understand which aggregation operations use the index.

Retrieved Document Count is approximately equal to Output Document Count

- Minimize cross partition queries.

Query's RU charge is acceptable but latency is still too high

- Improve proximity.
- Increase provisioned throughput.

Queries where retrieved document count exceeds output document count

The `retrievedDocumentCount` is the number of documents that the query engine needed to load. The `outputDocumentCount` is the number of documents returned by the query. If the `retrievedDocumentCount` is significantly higher than the `outputDocumentCount`, there was at least one part of your query that was unable to use an index and needed to do a scan.

Here's an example of scan query that wasn't entirely served by the index:

$explain command:

```
db.coll.find(
  {
    $and : [
            { "foodGroup" : "Cereal Grains and Pasta"},
            { "description" : "Oat bran, cooked"}
        ]
  }
).explain({"executionStatistics": true })
```

Output:


```
{
    "stages" : [
        {
            "stage" : "$query",
            "timeInclusiveMS" : 436.5716,
            "timeExclusiveMS" : 436.5716,
            "in" : 1,
            "out" : 1,
            "details" : {
                "database" : "db-test",
                "collection" : "indexing-test",
                "query" : {
                    "$and" : [
                        {
                            "foodGroup" : {
                                "$eq" : "Cereal Grains and Pasta"
                            }
                        },
                        {
                            "description" : {
                                "$eq" : "Oat bran, cooked"
                            }
                        }
                    ]
                },
                "pathsIndexed" : [],
                "pathsNotIndexed" : [
                    "foodGroup",
                    "description"
                ],
                "shardInformation" : [
                    {
                        "activityId" : "13a5977e-a10a-4329-b68e-87e4f0081cac",
                        "shardKeyRangeId" : "0",
                        "durationMS" : 435.4867,
                        "preemptions" : 1,
                        "outputDocumentCount" : 1,
                        "retrievedDocumentCount" : 8618
                    }
                ],
                "queryMetrics" : {
                    "retrievedDocumentCount" : 8618,
                    "retrievedDocumentSizeBytes" : 104963042,
                    "outputDocumentCount" : 1,
                    "outputDocumentSizeBytes" : 6064,
                    "indexHitRatio" : 0.0,
                    "totalQueryExecutionTimeMS" : 433.64,
                    "queryPreparationTimes" : {
                        "queryCompilationTimeMS" : 0.12,
                        "logicalPlanBuildTimeMS" : 0.09,
                        "physicalPlanBuildTimeMS" : 0.1,
                        "queryOptimizationTimeMS" : 0.02
                    },
                    "indexLookupTimeMS" : 0,
                    "documentLoadTimeMS" : 387.44,
```

```
                    "vmExecutionTimeMS" : 432.93,
                    "runtimeExecutionTimes" : {
                        "queryEngineExecutionTimeMS" : 45.36,
                        "systemFunctionExecutionTimeMS" : 16.86,
                        "userDefinedFunctionExecutionTimeMS" : 0
                    },
                    "documentWriteTimeMS" : 0.13
                }
            }
        }
    ],
    "estimatedDelayFromRateLimitingInMilliseconds" : 0.0,
    "continuation" : {
        "hasMore" : false
    },
    "ok" : 1.0
}
```

The `retrievedDocumentCount` (8618) is significantly higher than the `outputDocumentCount` (1), implying that this query required a document scan.

Include necessary indexes

You should check the `pathsNotIndexed` array and add these indexes. In this example, the paths `foodGroup` and `description` should be indexed.

```
"pathsNotIndexed" : [
                    "foodGroup",
                    "description"
                ]
```

Indexing best practices in Azure Cosmos DB's API for MongoDB are different from MongoDB. In Azure Cosmos DB's API for MongoDB, compound indexes are only used in queries that need to efficiently sort by multiple properties. If you have queries with filters on multiple properties, you should create single field indexes for each of these properties. Query predicates can use multiple single field indexes.

Wildcard indexes can simplify indexing. Unlike in MongoDB, wildcard indexes can support multiple fields in query predicates. There will not be a difference in query performance if you use one single wildcard index instead of creating a separate index for each property. Adding a wildcard index for all properties is the easiest way to optimize all of your queries.

You can add new indexes at any time, with no effect on write or read availability. You can track index transformation progress.

Understand which aggregation operations use the index

In most cases, aggregation operations in Azure Cosmos DB's API for MongoDB will partially use indexes. Typically, the query engine will apply equality and range filters first and use indexes. After applying these filters, the query engine can evaluate additional filters and resort to loading remaining documents to compute the aggregate, if needed.

Here's an example:

```
db.coll.aggregate( [
   { $match: { foodGroup: 'Fruits and Fruit Juices' } },
   {
     $group: {
        _id: "$foodGroup",
        total: { $max: "$version" }
     }
   }
] )
```

In this case, indexes can optimize the `$match` stage. Adding an index for `foodGroup` will significantly improve query performance. Like in MongoDB, you should place `$match` as early in the aggregation pipeline as possible to maximize usage of indexes.

In Azure Cosmos DB's API for MongoDB, indexes are not used for the actual aggregation, which in this case is `$max`. Adding an index on `version` will not improve query performance.

Queries where retrieved document count is equal to Output Document Count

If the `retrievedDocumentCount` is approximately equal to the `outputDocumentCount`, the query engine didn't have to scan many unnecessary documents.

Minimize cross partition queries

Azure Cosmos DB uses partitioning to scale individual containers as Request Unit and data storage needs increase. Each physical partition has a separate and independent index. If your query has an equality filter that matches your container's partition key, you'll need to check only the relevant partition's index. This optimization reduces the total number of RUs that the query requires. Learn more about the differences between in-partition queries and cross-partition queries.

If you have a large number of provisioned RUs (more than 30,000) or a large amount of data stored (more than approximately 100 GB), you probably have a large enough container to see a significant reduction in query RU charges.

You can check the `shardInformation` array to understand the query metrics for each individual physical partition. The number of unique `shardKeyRangeId` values is the number of physical partitions where the query needed to be executed. In this example, the query was executed on four physical partitions. It's important to understand that execution is completely independent from index utilization. In other words, cross-partition queries can still use indexes.

```
    "shardInformation" : [
                        {
                            "activityId" : "42f670a8-a201-4c58-8023-363ac18d9e18",
                            "shardKeyRangeId" : "5",
                            "durationMS" : 24.3859,
                            "preemptions" : 1,
                            "outputDocumentCount" : 463,
                            "retrievedDocumentCount" : 463
                        },
                        {
                            "activityId" : "a8bf762a-37b9-4c07-8ed4-ae49961373c0",
                            "shardKeyRangeId" : "2",
                            "durationMS" : 35.8328,
                            "preemptions" : 1,
                            "outputDocumentCount" : 905,
                            "retrievedDocumentCount" : 905
                        },
                        {
                            "activityId" : "3754e36b-4258-49a6-8d4d-010555628395",
                            "shardKeyRangeId" : "1",
                            "durationMS" : 67.3969,
                            "preemptions" : 1,
                            "outputDocumentCount" : 1479,
                            "retrievedDocumentCount" : 1479
                        },
                        {
                            "activityId" : "a69a44ee-db97-4fe9-b489-3791f3d52878",
                            "shardKeyRangeId" : "0",
                            "durationMS" : 185.1523,
                            "preemptions" : 1,
                            "outputDocumentCount" : 867,
                            "retrievedDocumentCount" : 867
                        }
                    ]
```

Optimizations that reduce query latency

In many cases, the RU charge might be acceptable when query latency is still too high. The following sections give an overview of tips for reducing query latency. If you run the same query multiple times on the same dataset, it will typically have the same RU charge each time. But query latency might vary between query executions.

Improve proximity

Queries that are run from a different region than the Azure Cosmos DB account will have higher latency than if they were run inside the same region. For example, if you're running code on your desktop computer, you should expect latency to be tens or hundreds of milliseconds higher (or more) than if the query came from a virtual machine within the same Azure region as Azure Cosmos DB. It's simple to globally distribute data in Azure Cosmos DB to ensure you can bring your data closer to your app.

Increase provisioned throughput

In Azure Cosmos DB, your provisioned throughput is measured in Request Units (RUs). Imagine you have a query that consumes 5 RUs of throughput. For example, if you provision 1,000 RUs, you would be able to run that query 200 times per second. If you tried to run the query when there wasn't enough throughput available, Azure Cosmos DB will rate limit the requests. Azure Cosmos DB's API for MongoDB will automatically retry this query after waiting for a short time. Throttled requests take longer, so increasing provisioned throughput can improve query latency.

The value `estimatedDelayFromRateLimitingInMilliseconds` gives a sense of the potential latency benefits if you increase throughput.

Next steps

- Troubleshoot query performance (SQL API)
- Manage indexing in Azure Cosmos DB's API for MongoDB

Use MongoDB extension commands to manage data stored in Azure Cosmos DB's API for MongoDB

12/4/2020 • 10 minutes to read • Edit Online

APPLIES TO: Azure Cosmos DB API for MongoDB

The following document contains the custom action commands that are specific to Azure Cosmos DB's API for MongoDB. These commands can be used to create and obtain database resources that are specific to the Azure Cosmos DB capacity model.

By using the Azure Cosmos DB's API for MongoDB, you can enjoy the benefits Cosmos DB such as global distribution, automatic sharding, high availability, latency guarantees, automatic, encryption at rest, backups, and many more, while preserving your investments in your MongoDB app. You can communicate with the Azure Cosmos DB's API for MongoDB by using any of the open-source MongoDB client drivers. The Azure Cosmos DB's API for MongoDB enables the use of existing client drivers by adhering to the MongoDB wire protocol.

MongoDB protocol support

Azure Cosmos DB's API for MongoDB is compatible with MongoDB server version 3.2 and 3.6. See supported features and syntax for more details.

The following extension commands provide the ability to create and modify Azure Cosmos DB-specific resources via database requests:

- Create database
- Update database
- Get database
- Create collection
- Update collection
- Get collection

Create database

The create database extension command creates a new MongoDB database. The database name can be used from the database context set by the `use database` command. The following table describes the parameters within the command:

FIELD	TYPE	DESCRIPTION
`customAction`	`string`	Name of the custom command, it must be "CreateDatabase".
`offerThroughput`	`int`	Provisioned throughput that you set on the database. This parameter is optional.

FIELD	TYPE	DESCRIPTION
`autoScaleSettings`	`Object`	Required for Autoscale mode. This object contains the settings associated with the Autoscale capacity mode. You can set up the `maxThroughput` value, which describes the highest amount of Request Units that the collection will be increased to dynamically.

Output

If the command is successful, it will return the following response:

```
{ "ok" : 1 }
```

See the default output of custom command for the parameters in the output.

Examples

Create a database

To create a database named `"test"` that uses all the default values, use the following command:

```
use test
db.runCommand({customAction: "CreateDatabase"});
```

This command will create a database without database-level throughput. This means that the collections within this database will need to specify the amount of throughput that you need to use.

Create a database with throughput

To create a database named `"test"` and to specify a database-level provisioned throughput of 1000 RUs, use the following command:

```
use test
db.runCommand({customAction: "CreateDatabase", offerThroughput: 1000 });
```

This will create a database and set a throughput to it. All collections within this database will share the set throughput, unless the collections are created with a specific throughput level.

Create a database with Autoscale throughput

To create a database named `"test"` and to specify an Autoscale max throughput of 20,000 RU/s at database-level, use the following command:

```
use test
db.runCommand({customAction: "CreateDatabase", autoScaleSettings: { maxThroughput: 20000 } });
```

Update database

The update database extension command updates the properties associated with the specified database. The following table describes the parameters within the command:

FIELD	TYPE	DESCRIPTION
`customAction`	`string`	Name of the custom command. Must be "UpdateDatabase".

FIELD	TYPE	DESCRIPTION
`offerThroughput`	`int`	New provisioned throughput that you want to set on the database if the database uses database-level throughput
`autoScaleSettings`	`Object`	Required for Autoscale mode. This object contains the settings associated with the Autoscale capacity mode. You can set up the `maxThroughput` value, which describes the highest amount of Request Units that the database will be increased to dynamically.

This command uses the database specified in the context of the session. This is the database you used in the `use <database>` command. At the moment, the database name can not be changed using this command.

Output

If the command is successful, it will return the following response:

```
{ "ok" : 1 }
```

See the default output of custom command for the parameters in the output.

Examples

Update the provisioned throughput associated with a database

To update the provisioned throughput of a database with name `"test"` to 1200 RUs, use the following command:

```
use test
db.runCommand({customAction: "UpdateDatabase", offerThroughput: 1200 });
```

Update the Autoscale throughput associated with a database

To update the provisioned throughput of a database with name `"test"` to 20,000 RUs, or to transform it to an Autoscale throughput level, use the following command:

```
use test
db.runCommand({customAction: "UpdateDatabase", autoScaleSettings: { maxThroughput: 20000 } });
```

Get database

The get database extension command returns the database object. The database name is used from the database context against which the command is executed.

```
{
  customAction: "GetDatabase"
}
```

The following table describes the parameters within the command:

FIELD	TYPE	DESCRIPTION

FIELD	TYPE	DESCRIPTION
`customAction`	`string`	Name of the custom command. Must be "GetDatabase"

Output

If the command succeeds, the response contains a document with the following fields:

FIELD	TYPE	DESCRIPTION
`ok`	`int`	Status of response. 1 == success. 0 == failure.
`database`	`string`	Name of the database.
`provisionedThroughput`	`int`	Provisioned throughput that is set on the database if the database is using manual database-level throughput
`autoScaleSettings`	`Object`	This object contains the capacity parameters associated with the database if it is using the Autoscale mode. The `maxThroughput` value describes the highest amount of Request Units that the database will be increased to dynamically.

If the command fails, a default custom command response is returned. See the default output of custom command for the parameters in the output.

Examples

Get the database

To get the database object for a database named `"test"`, use the following command:

```
use test
db.runCommand({customAction: "GetDatabase"});
```

If the database has no associated throughput, the output would be:

```
{ "database" : "test", "ok" : 1 }
```

If the database has a database-level manual throughput associated with it, the output would show the `provisionedThroughput` values:

```
{ "database" : "test", "provisionedThroughput" : 20000, "ok" : 1 }
```

If the database has a database-level Autoscale throughput associated with it, the output would show the `provisionedThroughput`, which describes the minimum RU/s for the database, and the `autoScaleSettings` object including the `maxThroughput`, which describes the maximum RU/s for the database.

```
{
        "database" : "test",
        "provisionedThroughput" : 2000,
        "autoScaleSettings" : {
                "maxThroughput" : 20000
        },
        "ok" : 1
}
```

Create collection

The create collection extension command creates a new MongoDB collection. The database name is used from the databases context set by the `use database` command. The format of the CreateCollection command is as follows:

```
{
  customAction: "CreateCollection",
  collection: "<Collection Name>",
  shardKey: "<Shard key path>",
  offerThroughput: (int), // Amount of throughput allocated to a specific collection

}
```

The following table describes the parameters within the command:

FIELD	TYPE	REQUIRED	DESCRIPTION
`customAction`	`string`	Required	Name of the custom command. Must be "CreateCollection".
`collection`	`string`	Required	Name of the collection. No special characters or spaces are allowed.
`offerThroughput`	`int`	Optional	Provisioned throughput to set on the database. If this parameter is not provided, it will default to the minimum, 400 RU/s. * To specify throughput beyond 10,000 RU/s, the `shardKey` parameter is required.
`shardKey`	`string`	Required for collections with large throughput	The path to the Shard Key for the sharded collection. This parameter is required if you set more than 10,000 RU/s in `offerThroughput`. If it is specified, all documents inserted will require this key and value.

FIELD	TYPE	REQUIRED	DESCRIPTION
`autoScaleSettings`	`Object`	Required for Autoscale mode	This object contains the settings associated with the Autoscale capacity mode. You can set up the `maxThroughput` value, which describes the highest amount of Request Units that the collection will be increased to dynamically.

Output

Returns a default custom command response. See the default output of custom command for the parameters in the output.

Examples

Create a collection with the minimum configuration

To create a new collection with name `"testCollection"` and the default values, use the following command:

```
use test
db.runCommand({customAction: "CreateCollection", collection: "testCollection"});
```

This will result in a new fixed, unsharded, collection with 400RU/s and an index on the `_id` field automatically created. This type of configuration will also apply when creating new collections via the `insert()` function. For example:

```
use test
db.newCollection.insert({});
```

Create a unsharded collection

To create a unsharded collection with name `"testCollection"` and provisioned throughput of 1000 RUs, use the following command:

```
use test
db.runCommand({customAction: "CreateCollection", collection: "testCollection", offerThroughput: 1000});
```

You can create a collection with up to 10,000 RU/s as the `offerThroughput` without needing to specify a shard key. For collections with larger throughput, check out the next section.

Create a sharded collection

To create a sharded collection with name `"testCollection"` and provisioned throughput of 11,000 RUs, and a `shardkey` property "a.b", use the following command:

```
use test
db.runCommand({customAction: "CreateCollection", collection: "testCollection", offerThroughput: 11000,
shardKey: "a.b" });
```

This command now requires the `shardKey` parameter, since more than 10,000 RU/s specified in the `offerThroughput`.

Create an unsharded Autoscale collection

To create an unsharded collection named `'testCollection'` that uses Autoscale throughput capacity set to 4,000 RU/s, use the following command:

```
use test
db.runCommand({
    customAction: "CreateCollection", collection: "testCollection",
    autoScaleSettings:{
      maxThroughput: 4000
    }
});
```

For the `autoScaleSettings.maxThroughput` value you can specify a range from 4,000 RU/s to 10,000 RU/s without a shard key. For higher autoscale throughput, you need to specify the `shardKey` parameter.

Create a sharded Autoscale collection

To create a sharded collection named `'testCollection'` with a shard key called `'a.b'`, and that uses Autoscale throughput capacity set to 20,000 RU/s, use the following command:

```
use test
db.runCommand({customAction: "CreateCollection", collection: "testCollection", shardKey: "a.b",
autoScaleSettings: { maxThroughput: 20000 }});
```

Update collection

The update collection extension command updates the properties associated with the specified collection.

```
{
  customAction: "UpdateCollection",
  collection: "<Name of the collection that you want to update>",
  offerThroughput: (int) // New throughput that will be set to the collection
}
```

The following table describes the parameters within the command:

FIELD	TYPE	DESCRIPTION
`customAction`	`string`	Name of the custom command. Must be "UpdateCollection".
`collection`	`string`	Name of the collection.
`offerThroughput`	`int`	Provisioned throughput to set on the collection.
`autoScaleSettings`	`Object`	Required for Autoscale mode. This object contains the settings associated with the Autoscale capacity mode. The `maxThroughput` value describes the highest amount of Request Units that the collection will be increased to dynamically.

Output

Returns a default custom command response. See the default output of custom command for the parameters in the output.

Examples

Update the provisioned throughput associated with a collection

To update the provisioned throughput of a collection with name `"testCollection"` to 1200 RUs, use the following command:

```
use test
db.runCommand({customAction: "UpdateCollection", collection: "testCollection", offerThroughput: 1200 });
```

Get collection

The get collection custom command returns the collection object.

```
{
  customAction: "GetCollection",
  collection: "<Name of the collection>"
}
```

The following table describes the parameters within the command:

FIELD	TYPE	DESCRIPTION
`customAction`	`string`	Name of the custom command. Must be "GetCollection".
`collection`	`string`	Name of the collection.

Output

If the command succeeds, the response contains a document with the following fields

FIELD	TYPE	DESCRIPTION
`ok`	`int`	Status of response. 1 == success. 0 == failure.
`database`	`string`	Name of the database.
`collection`	`string`	Name of the collection.
`shardKeyDefinition`	`document`	Index specification document used as a shard key. This is an optional response parameter.
`provisionedThroughput`	`int`	Provisioned Throughput to set on the collection. This is an optional response parameter.
`autoScaleSettings`	`Object`	This object contains the capacity parameters associated with the database if it is using the Autoscale mode. The `maxThroughput` value describes the highest amount of Request Units that the collection will be increased to dynamically.

If the command fails, a default custom command response is returned. See the default output of custom command

for the parameters in the output.

Examples

Get the collection

To get the collection object for a collection named `"testCollection"`, use the following command:

```
use test
db.runCommand({customAction: "GetCollection", collection: "testCollection"});
```

If the collection has an associated throughput capacity to it, it will include the `provisionedThroughput` value, and the output would be:

```
{
        "database" : "test",
        "collection" : "testCollection",
        "provisionedThroughput" : 400,
        "ok" : 1
}
```

If the collection has an associated Autoscale throughput, it will include the `autoScaleSettings` object with the `maxThroughput` parameter, which defines the maximum throughput the collection will increase to dynamically. Additionally, it will also include the `provisionedThroughput` value, which defines the minimum throughput this collection will reduce to if there are no requests in the collection:

```
{
        "database" : "test",
        "collection" : "testCollection",
        "provisionedThroughput" : 1000,
        "autoScaleSettings" : {
            "maxThroughput" : 10000
        },
        "ok" : 1
}
```

If the collection is sharing database-level throughput, either on Autoscale mode or manual, the output would be:

```
{ "database" : "test", "collection" : "testCollection", "ok" : 1 }
```

```
{
        "database" : "test",
        "provisionedThroughput" : 2000,
        "autoScaleSettings" : {
            "maxThroughput" : 20000
        },
        "ok" : 1
}
```

Default output of a custom command

If not specified, a custom response contains a document with the following fields:

FIELD	TYPE	DESCRIPTION

FIELD	TYPE	DESCRIPTION
`ok`	`int`	Status of response. 1 == success. 0 == failure.
`code`	`int`	Only returned when the command failed (i.e. ok == 0). Contains the MongoDB error code. This is an optional response parameter.
`errMsg`	`string`	Only returned when the command failed (i.e. ok == 0). Contains a user-friendly error message. This is an optional response parameter.

For example:

```
{ "ok" : 1 }
```

Next steps

Next you can proceed to learn the following Azure Cosmos DB concepts:

- Indexing in Azure Cosmos DB
- Expire data in Azure Cosmos DB automatically with time to live

Manage Azure Cosmos DB MongoDB API resources using Azure Resource Manager templates

12/4/2020 • 6 minutes to read • Edit Online

APPLIES TO: Azure Cosmos DB API for MongoDB

In this article, you learn how to use Azure Resource Manager templates to help deploy and manage your Azure Cosmos DB accounts for MongoDB API, databases, and collections.

This article has examples for Azure Cosmos DB's API for MongoDB only, to find examples for other API type accounts see: use Azure Resource Manager templates with Azure Cosmos DB's API for Cassandra, Gremlin, SQL, Table articles.

IMPORTANT

- Account names are limited to 44 characters, all lowercase.
- To change the throughput values, redeploy the template with updated RU/s.
- When you add or remove locations to an Azure Cosmos account, you can't simultaneously modify other properties. These operations must be done separately.

To create any of the Azure Cosmos DB resources below, copy the following example template into a new json file. You can optionally create a parameters json file to use when deploying multiple instances of the same resource with different names and values. There are many ways to deploy Azure Resource Manager templates including, Azure portal, Azure CLI, Azure PowerShell and GitHub.

Azure Cosmos account for MongoDB with autoscale provisioned throughput

This template will create an Azure Cosmos account for MongoDB API (3.2 or 3.6) with two collections that share autoscale throughput at the database level. This template is also available for one-click deploy from Azure Quickstart Templates Gallery.

Deploy to Azure

```
{
   "$schema": "https://schema.management.azure.com/schemas/2019-04-01/deploymentTemplate.json#",
   "contentVersion": "1.0.0.0",
   "parameters": {
      "accountName": {
         "type": "string",
         "defaultValue": "[concat('mongodb-', uniqueString(resourceGroup().id))]",
         "metadata": {
            "description": "Cosmos DB account name"
         }
      },
      "location": {
         "type": "string",
         "defaultValue": "[resourceGroup().location]",
         "metadata": {
            "description": "Location for the Cosmos DB account."
         }
      },
      "primaryRegion": {
```

```
      "primaryRegion": {
         "type": "string",
         "metadata": {
            "description": "The primary replica region for the Cosmos DB account."
         }
      },
      "secondaryRegion": {
         "type": "string",
         "metadata": {
            "description": "The secondary replica region for the Cosmos DB account."
         }
      },
      "serverVersion": {
         "defaultValue": "3.6",
         "allowedValues": [
            "3.2",
            "3.6"
         ],
         "type": "String",
         "metadata": {
            "description": "Specifies the MongoDB server version to use."
         }
      },
      "defaultConsistencyLevel": {
         "type": "string",
         "defaultValue": "Session",
         "allowedValues": [
            "Eventual",
            "ConsistentPrefix",
            "Session",
            "BoundedStaleness",
            "Strong"
         ],
         "metadata": {
            "description": "The default consistency level of the Cosmos DB account."
         }
      },
      "maxStalenessPrefix": {
         "type": "int",
         "defaultValue": 100000,
         "minValue": 10,
         "maxValue": 2147483647,
         "metadata": {
            "description": "Max stale requests. Required for BoundedStaleness. Valid ranges, Single Region:
10 to 1000000. Multi Region: 100000 to 1000000."
         }
      },
      "maxIntervalInSeconds": {
         "type": "int",
         "defaultValue": 300,
         "minValue": 5,
         "maxValue": 86400,
         "metadata": {
            "description": "Max lag time (seconds). Required for BoundedStaleness. Valid ranges, Single
Region: 5 to 84600. Multi Region: 300 to 86400."
         }
      },
      "databaseName": {
         "type": "string",
         "metadata": {
            "description": "The name for the Mongo DB database"
         }
      },
      "collection1Name": {
         "type": "string",
         "metadata": {
            "description": "The name for the first Mongo DB collection"
         }
      },
```

```
      "collection2Name": {
         "type": "string",
         "metadata": {
            "description": "The name for the second Mongo DB collection"
         }
      },
      "autoscaleMaxThroughput": {
         "type": "int",
         "defaultValue": 4000,
         "minValue": 4000,
         "maxValue": 1000000,
         "metadata": {
            "description": "Maximum throughput when using Autoscale Throughput Policy for the Database"
         }
      }
   },
   "variables": {
      "accountName": "[toLower(parameters('accountName'))]",
      "consistencyPolicy": {
         "Eventual": {
            "defaultConsistencyLevel": "Eventual"
         },
         "ConsistentPrefix": {
            "defaultConsistencyLevel": "ConsistentPrefix"
         },
         "Session": {
            "defaultConsistencyLevel": "Session"
         },
         "BoundedStaleness": {
            "defaultConsistencyLevel": "BoundedStaleness",
            "maxStalenessPrefix": "[parameters('maxStalenessPrefix')]",
            "maxIntervalInSeconds": "[parameters('maxIntervalInSeconds')]"
         },
         "Strong": {
            "defaultConsistencyLevel": "Strong"
         }
      },
      "locations": [
         {
            "locationName": "[parameters('primaryRegion')]",
            "failoverPriority": 0,
            "isZoneRedundant": false
         },
         {
            "locationName": "[parameters('secondaryRegion')]",
            "failoverPriority": 1,
            "isZoneRedundant": false
         }
      ]
   },
   "resources": [
      {
         "type": "Microsoft.DocumentDB/databaseAccounts",
         "name": "[variables('accountName')]",
         "apiVersion": "2020-04-01",
         "location": "[parameters('location')]",
         "kind": "MongoDB",
         "properties": {
            "consistencyPolicy": "[variables('consistencyPolicy')[parameters('defaultConsistencyLevel')]]",
            "locations": "[variables('locations')]",
            "databaseAccountOfferType": "Standard",
            "apiProperties": {
               "serverVersion": "[parameters('serverVersion')]"
            }
         }
      },
      {
         "type": "Microsoft.DocumentDB/databaseAccounts/mongodbDatabases",
         "name": "[concat(variables('accountName'), '/', parameters('databaseName'))]",
```

```
         "apiVersion": "2020-04-01",
         "dependsOn": [
            "[resourceId('Microsoft.DocumentDB/databaseAccounts/', variables('accountName'))]"
         ],
         "properties": {
            "resource": {
               "id": "[parameters('databaseName')]"
            },
            "options": {
               "autoscaleSettings": {
                  "maxThroughput": "[parameters('autoscaleMaxThroughput')]"
               }
            }
         }
      },
      {
         "type": "Microsoft.DocumentDb/databaseAccounts/mongodbDatabases/collections",
         "name": "[concat(variables('accountName'), '/', parameters('databaseName'), '/',
parameters('collection1Name'))]",
         "apiVersion": "2020-03-01",
         "dependsOn": [
            "[resourceId('Microsoft.DocumentDB/databaseAccounts/mongodbDatabases', variables('accountName'),
parameters('databaseName'))]"
         ],
         "properties": {
            "resource": {
               "id": "[parameters('collection1Name')]",
               "shardKey": {
                  "user_id": "Hash"
               },
               "indexes": [
                  {
                     "key": {
                        "keys": [ "user_id", "user_address" ]
                     },
                     "options": {
                        "unique": true
                     }
                  },
                  {
                     "key": {
                        "keys": [ "_ts" ],
                        "options": { "expireAfterSeconds": "2629746" }
                     }
                  }
               ],
               "options": {
                  "If-Match": "<ETag>"
               }
            }
         }
      },
      {
         "type": "Microsoft.DocumentDb/databaseAccounts/mongodbDatabases/collections",
         "name": "[concat(variables('accountName'), '/', parameters('databaseName'), '/',
parameters('collection2Name'))]",
         "apiVersion": "2020-03-01",
         "dependsOn": [
            "[resourceId('Microsoft.DocumentDB/databaseAccounts/mongodbDatabases', variables('accountName'),
parameters('databaseName'))]"
         ],
         "properties": {
            "resource": {
               "id": "[parameters('collection2Name')]",
               "shardKey": {
                  "company_id": "Hash"
               },
               "indexes": [
                  {
```

```
                    "key": {
                       "keys": [
                          "company_id",
                          "company_address"
                       ]
                    },
                    "options": {
                       "unique": true
                    }
                 },
                 {
                    "key": {
                       "keys": [ "_ts" ],
                       "options": { "expireAfterSeconds": "2629746" }
                    }
                 }
              ],
              "options": {
                 "If-Match": "<ETag>"
              }
           }
        }
     }
   ]
}
```

Azure Cosmos account for MongoDB with standard provisioned throughput

This template will create an Azure Cosmos account for MongoDB API (3.2 or 3.6) with two collections that share 400 RU/s standard (manual) throughput at the database level. This template is also available for one-click deploy from Azure Quickstart Templates Gallery.

Deploy to Azure

```
{
    "$schema": "https://schema.management.azure.com/schemas/2019-04-01/deploymentTemplate.json#",
    "contentVersion": "1.0.0.0",
    "parameters": {
        "accountName": {
            "type": "string",
            "defaultValue": "[concat('mongodb-', uniqueString(resourceGroup().id))]",
            "metadata": {
                "description": "Cosmos DB account name"
            }
        },
        "location": {
            "type": "string",
            "defaultValue": "[resourceGroup().location]",
            "metadata": {
                "description": "Location for the Cosmos DB account."
            }
        },
        "primaryRegion":{
            "type":"string",
            "metadata": {
                "description": "The primary replica region for the Cosmos DB account."
            }
        },
        "secondaryRegion":{
            "type":"string",
            "metadata": {
              "description": "The secondary replica region for the Cosmos DB account."
          }
```

```
        },
        "defaultConsistencyLevel": {
            "type": "string",
            "defaultValue": "Session",
            "allowedValues": [ "Eventual", "ConsistentPrefix", "Session", "BoundedStaleness", "Strong" ],
            "metadata": {
                "description": "The default consistency level of the Cosmos DB account."
            }
        },
        "serverVersion": {
            "defaultValue": "3.6",
            "allowedValues": [
                "3.2",
                "3.6"
            ],
            "type": "String",
            "metadata": {
                "description": "Specifies the MongoDB server version to use."
            }
        },
        "maxStalenessPrefix": {
            "type": "int",
            "defaultValue": 100000,
            "minValue": 10,
            "maxValue": 2147483647,
            "metadata": {
                "description": "Max stale requests. Required for BoundedStaleness. Valid ranges, Single
Region: 10 to 1000000. Multi Region: 100000 to 1000000."
            }
        },
        "maxIntervalInSeconds": {
            "type": "int",
            "defaultValue": 300,
            "minValue": 5,
            "maxValue": 86400,
            "metadata": {
                "description": "Max lag time (seconds). Required for BoundedStaleness. Valid ranges, Single
Region: 5 to 84600. Multi Region: 300 to 86400."
            }
        },
        "databaseName": {
            "type": "string",
            "metadata": {
                "description": "The name for the Mongo DB database"
            }
        },
        "throughput": {
            "type": "int",
            "defaultValue": 400,
            "minValue": 400,
            "maxValue": 1000000,
            "metadata": {
                "description": "The shared throughput for the Mongo DB database"
            }
        },
        "collection1Name": {
            "type": "string",
            "metadata": {
                "description": "The name for the first Mongo DB collection"
            }
        },
        "collection2Name": {
            "type": "string",
            "metadata": {
                "description": "The name for the second Mongo DB collection"
            }
        }
    },
    "variables": {
```

```
        "accountName": "[toLower(parameters('accountName'))]",
        "consistencyPolicy": {
            "Eventual": {
                "defaultConsistencyLevel": "Eventual"
            },
            "ConsistentPrefix": {
                "defaultConsistencyLevel": "ConsistentPrefix"
            },
            "Session": {
                "defaultConsistencyLevel": "Session"
            },
            "BoundedStaleness": {
                "defaultConsistencyLevel": "BoundedStaleness",
                "maxStalenessPrefix": "[parameters('maxStalenessPrefix')]",
                "maxIntervalInSeconds": "[parameters('maxIntervalInSeconds')]"
            },
            "Strong": {
                "defaultConsistencyLevel": "Strong"
            }
        },
        "locations":
        [
            {
                "locationName": "[parameters('primaryRegion')]",
                "failoverPriority": 0,
                "isZoneRedundant": false
            },
            {
                "locationName": "[parameters('secondaryRegion')]",
                "failoverPriority": 1,
                "isZoneRedundant": false
            }
        ]
    },
    "resources":
    [
        {
            "type": "Microsoft.DocumentDB/databaseAccounts",
            "name": "[variables('accountName')]",
            "apiVersion": "2020-03-01",
            "location": "[parameters('location')]",
            "kind": "MongoDB",
            "properties": {
                "consistencyPolicy": "[variables('consistencyPolicy')
[parameters('defaultConsistencyLevel')]]",
                "locations": "[variables('locations')]",
                "databaseAccountOfferType": "Standard",
                "apiProperties": {
                    "serverVersion": "[parameters('serverVersion')]"
                }
            }
        },
        {
            "type": "Microsoft.DocumentDB/databaseAccounts/mongodbDatabases",
            "name": "[concat(variables('accountName'), '/', parameters('databaseName'))]",
            "apiVersion": "2020-03-01",
            "dependsOn": [ "[resourceId('Microsoft.DocumentDB/databaseAccounts/', variables('accountName'))]"
],
            "properties":{
                "resource":{
                    "id": "[parameters('databaseName')]"
                },
                "options": { "throughput": "[parameters('throughput')]" }
            }
        },
        {
            "type": "Microsoft.DocumentDb/databaseAccounts/mongodbDatabases/collections",
            "name": "[concat(variables('accountName'), '/', parameters('databaseName'), '/',
parameters('collection1Name'))]"
```

```
parameters('collection1Name'))]",
            "apiVersion": "2020-03-01",
            "dependsOn": [ "[resourceId('Microsoft.DocumentDB/databaseAccounts/mongodbDatabases',
variables('accountName'), parameters('databaseName'))]" ],
            "properties":
            {
                "resource":{
                    "id":  "[parameters('collection1Name')]",
                    "shardKey": { "user_id": "Hash" },
                    "indexes": [
                        {
                            "key": { "keys":["user_id", "user_address"] },
                            "options": { "unique": "true" }
                        },
                        {
                            "key": { "keys":["_ts"] },
                            "options": { "expireAfterSeconds": "2629746" }
                        }
                    ],
                    "options": {
                        "If-Match": "<ETag>"
                    }
                }
            }
        },
        {
            "type": "Microsoft.DocumentDb/databaseAccounts/mongodbDatabases/collections",
            "name": "[concat(variables('accountName'), '/', parameters('databaseName'), '/',
parameters('collection2Name'))]",
            "apiVersion": "2020-03-01",
            "dependsOn": [ "[resourceId('Microsoft.DocumentDB/databaseAccounts/mongodbDatabases',
variables('accountName'),  parameters('databaseName'))]" ],
            "properties":
            {
                "resource":{
                    "id":  "[parameters('collection2Name')]",
                    "shardKey": { "company_id": "Hash" },
                    "indexes": [
                        {
                            "key": { "keys":["company_id", "company_address"] },
                            "options": { "unique": "true" }
                        },
                        {
                            "key": { "keys":["_ts"] },
                            "options": { "expireAfterSeconds": "2629746" }
                        }
                    ],
                    "options": {
                        "If-Match": "<ETag>"
                    }
                }
            }
        }
    ]
}
```

Next steps

Here are some additional resources:

- Azure Resource Manager documentation
- Azure Cosmos DB resource provider schema
- Azure Cosmos DB Quickstart templates
- Troubleshoot common Azure Resource Manager deployment errors

Azure PowerShell samples for Azure Cosmos DB API for MongoDB

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: Azure Cosmos DB API for MongoDB

The following table includes links to commonly used Azure PowerShell scripts for Azure Cosmos DB. Use the links on the right to navigate to API specific samples. Common samples are the same across all APIs. Reference pages for all Azure Cosmos DB PowerShell cmdlets are available in the Azure PowerShell Reference. Please check for updates to `Az.CosmosDB` regularly. You can also fork these PowerShell samples for Cosmos DB from our GitHub repository, Cosmos DB PowerShell Samples on GitHub.

Common Samples

TASK	DESCRIPTION
Update an account	Update a Cosmos DB account's default consistency level.
Update an account's regions	Update a Cosmos DB account's regions.
Change failover priority or trigger failover	Change the regional failover priority of an Azure Cosmos account or trigger a manual failover.
Account keys or connection strings	Get primary and secondary keys, connection strings or regenerate an account key of an Azure Cosmos DB account.
Create a Cosmos Account with IP Firewall	Create an Azure Cosmos DB account with IP Firewall enabled.

Mongo DB API Samples

TASK	DESCRIPTION
Create an account, database and collection	Creates an Azure Cosmos account, database and collection.
Create an account, database and collection with autoscale	Creates an Azure Cosmos account, database and collection with autoscale.
List or get databases or collections	List or get database or collection.
Throughput operations	Throughput operations for a database or collection including get, update and migrate between autoscale and standard throughput.
Lock resources from deletion	Prevent resources from being deleted with resource locks.

Azure CLI samples for Azure Cosmos DB API for MongoDB

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: Azure Cosmos DB API for MongoDB

The following table includes links to sample Azure CLI scripts for Azure Cosmos DB. Use the links on the right to navigate to API specific samples. Common samples are the same across all APIs. Reference pages for all Azure Cosmos DB CLI commands are available in the Azure CLI Reference. Azure Cosmos DB CLI script samples can also be found in the Azure Cosmos DB CLI GitHub Repository.

These samples require Azure CLI version 2.12.1 or later. Run `az --version` to find the version. If you need to install or upgrade, see Install Azure CLI

Common Samples

These samples apply to all Azure Cosmos DB APIs

TASK	DESCRIPTION
Add or failover regions	Add a region, change failover priority, trigger a manual failover.
Account keys and connection strings	List account keys, read-only keys, regenerate keys and list connection strings.
Secure with IP firewall	Create a Cosmos account with IP firewall configured.
Secure new account with service endpoints	Create a Cosmos account and secure with service-endpoints.
Secure existing account with service endpoints	Update a Cosmos account to secure with service-endpoints when the subnet is eventually configured.

MongoDB API Samples

TASK	DESCRIPTION
Create an Azure Cosmos account, database and collection	Creates an Azure Cosmos DB account, database, and collection for MongoDB API.
Create an Azure Cosmos account, database with autoscale and two collections with shared throughput	Creates an Azure Cosmos DB account, database with autoscale and two collections with shared throughput for MongoDB API.
Throughput operations	Read, update and migrate between autoscale and standard throughput on a database and collection.
Lock resources from deletion	Prevent resources from being deleted with resource locks.

TASK	DESCRIPTION

Troubleshoot common issues in Azure Cosmos DB's API for MongoDB

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: Azure Cosmos DB API for MongoDB

The following article describes common errors and solutions for databases using the Azure Cosmos DB API for MongoDB.

> **NOTE**
> Azure Cosmos DB does not host the MongoDB engine. It provides an implementation of the MongoDB wire protocol version 3.6 and legacy support for wire protocol version 3.2, therefore some of these errors are only found in Azure Cosmos DB's API for MongoDB.

Common errors and solutions

ERROR	CODE	DESCRIPTION	SOLUTION
ExceededTimeLimit	50	The request has exceeded the timeout of 60 seconds of execution.	There can be many causes for this error. One of the causes is when the current allocated request units capacity is not sufficient to complete the request. This can be solved by increasing the request units of that collection or database. In other cases, this error can be worked-around by splitting a large request into smaller ones.
TooManyRequests	16500	The total number of request units consumed is more than the provisioned request-unit rate for the collection and has been throttled.	Consider scaling the throughput assigned to a container or a set of containers from the Azure portal or you can retry the operation.
ExceededMemoryLimit	16501	As a multi-tenant service, the operation has gone over the client's memory allotment.	Reduce the scope of the operation through more restrictive query criteria or contact support from the Azure portal. Example: `db.getCollection('users').aggregate({name: "Andy"}}, {$sort: {age: -1}}]`
The index path corresponding to the specified order-by item is excluded / The order by query does not have a corresponding composite index that it can be served from.	2	The query requests a sort on a field that is not indexed.	Create a matching index (or composite index) for the sort query being attempted.
MongoDB wire version issues	-	The older versions of MongoDB drivers are unable to detect the Azure Cosmos account's name in the connection strings.	Append *appName=@**accountName**@* at the end of your Cosmos DB's API for MongoDB connection string, where ***accountName*** is your Cosmos DB account name.

Next steps

- Learn how to use Studio 3T with Azure Cosmos DB's API for MongoDB.
- Learn how to use Robo 3T with Azure Cosmos DB's API for MongoDB.
- Explore MongoDB samples with Azure Cosmos DB's API for MongoDB.

Introduction to Gremlin API in Azure Cosmos DB

12/4/2020 • 7 minutes to read • Edit Online

APPLIES TO: Gremlin API

Azure Cosmos DBis the globally distributed, multi-model database service from Microsoft for mission-critical applications. It is a multi-model database and supports document, key-value, graph, and column-family data models. "Azure Cosmos DB provides a graph database service via the Gremlin API on a fully managed database service designed for any scale.

This article provides an overview of the Azure Cosmos DB Gremlin API and explains how to use them to store massive graphs with billions of vertices and edges. You can query the graphs with millisecond latency and evolve the graph structure easily. Azure Cosmos DB's Gremlin API is built based on the Apache TinkerPop, a graph computing framework. The Gremlin API in Azure Cosmos DB uses the Gremlin query language.

Azure Cosmos DB's Gremlin API combines the power of graph database algorithms with highly scalable, managed infrastructure to provide a unique, flexible solution to most common data problems associated with lack of flexibility and relational approaches.

> **NOTE**
>
> The serverless capacity mode is now available on Azure Cosmos DB's Gremlin API.

Features of Azure Cosmos DB's Gremlin API

Azure Cosmos DB is a fully managed graph database that offers global distribution, elastic scaling of storage and throughput, automatic indexing and query, tunable consistency levels, and support for the TinkerPop standard.

The following are the differentiated features that Azure Cosmos DB Gremlin API offers:

- **Elastically scalable throughput and storage**

Graphs in the real world need to scale beyond the capacity of a single server. Azure Cosmos DB supports horizontally scalable graph databases that can have a virtually unlimited size in terms of storage and provisioned throughput. As the graph database scale grows, the data will be automatically distributed using graph partitioning.

- **Multi-region replication**

  Azure Cosmos DB can automatically replicate your graph data to any Azure region worldwide. Global replication simplifies the development of applications that require global access to data. In addition to minimizing read and write latency anywhere around the world, Azure Cosmos DB provides automatic regional failover mechanism that can ensure the continuity of your application in the rare case of a service interruption in a region.

- **Fast queries and traversals with the most widely adopted graph query standard**

  Store heterogeneous vertices and edges and query them through a familiar Gremlin syntax. Gremlin is an imperative, functional query language that provides a rich interface to implement common graph algorithms.

  Azure Cosmos DB enables rich real-time queries and traversals without the need to specify schema hints, secondary indexes, or views. Learn more in Query graphs by using Gremlin.

- **Fully managed graph database**

  Azure Cosmos DB eliminates the need to manage database and machine resources. Most existing graph database platforms are bound to the limitations of their infrastructure and often require a high degree of maintenance to ensure its operation.

  As a fully managed service, Cosmos DB removes the need to manage virtual machines, update runtime software, manage sharding or replication, or deal with complex data-tier upgrades. Every graph is automatically backed up and protected against regional failures. This allows developers to focus on delivering application value instead of operating and managing their graph databases.

- **Automatic indexing**

  By default, Azure Cosmos DB automatically indexes all the properties within nodes (also called as vertices) and edges in the graph and doesn't expect or require any schema or creation of secondary indices. Learn more about indexing in Azure Cosmos DB.

- **Compatibility with Apache TinkerPop**

  Azure Cosmos DB supports the open-source Apache TinkerPop standard. The Tinkerpop standard has an ample ecosystem of applications and libraries that can be easily integrated with Azure Cosmos DB's Gremlin API.

- **Tunable consistency levels**

  Azure Cosmos DB provides five well-defined consistency levels to achieve the right tradeoff between consistency and performance for your application. For queries and read operations, Azure Cosmos DB offers five distinct consistency levels: strong, bounded-staleness, session, consistent prefix, and eventual. These granular, well-defined consistency levels allow you to make sound tradeoffs among consistency, availability, and latency. Learn more in Tunable data consistency levels in Azure Cosmos DB.

Scenarios that use Gremlin API

Here are some scenarios where graph support of Azure Cosmos DB can be useful:

- **Social networks/Customer 365**

By combining data about your customers and their interactions with other people, you can develop personalized experiences, predict customer behavior, or connect people with others with similar interests. Azure Cosmos DB can be used to manage social networks and track customer preferences and data.

- **Recommendation engines**

  This scenario is commonly used in the retail industry. By combining information about products, users, and user interactions, like purchasing, browsing, or rating an item, you can build customized recommendations. The low latency, elastic scale, and native graph support of Azure Cosmos DB is ideal for these scenarios.

- **Geospatial**

  Many applications in telecommunications, logistics, and travel planning need to find a location of interest within an area or locate the shortest/optimal route between two locations. Azure Cosmos DB is a natural fit for these problems.

- **Internet of Things**

  With the network and connections between IoT devices modeled as a graph, you can build a better understanding of the state of your devices and assets. You also can learn how changes in one part of the network can potentially affect another part.

Introduction to graph databases

Data as it appears in the real world is naturally connected. Traditional data modeling focuses on defining entities separately and computing their relationships at runtime. While this model has its advantages, highly connected data can be challenging to manage under its constraints.

A graph database approach relies on persisting relationships in the storage layer instead, which leads to highly efficient graph retrieval operations. Azure Cosmos DB's Gremlin API supports the property graph model.

Property graph objects

A property graph is a structure that's composed of vertices and edges. Both objects can have an arbitrary number of key-value pairs as properties.

- **Vertices/nodes** - Vertices denote discrete entities, such as a person, a place, or an event.
- **Edges/relationships** - Edges denote relationships between vertices. For example, a person might know another person, be involved in an event, and recently been at a location.
- **Properties** - Properties express information about the vertices and edges. There can be any number of properties in either vertices or edges, and they can be used to describe and filter the objects in a query. Example properties include a vertex that has name and age, or an edge, which can have a time stamp and/or a weight.
- **Label** - A label is a name or the identifier of a vertex or an edge. Labels can group multiple vertices or edges such that all the vertices/edges in a group have a certain label. For example, a graph can have multiple vertices of label type "person".

Graph databases are often included within the NoSQL or non-relational database category, since there is no dependency on a schema or constrained data model. This lack of schema allows for modeling and storing connected structures naturally and efficiently.

Graph database by example

Let's use a sample graph to understand how queries can be expressed in Gremlin. The following figure shows a business application that manages data about users, interests, and devices in the form of a graph.

This graph has the following *vertex* types (these are also called "label" in Gremlin):

- **People**: The graph has three people, Robin, Thomas, and Ben
- **Interests**: Their interests, in this example, the game of Football
- **Devices**: The devices that people use
- **Operating Systems**: The operating systems that the devices run on
- **Place**: The places from which the devices are accessed

We represent the relationships between these entities via the following *edge* types:

- **Knows**: For example, "Thomas knows Robin"
- **Interested**: To represent the interests of the people in our graph, for example, "Ben is interested in Football"
- **RunsOS**: Laptop runs the Windows OS
- **Uses**: To represent which device a person uses. For example, Robin uses a Motorola phone with serial number 77
- **Located**: To represent the location from which the devices are accessed

The Gremlin Console is an interactive terminal offered by the Apache TinkerPop and this terminal is used to interact with the graph data. To learn more, see the quickstart doc on how to use the Gremlin console. You can also perform these operations using Gremlin drivers in the platform of your choice (Java, Node.js, Python, or .NET). The following examples show how to run queries against this graph data using the Gremlin Console.

First let's look at CRUD. The following Gremlin statement inserts the "Thomas" vertex into the graph:

```
:> g.addV('person').property('id', 'thomas.1').property('firstName', 'Thomas').property('lastName',
'Andersen').property('age', 44)
```

Next, the following Gremlin statement inserts a "knows" edge between Thomas and Robin.

```
:> g.V('thomas.1').addE('knows').to(g.V('robin.1'))
```

The following query returns the "person" vertices in descending order of their first names:

```
:> g.V().hasLabel('person').order().by('firstName', decr)
```

Where graphs shine is when you need to answer questions like "What operating systems do friends of Thomas use?". You can run this Gremlin traversal to get that information from the graph:

```
:> g.V('thomas.1').out('knows').out('uses').out('runsos').group().by('name').by(count())
```

Next steps

To learn more about graph support in Azure Cosmos DB, see:

- Get started with the Azure Cosmos DB graph tutorial.
- Learn about how to query graphs in Azure Cosmos DB by using Gremlin.

Azure Cosmos DB Gremlin graph support and compatibility with TinkerPop features

12/4/2020 • 9 minutes to read • Edit Online

APPLIES TO: Gremlin API

Azure Cosmos DB supports Apache Tinkerpop's graph traversal language, known as Gremlin. You can use the Gremlin language to create graph entities (vertices and edges), modify properties within those entities, perform queries and traversals, and delete entities.

Azure Cosmos DB Graph engine closely follows Apache TinkerPop traversal steps specification but there are differences in the implementation that are specific for Azure Cosmos DB. In this article, we provide a quick walkthrough of Gremlin and enumerate the Gremlin features that are supported by the Gremlin API.

Compatible client libraries

The following table shows popular Gremlin drivers that you can use against Azure Cosmos DB:

DOWNLOAD	SOURCE	GETTING STARTED	SUPPORTED CONNECTOR VERSION
.NET	Gremlin.NET on GitHub	Create Graph using .NET	3.4.6
Java	Gremlin JavaDoc	Create Graph using Java	3.2.0+
Node.js	Gremlin-JavaScript on GitHub	Create Graph using Node.js	3.3.4+
Python	Gremlin-Python on GitHub	Create Graph using Python	3.2.7
PHP	Gremlin-PHP on GitHub	Create Graph using PHP	3.1.0
Go Lang	Go Lang		This library is built by external contributors. The Azure Cosmos DB team doesn't offer any support or maintain the library.
Gremlin console	TinkerPop docs	Create Graph using Gremlin Console	3.2.0 +

Supported Graph Objects

TinkerPop is a standard that covers a wide range of graph technologies. Therefore, it has standard terminology to describe what features are provided by a graph provider. Azure Cosmos DB provides a persistent, high concurrency, writeable graph database that can be partitioned across multiple servers or clusters.

The following table lists the TinkerPop features that are implemented by Azure Cosmos DB:

CATEGORY	AZURE COSMOS DB IMPLEMENTATION	NOTES
Graph features	Provides Persistence and ConcurrentAccess. Designed to support Transactions	Computer methods can be implemented via the Spark connector.
Variable features	Supports Boolean, Integer, Byte, Double, Float, Integer, Long, String	Supports primitive types, is compatible with complex types via data model
Vertex features	Supports RemoveVertices, MetaProperties, AddVertices, MultiProperties, StringIds, UserSuppliedIds, AddProperty, RemoveProperty	Supports creating, modifying, and deleting vertices
Vertex property features	StringIds, UserSuppliedIds, AddProperty, RemoveProperty, BooleanValues, ByteValues, DoubleValues, FloatValues, IntegerValues, LongValues, StringValues	Supports creating, modifying, and deleting vertex properties
Edge features	AddEdges, RemoveEdges, StringIds, UserSuppliedIds, AddProperty, RemoveProperty	Supports creating, modifying, and deleting edges
Edge property features	Properties, BooleanValues, ByteValues, DoubleValues, FloatValues, IntegerValues, LongValues, StringValues	Supports creating, modifying, and deleting edge properties

Gremlin wire format

Azure Cosmos DB uses the JSON format when returning results from Gremlin operations. Azure Cosmos DB currently supports the JSON format. For example, the following snippet shows a JSON representation of a vertex *returned to the client* from Azure Cosmos DB:

```
{
  "id": "a7111ba7-0ea1-43c9-b6b2-efc5e3aea4c0",
  "label": "person",
  "type": "vertex",
  "outE": {
    "knows": [
      {
        "id": "3ee53a60-c561-4c5e-9a9f-9c7924bc9aef",
        "inV": "04779300-1c8e-489d-9493-50fd1325a658"
      },
      {
        "id": "21984248-ee9e-43a8-a7f6-30642bc14609",
        "inV": "a8e3e741-2ef7-4c01-b7c8-199f8e43e3bc"
      }
    ]
  },
  "properties": {
    "firstName": [
      {
        "value": "Thomas"
      }
    ],
    "lastName": [
      {
        "value": "Andersen"
      }
    ],
    "age": [
      {
        "value": 45
      }
    ]
  }
}
```

The properties used by the JSON format for vertices are described below:

PROPERTY	DESCRIPTION
`id`	The ID for the vertex. Must be unique (in combination with the value of `_partition` if applicable). If no value is provided, it will be automatically supplied with a GUID
`label`	The label of the vertex. This property is used to describe the entity type.
`type`	Used to distinguish vertices from non-graph documents
`properties`	Bag of user-defined properties associated with the vertex. Each property can have multiple values.
`_partition`	The partition key of the vertex. Used for graph partitioning.
`outE`	This property contains a list of out edges from a vertex. Storing the adjacency information with vertex allows for fast execution of traversals. Edges are grouped based on their labels.

And the edge contains the following information to help with navigation to other parts of the graph.

PROPERTY	DESCRIPTION
`id`	The ID for the edge. Must be unique (in combination with the value of `_partition` if applicable)
`label`	The label of the edge. This property is optional, and used to describe the relationship type.
`inV`	This property contains a list of in vertices for an edge. Storing the adjacency information with the edge allows for fast execution of traversals. Vertices are grouped based on their labels.
`properties`	Bag of user-defined properties associated with the edge. Each property can have multiple values.

Each property can store multiple values within an array.

PROPERTY	DESCRIPTION
`value`	The value of the property

Gremlin steps

Now let's look at the Gremlin steps supported by Azure Cosmos DB. For a complete reference on Gremlin, see TinkerPop reference.

STEP	DESCRIPTION	TINKERPOP 3.2 DOCUMENTATION
`addE`	Adds an edge between two vertices	addE step
`addV`	Adds a vertex to the graph	addV step
`and`	Ensures that all the traversals return a value	and step
`as`	A step modulator to assign a variable to the output of a step	as step
`by`	A step modulator used with `group` and `order`	by step
`coalesce`	Returns the first traversal that returns a result	coalesce step
`constant`	Returns a constant value. Used with `coalesce`	constant step
`count`	Returns the count from the traversal	count step
`dedup`	Returns the values with the duplicates removed	dedup step

STEP	DESCRIPTION	TINKERPOP 3.2 DOCUMENTATION
`drop`	Drops the values (vertex/edge)	drop step
`executionProfile`	Creates a description of all operations generated by the executed Gremlin step	executionProfile step
`fold`	Acts as a barrier that computes the aggregate of results	fold step
`group`	Groups the values based on the labels specified	group step
`has`	Used to filter properties, vertices, and edges. Supports `hasLabel`, `hasId`, `hasNot`, and `has` variants.	has step
`inject`	Inject values into a stream	inject step
`is`	Used to perform a filter using a boolean expression	is step
`limit`	Used to limit number of items in the traversal	limit step
`local`	Local wraps a section of a traversal, similar to a subquery	local step
`not`	Used to produce the negation of a filter	not step
`optional`	Returns the result of the specified traversal if it yields a result else it returns the calling element	optional step
`or`	Ensures at least one of the traversals returns a value	or step
`order`	Returns results in the specified sort order	order step
`path`	Returns the full path of the traversal	path step
`project`	Projects the properties as a Map	project step
`properties`	Returns the properties for the specified labels	properties step
`range`	Filters to the specified range of values	range step
`repeat`	Repeats the step for the specified number of times. Used for looping	repeat step

STEP	DESCRIPTION	TINKERPOP 3.2 DOCUMENTATION
`sample`	Used to sample results from the traversal	sample step
`select`	Used to project results from the traversal	select step
`store`	Used for non-blocking aggregates from the traversal	store step
`TextP.startingWith(string)`	String filtering function. This function is used as a predicate for the `has()` step to match a property with the beginning of a given string	TextP predicates
`TextP.endingWith(string)`	String filtering function. This function is used as a predicate for the `has()` step to match a property with the ending of a given string	TextP predicates
`TextP.containing(string)`	String filtering function. This function is used as a predicate for the `has()` step to match a property with the contents of a given string	TextP predicates
`TextP.notStartingWith(string)`	String filtering function. This function is used as a predicate for the `has()` step to match a property that doesn't start with a given string	TextP predicates
`TextP.notEndingWith(string)`	String filtering function. This function is used as a predicate for the `has()` step to match a property that doesn't end with a given string	TextP predicates
`TextP.notContaining(string)`	String filtering function. This function is used as a predicate for the `has()` step to match a property that doesn't contain a given string	TextP predicates
`tree`	Aggregate paths from a vertex into a tree	tree step
`unfold`	Unroll an iterator as a step	unfold step
`union`	Merge results from multiple traversals	union step
`V`	Includes the steps necessary for traversals between vertices and edges `V`, `E`, `out`, `in`, `both`, `outE`, `inE`, `bothE`, `outV`, `inV`, `bothV`, and `otherV` for	vertex steps

STEP	DESCRIPTION	TINKERPOP 3.2 DOCUMENTATION
`where`	Used to filter results from the traversal. Supports `eq`, `neq`, `lt`, `lte`, `gt`, `gte`, and `between` operators	where step

The write-optimized engine provided by Azure Cosmos DB supports automatic indexing of all properties within vertices and edges by default. Therefore, queries with filters, range queries, sorting, or aggregates on any property are processed from the index, and served efficiently. For more information on how indexing works in Azure Cosmos DB, see our paper on schema-agnostic indexing.

Behavior differences

- Azure Cosmos DB Graph engine runs ***breadth-first*** traversal while TinkerPop Gremlin is depth-first. This behavior achieves better performance in horizontally scalable system like Cosmos DB.

Unsupported features

- ***Gremlin Bytecode*** is a programming language agnostic specification for graph traversals. Cosmos DB Graph doesn't support it yet. Use `GremlinClient.SubmitAsync()` and pass traversal as a text string.
- *`property(set, 'xyz', 1)`* set cardinality isn't supported today. Use `property(list, 'xyz', 1)` instead. To learn more, see Vertex properties with TinkerPop.
- The *`match()`* ***step*** isn't currently available. This step provides declarative querying capabilities.
- ***Objects as properties*** on vertices or edges aren't supported. Properties can only be primitive types or arrays.
- ***Sorting by array properties*** `order().by(<array property>)` isn't supported. Sorting is supported only by primitive types.
- ***Non-primitive JSON types*** aren't supported. Use `string`, `number`, or `true`/`false` types. `null` values aren't supported.
- ***GraphSONv3*** serializer isn't currently supported. Use `GraphSONv2` Serializer, Reader, and Writer classes in the connection configuration. The results returned by the Azure Cosmos DB Gremlin API don't have the same format as the GraphSON format.
- **Lambda expressions and functions** aren't currently supported. This includes the `.map{<expression>}`, the `.by{<expression>}`, and the `.filter{<expression>}` functions. To learn more, and to learn how to rewrite them using Gremlin steps, see A Note on Lambdas.
- ***Transactions*** aren't supported because of distributed nature of the system. Configure appropriate consistency model on Gremlin account to "read your own writes" and use optimistic concurrency to resolve conflicting writes.

Known limitations

- **Index utilization for Gremlin queries with mid-traversal `.V()` steps**: Currently, only the first `.V()` call of a traversal will make use of the index to resolve any filters or predicates attached to it. Subsequent calls will not consult the index, which might increase the latency and cost of the query.

  Assuming default indexing, a typical read Gremlin query that starts with the `.V()` step would use parameters in its attached filtering steps, such as `.has()` or `.where()` to optimize the cost and performance of the query. For example:

```
g.V().has('category', 'A')
```

However, when more than one `.V()` step is included in the Gremlin query, the resolution of the data for the query might not be optimal. Take the following query as an example:

```
g.V().has('category', 'A').as('a').V().has('category', 'B').as('b').select('a', 'b')
```

This query will return two groups of vertices based on their property called `category`. In this case, only the first call, `g.V().has('category', 'A')` will make use of the index to resolve the vertices based on the values of their properties.

A workaround for this query is to use subtraversal steps such as `.map()` and `union()`. This is exemplified below:

```
// Query workaround using .map()
g.V().has('category', 'A').as('a').map(__.V().has('category', 'B')).as('b').select('a','b')

// Query workaround using .union()
g.V().has('category', 'A').fold().union(unfold(), __.V().has('category', 'B'))
```

You can review the performance of the queries by using the Gremlin `executionProfile()` step.

Next steps

- Get started building a graph application using our SDKs
- Learn more about graph support in Azure Cosmos DB

Quickstart: Create, query, and traverse an Azure Cosmos DB graph database using the Gremlin console

12/4/2020 • 8 minutes to read • Edit Online

APPLIES TO: Gremlin API

Azure Cosmos DB is Microsoft's globally distributed multi-model database service. You can quickly create and query document, key/value, and graph databases, all of which benefit from the global distribution and horizontal scale capabilities at the core of Azure Cosmos DB.

This quickstart demonstrates how to create an Azure Cosmos DB Gremlin API account, database, and graph (container) using the Azure portal and then use the Gremlin Console from Apache TinkerPop to work with Gremlin API data. In this tutorial, you create and query vertices and edges, updating a vertex property, query vertices, traverse the graph, and drop a vertex.

The Gremlin console is Groovy/Java based and runs on Linux, Mac, and Windows. You can download it from the Apache TinkerPop site.

Prerequisites

You need to have an Azure subscription to create an Azure Cosmos DB account for this quickstart.

If you don't have an Azure subscription, create a free account before you begin.

You also need to install the Gremlin Console. The **recommended version is v3.4.3** or earlier. (To use Gremlin Console on Windows, you need to install Java Runtime).

Create a database account

1. In a new browser window, sign in to the Azure portal.

2. In the left menu, select **Create a resource**.

3. On the **New** page, select **Databases** > **Azure Cosmos DB**.

> **TIP**
>
> If you don't see Azure Cosmos DB in the list, just type it in the search box at the top of the page, and press *Enter* key.

4. On the **Create Azure Cosmos DB Account** page, enter the settings for the new Azure Cosmos DB account.

SETTING	VALUE	DESCRIPTION
Subscription	Your subscription	Select the Azure subscription that you want to use for this Azure Cosmos DB account.

SETTING	VALUE	DESCRIPTION
Resource Group	Create new Then enter the same name as Account Name	Select **Create new**. Then enter a new resource group name for your account. For simplicity, use the same name as your Azure Cosmos DB account name.
Account Name	Enter a unique name	Enter a unique name to identify your Azure Cosmos DB account. Your account URI will be *gremlin.azure.com* appended to your unique account name. The account name can use only lowercase letters, numbers, and hyphens (-), and must be between 3 and 31 characters long.
API	Gremlin (graph)	The API determines the type of account to create. Azure Cosmos DB provides five APIs: Core (SQL) for document databases, Gremlin for graph databases, MongoDB for document databases, Azure Table, and Cassandra. You must create a separate account for each API. Select **Gremlin (graph)**, because in this quickstart you are creating a table that works with the Gremlin API. Learn more about the Gremlin API.
Location	Select the region closest to your users	Select a geographic location to host your Azure Cosmos DB account. Use the location that's closest to your users to give them the fastest access to the data.
Capacity mode	Provisioned throughput or Serverless	Select **Provisioned throughput** to create an account in provisioned throughput mode. Select **Serverless** to create an account in serverless mode.

Select **Review+Create**. You can skip the **Network** and **Tags** section.

5. The account creation takes a few minutes. Wait for the portal to display the **Congratulations! Your Azure Cosmos DB account was created** page.

Add a graph

You can now use the Data Explorer tool in the Azure portal to create a graph database.

1. Select **Data Explorer** > **New Graph**.

   The **Add Graph** area is displayed on the far right, you may need to scroll right to see it.

2. In the **Add graph** page, enter the settings for the new graph.

SETTING	SUGGESTED VALUE	DESCRIPTION
Database ID	sample-database	Enter *sample-database* as the name for the new database. Database names must be between 1 and 255 characters, and cannot contain `/ \ # ?` or a trailing space.
Throughput	400 RUs	Change the throughput to 400 request units per second (RU/s). If you want to reduce latency, you can scale up the throughput later.
Graph ID	sample-graph	Enter *sample-graph* as the name for your new collection. Graph names have the same character requirements as database IDs.
Partition Key	/pk	All Cosmos DB accounts need a partition key to horizontally scale. Learn how to select an appropriate partition key in the Graph Data Partitioning article.

3. Once the form is filled out, select **OK**.

Connect to your app service/Graph

1. Before starting the Gremlin Console, create or modify the remote-secure.yaml configuration file in the `apache-tinkerpop-gremlin-console-3.2.5/conf` directory.

2. Fill in your *host*, *port*, *username*, *password*, *connectionPool*, and *serializer* configurations as defined in the following table:

SETTING	SUGGESTED VALUE	DESCRIPTION
hosts	[*account-name*.**gremlin**.cosmos.azure.com]	See the following screenshot. This is the **Gremlin URI** value on the Overview page of the Azure portal, in square brackets, with the trailing :443/ removed. Note: Be sure to use the Gremlin value, and **not** the URI that ends with [*account-name*.documents.azure.com] which would likely result in a "Host did not respond in a timely fashion" exception when attempting to execute Gremlin queries later.
port	443	Set to 443.
username	*Your username*	The resource of the form `/dbs/<db>/colls/<coll>` where `<db>` is your database name and `<coll>` is your collection name.
password	*Your primary key*	See second screenshot below. This is your primary key, which you can retrieve from the Keys page of the Azure portal, in the Primary Key box. Use the copy button on the left side of the box to copy the value.
connectionPool	{enableSsl: true}	Your connection pool setting for TLS.
serializer	{ className: org.apache.tinkerpop.gremlin.driver.ser.GraphSONMessageSerializerV2d0, config: { serializeResultToString: true }}	Set to this value and delete any `\n` line breaks when pasting in the value.

For the hosts value, copy the **Gremlin URI** value from the **Overview** page:

For the password value, copy the **Primary key** from the **Keys** page:

Your remote-secure.yaml file should look like this:

```
hosts: [your_database_server.gremlin.cosmos.azure.com]
port: 443
username: /dbs/your_database_account/colls/your_collection
password: your_primary_key
connectionPool: {
  enableSsl: true
}
serializer: { className: org.apache.tinkerpop.gremlin.driver.ser.GraphSONMessageSerializerV2d0, config:
{ serializeResultToString: true }}
```

make sure to wrap the value of hosts parameter within brackets [].

3. In your terminal, run `bin/gremlin.bat` or `bin/gremlin.sh` to start the Gremlin Console.

4. In your terminal, run `:remote connect tinkerpop.server conf/remote-secure.yaml` to connect to your app service.

> **TIP**
>
> If you receive the error `No appenders could be found for logger` ensure that you updated the serializer value in the remote-secure.yaml file as described in step 2. If your configuration is correct, then this warning can be safely ignored as it should not impact the use of the console.

5. Next run `:remote console` to redirect all console commands to the remote server.

> **NOTE**
> If you don't run the `:remote console` command but would like to redirect all console commands to the remote server, you should prefix the command with `:>`, for example you should run the command as `:> g.V().count()`. This prefix is a part of the command and it is important when using the Gremlin console with Azure Cosmos DB. Omitting this prefix instructs the console to execute the command locally, often against an in-memory graph. Using this prefix `:>` tells the console to execute a remote command, in this case against Azure Cosmos DB (either the localhost emulator, or an Azure instance).

Great! Now that we finished the setup, let's start running some console commands.

Let's try a simple count() command. Type the following into the console at the prompt:

```
g.V().count()
```

Create vertices and edges

Let's begin by adding five person vertices for *Thomas*, *Mary Kay*, *Robin*, *Ben*, and *Jack*.

Input (Thomas):

```
g.addV('person').property('firstName', 'Thomas').property('lastName', 'Andersen').property('age',
44).property('userid', 1).property('pk', 'pk')
```

Output:

```
==>[id:796cdccc-2acd-4e58-a324-91d6f6f5ed6d,label:person,type:vertex,properties:[firstName:[[id:f02a749f-
b67c-4016-850e-910242d68953,value:Thomas]],lastName:[[id:f5fa3126-8818-4fda-88b0-
9bb55145ce5c,value:Andersen]],age:[[id:f6390f9c-e563-433e-acbf-25627628016e,value:44]],userid:[[id:796cdccc-
2acd-4e58-a324-91d6f6f5ed6d|userid,value:1]]]]
```

Input (Mary Kay):

```
g.addV('person').property('firstName', 'Mary Kay').property('lastName', 'Andersen').property('age',
39).property('userid', 2).property('pk', 'pk')
```

Output:

```
==>[id:0ac9be25-a476-4a30-8da8-e79f0119ea5e,label:person,type:vertex,properties:[firstName:[[id:ea0604f8-
14ee-4513-a48a-1734a1f28dc0,value:Mary Kay]],lastName:[[id:86d3bba5-fd60-4856-9396-
c195ef7d7f4b,value:Andersen]],age:[[id:bc81b78d-30c4-4e03-8f40-50f72eb5f6da,value:39]],userid:[[id:0ac9be25-
a476-4a30-8da8-e79f0119ea5e|userid,value:2]]]]
```

Input (Robin):

```
g.addV('person').property('firstName', 'Robin').property('lastName', 'Wakefield').property('userid',
3).property('pk', 'pk')
```

Output:

```
==>[id:8dc14d6a-8683-4a54-8d74-7eef1fb43a3e,label:person,type:vertex,properties:[firstName:[[id:ec65f078-
7a43-4cbe-bc06-e50f2640dc4e,value:Robin]],lastName:[[id:a3937d07-0e88-45d3-a442-
26fcdfb042ce,value:Wakefield]],userid:[[id:8dc14d6a-8683-4a54-8d74-7eef1fb43a3e|userid,value:3]]]]
```

Input (Ben):

```
g.addV('person').property('firstName', 'Ben').property('lastName', 'Miller').property('userid',
4).property('pk', 'pk')
```

Output:

```
==>[id:ee86b670-4d24-4966-9a39-30529284b66f,label:person,type:vertex,properties:[firstName:[[id:a632469b-
30fc-4157-840c-b80260871e9a,value:Ben]],lastName:[[id:4a08d307-0719-47c6-84ae-
1b0b06630928,value:Miller]],userid:[[id:ee86b670-4d24-4966-9a39-30529284b66f|userid,value:4]]]]
```

Input (Jack):

```
g.addV('person').property('firstName', 'Jack').property('lastName', 'Connor').property('userid',
5).property('pk', 'pk')
```

Output:

```
==>[id:4c835f2a-ea5b-43bb-9b6b-215488ad8469,label:person,type:vertex,properties:[firstName:[[id:4250824e-
4b72-417f-af98-8034aa15559f,value:Jack]],lastName:[[id:44c1d5e1-a831-480a-bf94-
5167d133549e,value:Connor]],userid:[[id:4c835f2a-ea5b-43bb-9b6b-215488ad8469|userid,value:5]]]]
```

Next, let's add edges for relationships between our people.

Input (Thomas -> Mary Kay):

```
g.V().hasLabel('person').has('firstName',
'Thomas').addE('knows').to(g.V().hasLabel('person').has('firstName', 'Mary Kay'))
```

Output:

```
==>[id:c12bf9fb-96a1-4cb7-a3f8-
431e196e702f,label:knows,type:edge,inVLabel:person,outVLabel:person,inV:0d1fa428-780c-49a5-bd3a-
a68d96391d5c,outV:1ce821c6-aa3d-4170-a0b7-d14d2a4d18c3]
```

Input (Thomas -> Robin):

```
g.V().hasLabel('person').has('firstName',
'Thomas').addE('knows').to(g.V().hasLabel('person').has('firstName', 'Robin'))
```

Output:

```
==>[id:58319bdd-1d3e-4f17-a106-
0ddf18719d15,label:knows,type:edge,inVLabel:person,outVLabel:person,inV:3e324073-ccfc-4ae1-8675-
d450858ca116,outV:1ce821c6-aa3d-4170-a0b7-d14d2a4d18c3]
```

Input (Robin -> Ben):

```
g.V().hasLabel('person').has('firstName', 'Robin').addE('knows').to(g.V().hasLabel('person').has('firstName',
'Ben'))
```

Output:

```
==>[id:889c4d3c-549e-4d35-bc21-
a3d1bfa11e00,label:knows,type:edge,inVLabel:person,outVLabel:person,inV:40fd641d-546e-412a-abcc-
58fe53891aab,outV:3e324073-ccfc-4ae1-8675-d450858ca116]
```

Update a vertex

Let's update the *Thomas* vertex with a new age of *45*.

Input:

```
g.V().hasLabel('person').has('firstName', 'Thomas').property('age', 45)
```

Output:

```
==>[id:ae36f938-210e-445a-92df-519f2b64c8ec,label:person,type:vertex,properties:[firstName:[[id:872090b6-
6a77-456a-9a55-a59141d4ebc2,value:Thomas]],lastName:[[id:7ee7a39a-a414-4127-89b4-
870bc4ef99f3,value:Andersen]],age:[[id:a2a75d5a-ae70-4095-806d-a35abcbfe71d,value:45]]]]
```

Query your graph

Now, let's run a variety of queries against your graph.

First, let's try a query with a filter to return only people who are older than 40 years old.

Input (filter query):

```
g.V().hasLabel('person').has('age', gt(40))
```

Output:

```
==>[id:ae36f938-210e-445a-92df-519f2b64c8ec,label:person,type:vertex,properties:[firstName:[[id:872090b6-
6a77-456a-9a55-a59141d4ebc2,value:Thomas]],lastName:[[id:7ee7a39a-a414-4127-89b4-
870bc4ef99f3,value:Andersen]],age:[[id:a2a75d5a-ae70-4095-806d-a35abcbfe71d,value:45]]]]
```

Next, let's project the first name for the people who are older than 40 years old.

Input (filter + projection query):

```
g.V().hasLabel('person').has('age', gt(40)).values('firstName')
```

Output:

```
==>Thomas
```

Traverse your graph

Let's traverse the graph to return all of Thomas's friends.

Input (friends of Thomas):

```
g.V().hasLabel('person').has('firstName', 'Thomas').outE('knows').inV().hasLabel('person')
```

Output:

```
==>[id:f04bc00b-cb56-46c4-a3bb-a5870c42f7ff,label:person,type:vertex,properties:[firstName:[[id:14feedec-
b070-444e-b544-62be15c7167c,value:Mary Kay]],lastName:[[id:107ab421-7208-45d4-b969-
bbc54481992a,value:Andersen]],age:[[id:4b08d6e4-58f5-45df-8e69-6b790b692e0a,value:39]]]]
==>[id:91605c63-4988-4b60-9a30-5144719ae326,label:person,type:vertex,properties:[firstName:[[id:f760e0e6-
652a-481a-92b0-1767d9bf372e,value:Robin]],lastName:[[id:352a4caa-bad6-47e3-a7dc-
90ff342cf870,value:Wakefield]]]]
```

Next, let's get the next layer of vertices. Traverse the graph to return all the friends of Thomas's friends.

Input (friends of friends of Thomas):

```
g.V().hasLabel('person').has('firstName',
'Thomas').outE('knows').inV().hasLabel('person').outE('knows').inV().hasLabel('person')
```

Output:

```
==>[id:a801a0cb-ee85-44ee-a502-271685ef212e,label:person,type:vertex,properties:[firstName:[[id:b9489902-
d29a-4673-8c09-c2b3fe7f8b94,value:Ben]],lastName:[[id:e084f933-9a4b-4dbc-8273-f0171265cf1d,value:Miller]]]]
```

Drop a vertex

Let's now delete a vertex from the graph database.

Input (drop Jack vertex):

```
g.V().hasLabel('person').has('firstName', 'Jack').drop()
```

Clear your graph

Finally, let's clear the database of all vertices and edges.

Input:

```
g.E().drop()
g.V().drop()
```

Congratulations! You've completed this Azure Cosmos DB: Gremlin API tutorial!

Review SLAs in the Azure portal

The Azure portal monitors your Cosmos DB account throughput, storage, availability, latency, and consistency. Charts for metrics associated with an Azure Cosmos DB Service Level Agreement (SLA) show the SLA value compared to actual performance. This suite of metrics makes monitoring your SLAs transparent.

To review metrics and SLAs:

1. Select **Metrics** in your Cosmos DB account's navigation menu.

2. Select a tab such as **Latency**, and select a timeframe on the right. Compare the **Actual** and **SLA** lines on the charts.

3. Review the metrics on the other tabs.

Clean up resources

When you're done with your app and Azure Cosmos DB account, you can delete the Azure resources you created so you don't incur more charges. To delete the resources:

1. In the Azure portal Search bar, search for and select **Resource groups**.

2. From the list, select the resource group you created for this quickstart.

3. On the resource group **Overview** page, select **Delete resource group**.

4. In the next window, enter the name of the resource group to delete, and then select **Delete**.

Next steps

In this quickstart, you've learned how to create an Azure Cosmos DB account, create a graph using the Data Explorer, create vertices and edges, and traverse your graph using the Gremlin console. You can now build more

complex queries and implement powerful graph traversal logic using Gremlin.

Query using Gremlin

Quickstart: Build a .NET Framework or Core application using the Azure Cosmos DB Gremlin API account

12/4/2020 • 9 minutes to read • Edit Online

APPLIES TO: Gremlin API

Azure Cosmos DB is Microsoft's globally distributed multi-model database service. You can quickly create and query document, key/value, and graph databases, all of which benefit from the global distribution and horizontal scale capabilities at the core of Azure Cosmos DB.

This quickstart demonstrates how to create an Azure Cosmos DB Gremlin API account, database, and graph (container) using the Azure portal. You then build and run a console app built using the open-source driver Gremlin.Net.

Prerequisites

If you don't already have Visual Studio 2019 installed, you can download and use the **free** Visual Studio 2019 Community Edition. Make sure that you enable **Azure development** during the Visual Studio setup.

If you don't have an Azure subscription, create a free account before you begin.

Create a database account

1. In a new browser window, sign in to the Azure portal.

2. In the left menu, select **Create a resource**.

3. On the **New** page, select **Databases** > **Azure Cosmos DB**.

> **TIP**
>
> If you don't see Azure Cosmos DB in the list, just type it in the search box at the top of the page, and press *Enter* key.

4. On the **Create Azure Cosmos DB Account** page, enter the settings for the new Azure Cosmos DB account.

SETTING	VALUE	DESCRIPTION
Subscription	Your subscription	Select the Azure subscription that you want to use for this Azure Cosmos DB account.
Resource Group	Create new Then enter the same name as Account Name	Select **Create new**. Then enter a new resource group name for your account. For simplicity, use the same name as your Azure Cosmos DB account name.

SETTING	VALUE	DESCRIPTION
Account Name	Enter a unique name	Enter a unique name to identify your Azure Cosmos DB account. Your account URI will be *gremlin.azure.com* appended to your unique account name. The account name can use only lowercase letters, numbers, and hyphens (-), and must be between 3 and 31 characters long.
API	Gremlin (graph)	The API determines the type of account to create. Azure Cosmos DB provides five APIs: Core (SQL) for document databases, Gremlin for graph databases, MongoDB for document databases, Azure Table, and Cassandra. You must create a separate account for each API. Select **Gremlin (graph)**, because in this quickstart you are creating a table that works with the Gremlin API. Learn more about the Gremlin API.
Location	Select the region closest to your users	Select a geographic location to host your Azure Cosmos DB account. Use the location that's closest to your users to give them the fastest access to the data.
Capacity mode	Provisioned throughput or Serverless	Select **Provisioned throughput** to create an account in provisioned throughput mode. Select **Serverless** to create an account in serverless mode.

Select **Review+Create**. You can skip the **Network** and **Tags** section.

5. The account creation takes a few minutes. Wait for the portal to display the **Congratulations! Your Azure Cosmos DB account was created** page.

Add a graph

You can now use the Data Explorer tool in the Azure portal to create a graph database.

1. Select **Data Explorer** > **New Graph**.

The **Add Graph** area is displayed on the far right, you may need to scroll right to see it.

2. In the **Add graph** page, enter the settings for the new graph.

SETTING	SUGGESTED VALUE	DESCRIPTION
Database ID	sample-database	Enter *sample-database* as the name for the new database. Database names must be between 1 and 255 characters, and cannot contain `/ \ # ?` or a trailing space.
Throughput	400 RUs	Change the throughput to 400 request units per second (RU/s). If you want to reduce latency, you can scale up the throughput later.
Graph ID	sample-graph	Enter *sample-graph* as the name for your new collection. Graph names have the same character requirements as database IDs.
Partition Key	/pk	All Cosmos DB accounts need a partition key to horizontally scale. Learn how to select an appropriate partition key in the Graph Data Partitioning article.

3. Once the form is filled out, select **OK**.

Clone the sample application

Now let's clone a Gremlin API app from GitHub, set the connection string, and run it. You'll see how easy it is to work with data programmatically.

1. Open a command prompt, create a new folder named git-samples, then close the command prompt.

```
md "C:\git-samples"
```

2. Open a git terminal window, such as git bash, and use the `cd` command to change to the new folder to install the sample app.

```
cd "C:\git-samples"
```

3. Run the following command to clone the sample repository. This command creates a copy of the sample app on your computer.

```
git clone https://github.com/Azure-Samples/azure-cosmos-db-graph-gremlindotnet-getting-started.git
```

4. Then open Visual Studio and open the solution file.
5. Restore the NuGet packages in the project. This should include the Gremlin.Net driver, as well as the Newtonsoft.Json package.
6. You can also install the Gremlin.Net driver manually using the Nuget package manager, or the nuget command-line utility:

```
nuget install Gremlin.Net
```

Review the code

This step is optional. If you're interested in learning how the database resources are created in the code, you can review the following snippets. Otherwise, you can skip ahead to Update your connection string.

The following snippets are all taken from the Program.cs file.

- Set your connection parameters based on the account created above:

```
private static string Host => Environment.GetEnvironmentVariable("Host") ?? throw new
ArgumentException("Missing env var: Host");
private static string PrimaryKey => Environment.GetEnvironmentVariable("PrimaryKey") ?? throw new
ArgumentException("Missing env var: PrimaryKey");
private static string Database => Environment.GetEnvironmentVariable("DatabaseName") ?? throw new
ArgumentException("Missing env var: DatabaseName");
private static string Container => Environment.GetEnvironmentVariable("ContainerName") ?? throw new
ArgumentException("Missing env var: ContainerName");

private static bool EnableSSL
{
    get
    {
        if (Environment.GetEnvironmentVariable("EnableSSL") == null)
        {
            return true;
        }

        if (!bool.TryParse(Environment.GetEnvironmentVariable("EnableSSL"), out bool value))
        {
            throw new ArgumentException("Invalid env var: EnableSSL is not a boolean");
        }

        return value;
    }
}

private static int Port
{
    get
    {
        if (Environment.GetEnvironmentVariable("Port") == null)
        {
            return 443;
        }

        if (!int.TryParse(Environment.GetEnvironmentVariable("Port"), out int port))
        {
            throw new ArgumentException("Invalid env var: Port is not an integer");
        }

        return port;
    }
}
```

- The Gremlin commands to be executed are listed in a Dictionary:

```
private static Dictionary<string, string> gremlinQueries = new Dictionary<string, string>
{
    { "Cleanup",        "g.V().drop()" },
    { "AddVertex 1",    "g.addV('person').property('id', 'thomas').property('firstName',
'Thomas').property('age', 44).property('pk', 'pk')" },
    { "AddVertex 2",    "g.addV('person').property('id', 'mary').property('firstName',
'Mary').property('lastName', 'Andersen').property('age', 39).property('pk', 'pk')" },
    { "AddVertex 3",    "g.addV('person').property('id', 'ben').property('firstName',
'Ben').property('lastName', 'Miller').property('pk', 'pk')" },
    { "AddVertex 4",    "g.addV('person').property('id', 'robin').property('firstName',
'Robin').property('lastName', 'Wakefield').property('pk', 'pk')" },
    { "AddEdge 1",      "g.V('thomas').addE('knows').to(g.V('mary'))" },
    { "AddEdge 2",      "g.V('thomas').addE('knows').to(g.V('ben'))" },
    { "AddEdge 3",      "g.V('ben').addE('knows').to(g.V('robin'))" },
    { "UpdateVertex",   "g.V('thomas').property('age', 44)" },
    { "CountVertices",  "g.V().count()" },
    { "Filter Range",   "g.V().hasLabel('person').has('age', gt(40))" },
    { "Project",        "g.V().hasLabel('person').values('firstName')" },
    { "Sort",           "g.V().hasLabel('person').order().by('firstName', decr)" },
    { "Traverse",       "g.V('thomas').out('knows').hasLabel('person')" },
    { "Traverse 2x",
"g.V('thomas').out('knows').hasLabel('person').out('knows').hasLabel('person')" },
    { "Loop",           "g.V('thomas').repeat(out()).until(has('id', 'robin')).path()" },
    { "DropEdge",       "g.V('thomas').outE('knows').where(inV().has('id', 'mary')).drop()" },
    { "CountEdges",     "g.E().count()" },
    { "DropVertex",     "g.V('thomas').drop()" },
};
```

- Create a new `GremlinServer` and `GremlinClient` connection objects using the parameters provided above:

```
string containerLink = "/dbs/" + Database + "/colls/" + Container;
Console.WriteLine($"Connecting to: host: {Host}, port: {Port}, container: {containerLink}, ssl:
{EnableSSL}");
var gremlinServer = new GremlinServer(Host, Port, enableSsl: EnableSSL,
                                      username: containerLink,
                                      password: PrimaryKey);

ConnectionPoolSettings connectionPoolSettings = new ConnectionPoolSettings()
{
    MaxInProcessPerConnection = 10,
    PoolSize = 30,
    ReconnectionAttempts= 3,
    ReconnectionBaseDelay = TimeSpan.FromMilliseconds(500)
};

var webSocketConfiguration =
    new Action<ClientWebSocketOptions>(options =>
    {
        options.KeepAliveInterval = TimeSpan.FromSeconds(10);
    });


using (var gremlinClient = new GremlinClient(
    gremlinServer,
    new GraphSON2Reader(),
    new GraphSON2Writer(),
    GremlinClient.GraphSON2MimeType,
    connectionPoolSettings,
    webSocketConfiguration))
{
```

- Execute each Gremlin query using the `GremlinClient` object with an async task. You can read the Gremlin queries from the dictionary defined in the previous step and execute them. Later get the result and read

the values, which are formatted as a dictionary, using the `JsonSerializer` class from Newtonsoft.Json package:

```
foreach (var query in gremlinQueries)
{
    Console.WriteLine(String.Format("Running this query: {0}: {1}", query.Key, query.Value));

    // Create async task to execute the Gremlin query.
    var resultSet = SubmitRequest(gremlinClient, query).Result;
    if (resultSet.Count > 0)
    {
        Console.WriteLine("\tResult:");
        foreach (var result in resultSet)
        {
            // The vertex results are formed as Dictionaries with a nested dictionary for their
properties
            string output = JsonConvert.SerializeObject(result);
            Console.WriteLine($"\t{output}");
        }
        Console.WriteLine();
    }

    // Print the status attributes for the result set.
    // This includes the following:
    //  x-ms-status-code            : This is the sub-status code which is specific to Cosmos DB.
    //  x-ms-total-request-charge   : The total request units charged for processing a request.
    //  x-ms-total-server-time-ms   : The total time executing processing the request on the server.
    PrintStatusAttributes(resultSet.StatusAttributes);
    Console.WriteLine();
}
```

Update your connection string

Now go back to the Azure portal to get your connection string information and copy it into the app.

1. From the Azure portal, navigate to your graph database account. In the **Overview** tab, you can see two endpoints-

   .NET SDK URI - This value is used when you connect to the graph account by using Microsoft.Azure.Graphs library.

   Gremlin Endpoint - This value is used when you connect to the graph account by using Gremlin.Net library.

To run this sample, copy the **Gremlin Endpoint** value, delete the port number at the end, that is the URI becomes `https://<your cosmos db account name>.gremlin.cosmosdb.azure.com`. The endpoint value should look like `testgraphacct.gremlin.cosmosdb.azure.com`

2. Next, navigate to the **Keys** tab and copy the **PRIMARY KEY** value from the Azure portal.

3. After you have copied the URI and PRIMARY KEY of your account, save them to a new environment variable on the local machine running the application. To set the environment variable, open a command prompt window, and run the following command. Make sure to replace <Your_Azure_Cosmos_account_URI> and <Your_Azure_Cosmos_account_PRIMARY_KEY> values.

   ```
   setx Host "<your Azure Cosmos account name>.gremlin.cosmosdb.azure.com"
   setx PrimaryKey "<Your_Azure_Cosmos_account_PRIMARY_KEY>"
   ```

4. Open the *Program.cs* file and update the "database and "container" variables with the database and container (which is also the graph name) names created above.

   ```
   private static string database = "your-database-name";
   ```

   ```
   private static string container = "your-container-or-graph-name";
   ```

5. Save the Program.cs file.

You've now updated your app with all the info it needs to communicate with Azure Cosmos DB.

Run the console app

Click CTRL + F5 to run the application. The application will print both the Gremlin query commands and results in the console.

The console window displays the vertexes and edges being added to the graph. When the script completes, press ENTER to close the console window.

Browse using the Data Explorer

You can now go back to Data Explorer in the Azure portal and browse and query your new graph data.

1. In Data Explorer, the new database appears in the Graphs pane. Expand the database and container nodes, and then click **Graph**.

2. Click the **Apply Filter** button to use the default query to view all the vertices in the graph. The data generated by the sample app is displayed in the Graphs pane.

   You can zoom in and out of the graph, you can expand the graph display space, add additional vertices, and move vertices on the display surface.

Review SLAs in the Azure portal

The Azure portal monitors your Cosmos DB account throughput, storage, availability, latency, and consistency. Charts for metrics associated with an Azure Cosmos DB Service Level Agreement (SLA) show the SLA value compared to actual performance. This suite of metrics makes monitoring your SLAs transparent.

To review metrics and SLAs:

1. Select **Metrics** in your Cosmos DB account's navigation menu.
2. Select a tab such as **Latency**, and select a timeframe on the right. Compare the **Actual** and **SLA** lines on the charts.

3. Review the metrics on the other tabs.

Clean up resources

When you're done with your app and Azure Cosmos DB account, you can delete the Azure resources you created so you don't incur more charges. To delete the resources:

1. In the Azure portal Search bar, search for and select **Resource groups**.
2. From the list, select the resource group you created for this quickstart.

3. On the resource group **Overview** page, select **Delete resource group**.

4. In the next window, enter the name of the resource group to delete, and then select **Delete**.

Next steps

In this quickstart, you've learned how to create an Azure Cosmos DB account, create a graph using the Data Explorer, and run an app. You can now build more complex queries and implement powerful graph traversal logic using Gremlin.

Query using Gremlin

Quickstart: Build a graph database with the Java SDK and the Azure Cosmos DB Gremlin API

12/4/2020 • 9 minutes to read • Edit Online

APPLIES TO: Gremlin API

In this quickstart, you create and manage an Azure Cosmos DB Gremlin (graph) API account from the Azure portal, and add data by using a Java app cloned from GitHub. Azure Cosmos DB is a multi-model database service that lets you quickly create and query document, table, key-value, and graph databases with global distribution and horizontal scale capabilities.

Prerequisites

- An Azure account with an active subscription. Create one for free.
- Java Development Kit (JDK) 8. Point your `JAVA_HOME` environment variable to the folder where the JDK is installed.
- A Maven binary archive.
- Git.

Create a database account

Before you can create a graph database, you need to create a Gremlin (Graph) database account with Azure Cosmos DB.

1. In a new browser window, sign in to the Azure portal.

2. In the left menu, select **Create a resource**.

3. On the **New** page, select **Databases** > **Azure Cosmos DB**.

> **TIP**
>
> If you don't see Azure Cosmos DB in the list, just type it in the search box at the top of the page, and press *Enter* key.

4. On the **Create Azure Cosmos DB Account** page, enter the settings for the new Azure Cosmos DB account.

SETTING	VALUE	DESCRIPTION
Subscription	Your subscription	Select the Azure subscription that you want to use for this Azure Cosmos DB account.
Resource Group	Create new Then enter the same name as Account Name	Select **Create new**. Then enter a new resource group name for your account. For simplicity, use the same name as your Azure Cosmos DB account name.

SETTING	VALUE	DESCRIPTION
Account Name	Enter a unique name	Enter a unique name to identify your Azure Cosmos DB account. Your account URI will be *gremlin.azure.com* appended to your unique account name. The account name can use only lowercase letters, numbers, and hyphens (-), and must be between 3 and 31 characters long.
API	Gremlin (graph)	The API determines the type of account to create. Azure Cosmos DB provides five APIs: Core (SQL) for document databases, Gremlin for graph databases, MongoDB for document databases, Azure Table, and Cassandra. You must create a separate account for each API. Select **Gremlin (graph)**, because in this quickstart you are creating a table that works with the Gremlin API. Learn more about the Gremlin API.
Location	Select the region closest to your users	Select a geographic location to host your Azure Cosmos DB account. Use the location that's closest to your users to give them the fastest access to the data.
Capacity mode	Provisioned throughput or Serverless	Select **Provisioned throughput** to create an account in provisioned throughput mode. Select **Serverless** to create an account in serverless mode.

Select **Review+Create**. You can skip the **Network** and **Tags** section.

5. The account creation takes a few minutes. Wait for the portal to display the **Congratulations! Your Azure Cosmos DB account was created** page.

Add a graph

You can now use the Data Explorer tool in the Azure portal to create a graph database.

1. Select **Data Explorer** > **New Graph**.

   The **Add Graph** area is displayed on the far right, you may need to scroll right to see it.

2. In the **Add graph** page, enter the settings for the new graph.

SETTING	SUGGESTED VALUE	DESCRIPTION
Database ID	sample-database	Enter *sample-database* as the name for the new database. Database names must be between 1 and 255 characters, and cannot contain `/ \ # ?` or a trailing space.
Throughput	400 RUs	Change the throughput to 400 request units per second (RU/s). If you want to reduce latency, you can scale up the throughput later.
Graph ID	sample-graph	Enter *sample-graph* as the name for your new collection. Graph names have the same character requirements as database IDs.
Partition Key	/pk	All Cosmos DB accounts need a partition key to horizontally scale. Learn how to select an appropriate partition key in the Graph Data Partitioning article.

3. Once the form is filled out, select **OK**.

Clone the sample application

Now let's switch to working with code. Let's clone a Gremlin API app from GitHub, set the connection string, and run it. You'll see how easy it is to work with data programmatically.

1. Open a command prompt, create a new folder named git-samples, then close the command prompt.

```
md "C:\git-samples"
```

2. Open a git terminal window, such as git bash, and use the `cd` command to change to a folder to install the sample app.

```
cd "C:\git-samples"
```

3. Run the following command to clone the sample repository. This command creates a copy of the sample app on your computer.

```
git clone https://github.com/Azure-Samples/azure-cosmos-db-graph-java-getting-started.git
```

Review the code

This step is optional. If you're interested in learning how the database resources are created in the code, you can review the following snippets. Otherwise, you can skip ahead to Update your connection string.

The following snippets are all taken from the *C:\git-samples\azure-cosmos-db-graph-java-getting-started\src\GetStarted\Program.java* file.

This Java console app uses a Gremlin API database with the OSS Apache TinkerPop driver.

- The Gremlin `Client` is initialized from the configuration in the *C:\git-samples\azure-cosmos-db-graph-java-getting-started\src\remote.yaml* file.

```
cluster = Cluster.build(new File("src/remote.yaml")).create();
...
client = cluster.connect();
```

- Series of Gremlin steps are executed using the `client.submit` method.

```
ResultSet results = client.submit(gremlin);

CompletableFuture<List<Result>> completableFutureResults = results.all();
List<Result> resultList = completableFutureResults.get();

for (Result result : resultList) {
    System.out.println(result.toString());
}
```

Update your connection information

Now go back to the Azure portal to get your connection information and copy it into the app. These settings enable your app to communicate with your hosted database.

1. In your Azure Cosmos DB account in the Azure portal, select **Keys**.

   Copy the first portion of the URI value.

2. Open the *src/remote.yaml* file and paste the unique ID value over `$name$` in `hosts: [$name$.graphs.azure.com]`.

   Line 1 of *remote.yaml* should now look similar to

   `hosts: [test-graph.graphs.azure.com]`

3. Change `graphs` to `gremlin.cosmosdb` in the `endpoint` value. (If you created your graph database account before December 20, 2017, make no changes to the endpoint value and continue to the next step.)

   The endpoint value should now look like this:

   `"endpoint": "https://testgraphacct.gremlin.cosmosdb.azure.com:443/"`

4. In the Azure portal, use the copy button to copy the PRIMARY KEY and paste it over `$masterKey$` in `password: $masterKey$`.

   Line 4 of *remote.yaml* should now look similar to

   `password: 2Ggkr662ifxz2Mg==`

5. Change line 3 of *remote.yaml* from

   `username: /dbs/$database$/colls/$collection$`

   to

   `username: /dbs/sample-database/colls/sample-graph`

   If you used a unique name for your sample database or graph, update the values as appropriate.

6. Save the *remote.yaml* file.

Run the console app

1. In the git terminal window, `cd` to the azure-cosmos-db-graph-java-getting-started folder.

```
cd "C:\git-samples\azure-cosmos-db-graph-java-getting-started"
```

2. In the git terminal window, use the following command to install the required Java packages.

```
mvn package
```

3. In the git terminal window, use the following command to start the Java application.

```
mvn exec:java -D exec.mainClass=GetStarted.Program
```

The terminal window displays the vertices being added to the graph.

If you experience timeout errors, check that you updated the connection information correctly in Update your connection information, and also try running the last command again.

Once the program stops, select Enter, then switch back to the Azure portal in your internet browser.

Review and add sample data

You can now go back to Data Explorer and see the vertices added to the graph, and add additional data points.

1. In your Azure Cosmos DB account in the Azure portal, select **Data Explorer**, expand **sample-graph**, select **Graph**, and then select **Apply Filter**.

2. In the **Results** list, notice the new users added to the graph. Select **ben** and notice that the user is connected to robin. You can move the vertices around by dragging and dropping, zoom in and out by scrolling the wheel of your mouse, and expand the size of the graph with the double-arrow.

3. Let's add a few new users. Select **New Vertex** to add data to your graph.

4. In the label box, enter *person*.

5. Select **Add property** to add each of the following properties. Notice that you can create unique properties for each person in your graph. Only the id key is required.

KEY	VALUE	NOTES
id	ashley	The unique identifier for the vertex. If you don't specify an id, one is generated for you.
gender	female	
tech	java	

> **NOTE**
> In this quickstart you create a non-partitioned collection. However, if you create a partitioned collection by specifying a partition key during the collection creation, then you need to include the partition key as a key in each new vertex.

6. Select **OK**. You may need to expand your screen to see **OK** on the bottom of the screen.

7. Select **New Vertex** again and add an additional new user.

8. Enter a label of *person*.

9. Select **Add property** to add each of the following properties:

KEY	VALUE	NOTES
id	rakesh	The unique identifier for the vertex. If you don't specify an id, one is generated for you.
gender	male	

KEY	VALUE	NOTES
school	MIT	

10. Select **OK**.

11. Select the **Apply Filter** button with the default `g.V()` filter to display all the values in the graph. All of the users now show in the **Results** list.

    As you add more data, you can use filters to limit your results. By default, Data Explorer uses `g.V()` to retrieve all vertices in a graph. You can change it to a different graph query, such as `g.V().count()`, to return a count of all the vertices in the graph in JSON format. If you changed the filter, change the filter back to `g.V()` and select **Apply Filter** to display all the results again.

12. Now you can connect rakesh, and ashley. Ensure **ashley** is selected in the **Results** list, then select next to **Targets** on lower right side. You may need to widen your window to see the button.

13. In the **Target** box enter *rakesh*, and in the **Edge label** box enter *knows*, and then select the check box.

14. Now select **rakesh** from the results list and see that ashley and rakesh are connected.

That completes the resource creation part of this tutorial. You can continue to add vertexes to your graph, modify

the existing vertexes, or change the queries. Now let's review the metrics Azure Cosmos DB provides, and then clean up the resources.

Review SLAs in the Azure portal

The Azure portal monitors your Cosmos DB account throughput, storage, availability, latency, and consistency. Charts for metrics associated with an Azure Cosmos DB Service Level Agreement (SLA) show the SLA value compared to actual performance. This suite of metrics makes monitoring your SLAs transparent.

To review metrics and SLAs:

1. Select **Metrics** in your Cosmos DB account's navigation menu.

2. Select a tab such as **Latency**, and select a timeframe on the right. Compare the **Actual** and **SLA** lines on the charts.

3. Review the metrics on the other tabs.

Clean up resources

When you're done with your app and Azure Cosmos DB account, you can delete the Azure resources you created so you don't incur more charges. To delete the resources:

1. In the Azure portal Search bar, search for and select **Resource groups**.

2. From the list, select the resource group you created for this quickstart.

3. On the resource group **Overview** page, select **Delete resource group**.

4. In the next window, enter the name of the resource group to delete, and then select **Delete**.

Next steps

In this quickstart, you learned how to create an Azure Cosmos DB account, create a graph using the Data Explorer, and run a Java app that adds data to the graph. You can now build more complex queries and implement powerful graph traversal logic using Gremlin.

Query using Gremlin

Quickstart: Build a Node.js application by using Azure Cosmos DB Gremlin API account

12/4/2020 • 7 minutes to read • Edit Online

APPLIES TO: Gremlin API

In this quickstart, you create and manage an Azure Cosmos DB Gremlin (graph) API account from the Azure portal, and add data by using a Node.js app cloned from GitHub. Azure Cosmos DB is a multi-model database service that lets you quickly create and query document, table, key-value, and graph databases with global distribution and horizontal scale capabilities.

Prerequisites

- An Azure account with an active subscription. Create one for free.
- Node.js 0.10.29+.
- Git.

Create a database account

1. In a new browser window, sign in to the Azure portal.
2. In the left menu, select **Create a resource**.

3. On the **New** page, select **Databases** > **Azure Cosmos DB**.

> **TIP**
>
> If you don't see Azure Cosmos DB in the list, just type it in the search box at the top of the page, and press *Enter* key.

4. On the **Create Azure Cosmos DB Account** page, enter the settings for the new Azure Cosmos DB account.

SETTING	VALUE	DESCRIPTION
Subscription	Your subscription	Select the Azure subscription that you want to use for this Azure Cosmos DB account.
Resource Group	Create new Then enter the same name as Account Name	Select **Create new**. Then enter a new resource group name for your account. For simplicity, use the same name as your Azure Cosmos DB account name.

SETTING	VALUE	DESCRIPTION
Account Name	Enter a unique name	Enter a unique name to identify your Azure Cosmos DB account. Your account URI will be *gremlin.azure.com* appended to your unique account name. The account name can use only lowercase letters, numbers, and hyphens (-), and must be between 3 and 31 characters long.
API	Gremlin (graph)	The API determines the type of account to create. Azure Cosmos DB provides five APIs: Core (SQL) for document databases, Gremlin for graph databases, MongoDB for document databases, Azure Table, and Cassandra. You must create a separate account for each API. Select **Gremlin (graph)**, because in this quickstart you are creating a table that works with the Gremlin API. Learn more about the Gremlin API.
Location	Select the region closest to your users	Select a geographic location to host your Azure Cosmos DB account. Use the location that's closest to your users to give them the fastest access to the data.
Capacity mode	Provisioned throughput or Serverless	Select **Provisioned throughput** to create an account in provisioned throughput mode. Select **Serverless** to create an account in serverless mode.

Select **Review+Create**. You can skip the **Network** and **Tags** section.

5. The account creation takes a few minutes. Wait for the portal to display the **Congratulations! Your Azure Cosmos DB account was created** page.

Add a graph

You can now use the Data Explorer tool in the Azure portal to create a graph database.

1. Select **Data Explorer** > **New Graph**.

   The **Add Graph** area is displayed on the far right, you may need to scroll right to see it.

2. In the **Add graph** page, enter the settings for the new graph.

SETTING	SUGGESTED VALUE	DESCRIPTION
Database ID	sample-database	Enter *sample-database* as the name for the new database. Database names must be between 1 and 255 characters, and cannot contain `/ \ # ?` or a trailing space.
Throughput	400 RUs	Change the throughput to 400 request units per second (RU/s). If you want to reduce latency, you can scale up the throughput later.
Graph ID	sample-graph	Enter *sample-graph* as the name for your new collection. Graph names have the same character requirements as database IDs.
Partition Key	/pk	All Cosmos DB accounts need a partition key to horizontally scale. Learn how to select an appropriate partition key in the Graph Data Partitioning article.

3. Once the form is filled out, select **OK**.

Clone the sample application

Now let's clone a Gremlin API app from GitHub, set the connection string, and run it. You'll see how easy it is to work with data programmatically.

1. Open a command prompt, create a new folder named git-samples, then close the command prompt.

```
md "C:\git-samples"
```

2. Open a git terminal window, such as git bash, and use the `cd` command to change to the new folder to install the sample app.

```
cd "C:\git-samples"
```

3. Run the following command to clone the sample repository. This command creates a copy of the sample app on your computer.

```
git clone https://github.com/Azure-Samples/azure-cosmos-db-graph-nodejs-getting-started.git
```

4. Open the solution file in Visual Studio.

Review the code

This step is optional. If you're interested in learning how the database resources are created in the code, you can review the following snippets. Otherwise, you can skip ahead to Update your connection string.

The following snippets are all taken from the *app.js* file.

This console app uses the open-source Gremlin Node.js driver.

- The Gremlin client is created.

```
const authenticator = new Gremlin.driver.auth.PlainTextSaslAuthenticator(
    `/dbs/${config.database}/colls/${config.collection}`,
    config.primaryKey
)


const client = new Gremlin.driver.Client(
    config.endpoint,
    {
        authenticator,
        traversalsource : "g",
        rejectUnauthorized : true,
        mimeType : "application/vnd.gremlin-v2.0+json"
    }
);
```

  The configurations are all in *config.js*, which we edit in the following section.

- A series of functions are defined to execute different Gremlin operations. This is one of them:

```
function addVertex1()
{
    console.log('Running Add Vertex1');
    return client.submit("g.addV(label).property('id', id).property('firstName',
firstName).property('age', age).property('userid', userid).property('pk', 'pk')", {
            label:"person",
            id:"thomas",
            firstName:"Thomas",
            age:44, userid: 1
        }).then(function (result) {
                console.log("Result: %s\n", JSON.stringify(result));
        });
}
```

- Each function executes a `client.execute` method with a Gremlin query string parameter. Here is an example of how `g.V().count()` is executed:

```
function countVertices()
{
    console.log('Running Count');
    return client.submit("g.V().count()", { }).then(function (result) {
        console.log("Result: %s\n", JSON.stringify(result));
    });
}
```

- At the end of the file, all methods are then invoked. This will execute them one after the other:

```
client.open()
.then(dropGraph)
.then(addVertex1)
.then(addVertex2)
.then(addEdge)
.then(countVertices)
.catch((err) => {
    console.error("Error running query...");
    console.error(err)
}).then((res) => {
    client.close();
    finish();
}).catch((err) =>
    console.error("Fatal error:", err)
);
```

Update your connection string

1. Open the *config.js* file.
2. In *config.js*, fill in the `config.endpoint` key with the **Gremlin Endpoint** value from the **Overview** page of your Cosmos DB account in the Azure portal.

```
config.endpoint = "https://<your_Gremlin_account_name>.gremlin.cosmosdb.azure.com:443/";
```

3. In *config.js*, fill in the config.primaryKey value with the **Primary Key** value from the **Keys** page of your Cosmos DB account in the Azure portal.

```
config.primaryKey = "PRIMARYKEY";
```

4. Enter the database name, and graph (container) name for the value of config.database and config.collection.

Here's an example of what your completed *config.js* file should look like:

```
var config = {}

// Note that this must include the protocol (HTTPS:// for .NET SDK URI or wss:// for Gremlin Endpoint) and
the port number
config.endpoint = "https://testgraphacct.gremlin.cosmosdb.azure.com:443/";
config.primaryKey =
"Pams6e7LEUS7LJ2Qk0fjZf3eGo65JdMWHmyn65i52w8ozPX2oxY3iP0yu05t9v1WymAHNcMwPIqNAEv3XDFsEg==";
config.database = "graphdb"
config.collection = "Persons"

module.exports = config;
```

Run the console app

1. Open a terminal window and change (via `cd` command) to the installation directory for the *package.json* file that's included in the project.

2. Run `npm install` to install the required npm modules, including `gremlin`.

3. Run `node app.js` in a terminal to start your node application.

Browse with Data Explorer

You can now go back to Data Explorer in the Azure portal to view, query, modify, and work with your new graph data.

In Data Explorer, the new database appears in the **Graphs** pane. Expand the database, followed by the container, and then select **Graph**.

The data generated by the sample app is displayed in the next pane within the **Graph** tab when you select **Apply Filter**.

Try completing `g.V()` with `.has('firstName', 'Thomas')` to test the filter. Note that the value is case sensitive.

Review SLAs in the Azure portal

The Azure portal monitors your Cosmos DB account throughput, storage, availability, latency, and consistency. Charts for metrics associated with an Azure Cosmos DB Service Level Agreement (SLA) show the SLA value compared to actual performance. This suite of metrics makes monitoring your SLAs transparent.

To review metrics and SLAs:

1. Select **Metrics** in your Cosmos DB account's navigation menu.
2. Select a tab such as **Latency**, and select a timeframe on the right. Compare the **Actual** and **SLA** lines on the charts.

3. Review the metrics on the other tabs.

Clean up your resources

When you're done with your app and Azure Cosmos DB account, you can delete the Azure resources you created so you don't incur more charges. To delete the resources:

1. In the Azure portal Search bar, search for and select **Resource groups**.
2. From the list, select the resource group you created for this quickstart.

3. On the resource group **Overview** page, select **Delete resource group**.

4. In the next window, enter the name of the resource group to delete, and then select **Delete**.

Next steps

In this article, you learned how to create an Azure Cosmos DB account, create a graph by using Data Explorer, and run a Node.js app to add data to the graph. You can now build more complex queries and implement powerful graph traversal logic by using Gremlin.

Query by using Gremlin

Quickstart: Create a graph database in Azure Cosmos DB using Python and the Azure portal

12/4/2020 • 9 minutes to read • Edit Online

APPLIES TO: Gremlin API

In this quickstart, you create and manage an Azure Cosmos DB Gremlin (graph) API account from the Azure portal, and add data by using a Python app cloned from GitHub. Azure Cosmos DB is a multi-model database service that lets you quickly create and query document, table, key-value, and graph databases with global distribution and horizontal scale capabilities.

Prerequisites

- An Azure account with an active subscription. Create one for free. Or try Azure Cosmos DB for free without an Azure subscription.
- Python 3.5+ including pip package installer.
- Python Driver for Gremlin.
- Git.

> **NOTE**
> This quickstart requires a graph database account created after December 20th, 2017. Existing accounts will support Python once they're migrated to general availability.

Create a database account

Before you can create a graph database, you need to create a Gremlin (Graph) database account with Azure Cosmos DB.

1. In a new browser window, sign in to the Azure portal.

2. In the left menu, select **Create a resource**.

3. On the **New** page, select **Databases** > **Azure Cosmos DB**.

> **TIP**
>
> If you don't see Azure Cosmos DB in the list, just type it in the search box at the top of the page, and press *Enter* key.

4. On the **Create Azure Cosmos DB Account** page, enter the settings for the new Azure Cosmos DB account.

SETTING	VALUE	DESCRIPTION
Subscription	Your subscription	Select the Azure subscription that you want to use for this Azure Cosmos DB account.
Resource Group	Create new Then enter the same name as Account Name	Select **Create new**. Then enter a new resource group name for your account. For simplicity, use the same name as your Azure Cosmos DB account name.

SETTING	VALUE	DESCRIPTION
Account Name	Enter a unique name	Enter a unique name to identify your Azure Cosmos DB account. Your account URI will be *gremlin.azure.com* appended to your unique account name. The account name can use only lowercase letters, numbers, and hyphens (-), and must be between 3 and 31 characters long.
API	Gremlin (graph)	The API determines the type of account to create. Azure Cosmos DB provides five APIs: Core (SQL) for document databases, Gremlin for graph databases, MongoDB for document databases, Azure Table, and Cassandra. You must create a separate account for each API. Select **Gremlin (graph)**, because in this quickstart you are creating a table that works with the Gremlin API. Learn more about the Gremlin API.
Location	Select the region closest to your users	Select a geographic location to host your Azure Cosmos DB account. Use the location that's closest to your users to give them the fastest access to the data.
Capacity mode	Provisioned throughput or Serverless	Select **Provisioned throughput** to create an account in provisioned throughput mode. Select **Serverless** to create an account in serverless mode.

Select **Review+Create**. You can skip the **Network** and **Tags** section.

5. The account creation takes a few minutes. Wait for the portal to display the **Congratulations! Your Azure Cosmos DB account was created** page.

Add a graph

You can now use the Data Explorer tool in the Azure portal to create a graph database.

1. Select **Data Explorer** > **New Graph**.

   The **Add Graph** area is displayed on the far right, you may need to scroll right to see it.

2. In the **Add graph** page, enter the settings for the new graph.

SETTING	SUGGESTED VALUE	DESCRIPTION
Database ID	sample-database	Enter *sample-database* as the name for the new database. Database names must be between 1 and 255 characters, and cannot contain `/ \ # ?` or a trailing space.
Throughput	400 RUs	Change the throughput to 400 request units per second (RU/s). If you want to reduce latency, you can scale up the throughput later.
Graph ID	sample-graph	Enter *sample-graph* as the name for your new collection. Graph names have the same character requirements as database IDs.
Partition Key	/pk	All Cosmos DB accounts need a partition key to horizontally scale. Learn how to select an appropriate partition key in the Graph Data Partitioning article.

3. Once the form is filled out, select **OK**.

Clone the sample application

Now let's switch to working with code. Let's clone a Gremlin API app from GitHub, set the connection string, and run it. You'll see how easy it is to work with data programmatically.

1. Open a command prompt, create a new folder named git-samples, then close the command prompt.

```
md "C:\git-samples"
```

2. Open a git terminal window, such as git bash, and use the `cd` command to change to a folder to install the sample app.

```
cd "C:\git-samples"
```

3. Run the following command to clone the sample repository. This command creates a copy of the sample app on your computer.

```
git clone https://github.com/Azure-Samples/azure-cosmos-db-graph-python-getting-started.git
```

Review the code

This step is optional. If you're interested in learning how the database resources are created in the code, you can review the following snippets. The snippets are all taken from the *connect.py* file in the *C:\git-samples\azure-cosmos-db-graph-python-getting-started* folder. Otherwise, you can skip ahead to Update your connection string.

- The Gremlin `client` is initialized in line 104 in *connect.py*:

```
...
client = client.Client('wss://<YOUR_ENDPOINT>.gremlin.cosmosdb.azure.com:443/','g',
    username="/dbs/<YOUR_DATABASE>/colls/<YOUR_COLLECTION_OR_GRAPH>",
    password="<YOUR_PASSWORD>")
...
```

- A series of Gremlin steps are declared at the beginning of the *connect.py* file. They are then executed using the `client.submitAsync()` method:

```
client.submitAsync(_gremlin_cleanup_graph)
```

Update your connection information

Now go back to the Azure portal to get your connection information and copy it into the app. These settings enable your app to communicate with your hosted database.

1. In your Azure Cosmos DB account in the Azure portal, select **Keys**.

   Copy the first portion of the URI value.

2. Open the *connect.py* file and in line 104 paste the URI value over `<YOUR_ENDPOINT>` in here:

```
client = client.Client('wss://<YOUR_ENDPOINT>.gremlin.cosmosdb.azure.com:443/','g',
    username="/dbs/<YOUR_DATABASE>/colls/<YOUR_COLLECTION_OR_GRAPH>",
    password="<YOUR_PASSWORD>")
```

   The URI portion of the client object should now look similar to this code:

```
client = client.Client('wss://test.gremlin.cosmosdb.azure.com:443/','g',
    username="/dbs/<YOUR_DATABASE>/colls/<YOUR_COLLECTION_OR_GRAPH>",
    password="<YOUR_PASSWORD>")
```

3. Change the second parameter of the `client` object to replace the `<YOUR_DATABASE>` and `<YOUR_COLLECTION_OR_GRAPH>` strings. If you used the suggested values, the parameter should look like this code:

   `username="/dbs/sample-database/colls/sample-graph"`

   The entire `client` object should now look like this code:

```
client = client.Client('wss://test.gremlin.cosmosdb.azure.com:443/','g',
    username="/dbs/sample-database/colls/sample-graph",
    password="<YOUR_PASSWORD>")
```

4. On the **Keys** page, use the copy button to copy the PRIMARY KEY and paste it over `<YOUR_PASSWORD>` in the `password=<YOUR_PASSWORD>` parameter.

   The entire `client` object definition should now look like this code:

```
client = client.Client('wss://test.gremlin.cosmosdb.azure.com:443/','g',
    username="/dbs/sample-database/colls/sample-graph",
    password="asdb13Fadsf14FASc22Ggkr662ifxz2Mg==")
```

5. Save the *connect.py* file.

Run the console app

1. In the git terminal window, `cd` to the azure-cosmos-db-graph-python-getting-started folder.

```
cd "C:\git-samples\azure-cosmos-db-graph-python-getting-started"
```

2. In the git terminal window, use the following command to install the required Python packages.

```
pip install -r requirements.txt
```

3. In the git terminal window, use the following command to start the Python application.

```
python connect.py
```

The terminal window displays the vertices and edges being added to the graph.

If you experience timeout errors, check that you updated the connection information correctly in Update your connection information, and also try running the last command again.

Once the program stops, press Enter, then switch back to the Azure portal in your internet browser.

Review and add sample data

After the vertices and edges are inserted, you can now go back to Data Explorer and see the vertices added to the graph, and add additional data points.

1. In your Azure Cosmos DB account in the Azure portal, select **Data Explorer**, expand **sample-graph**, select **Graph**, and then select **Apply Filter**.

2. In the **Results** list, notice three new users are added to the graph. You can move the vertices around by dragging and dropping, zoom in and out by scrolling the wheel of your mouse, and expand the size of the graph with the double-arrow.

3. Let's add a few new users. Select the **New Vertex** button to add data to your graph.

4. Enter a label of *person*.

5. Select **Add property** to add each of the following properties. Notice that you can create unique properties for each person in your graph. Only the id key is required.

KEY	VALUE	NOTES
pk	/pk	
id	ashley	The unique identifier for the vertex. If you don't specify an id, one is generated for you.
gender	female	
tech	java	

> **NOTE**
> In this quickstart create a non-partitioned collection. However, if you create a partitioned collection by specifying a partition key during the collection creation, then you need to include the partition key as a key in each new vertex.

6. Select **OK**. You may need to expand your screen to see **OK** on the bottom of the screen.
7. Select **New Vertex** again and add an additional new user.
8. Enter a label of *person*.
9. Select **Add property** to add each of the following properties:

KEY	VALUE	NOTES
pk	/pk	
id	rakesh	The unique identifier for the vertex. If you don't specify an id, one is generated for you.
gender	male	
school	MIT	

10. Select **OK**.
11. Select the **Apply Filter** button with the default `g.V()` filter to display all the values in the graph. All of the users now show in the **Results** list.

    As you add more data, you can use filters to limit your results. By default, Data Explorer uses `g.V()` to retrieve all vertices in a graph. You can change it to a different graph query, such as `g.V().count()`, to return a count of all the vertices in the graph in JSON format. If you changed the filter, change the filter back to `g.V()` and select **Apply Filter** to display all the results again.

12. Now we can connect rakesh and ashley. Ensure **ashley** is selected in the **Results** list, then select the edit button next to **Targets** on lower right side. You may need to widen your window to see the **Properties** area.

13. In the **Target** box type *rakesh*, and in the **Edge label** box type *knows*, and then select the check.

14. Now select **rakesh** from the results list and see that ashley and rakesh are connected.

That completes the resource creation part of this tutorial. You can continue to add vertexes to your graph, modify the existing vertexes, or change the queries. Now let's review the metrics Azure Cosmos DB provides, and then clean up the resources.

Review SLAs in the Azure portal

The Azure portal monitors your Cosmos DB account throughput, storage, availability, latency, and consistency. Charts for metrics associated with an Azure Cosmos DB Service Level Agreement (SLA) show the SLA value compared to actual performance. This suite of metrics makes monitoring your SLAs transparent.

To review metrics and SLAs:

1. Select **Metrics** in your Cosmos DB account's navigation menu.

2. Select a tab such as **Latency**, and select a timeframe on the right. Compare the **Actual** and **SLA** lines on the charts.

3. Review the metrics on the other tabs.

Clean up resources

When you're done with your app and Azure Cosmos DB account, you can delete the Azure resources you created so you don't incur more charges. To delete the resources:

1. In the Azure portal Search bar, search for and select **Resource groups**.

2. From the list, select the resource group you created for this quickstart.

3. On the resource group **Overview** page, select **Delete resource group**.

4. In the next window, enter the name of the resource group to delete, and then select **Delete**.

Next steps

In this quickstart, you learned how to create an Azure Cosmos DB account, create a graph using the Data Explorer, and run a Python app to add data to the graph. You can now build more complex queries and implement powerful graph traversal logic using Gremlin.

Query using Gremlin

Quickstart: Create a graph database in Azure Cosmos DB using PHP and the Azure portal

12/4/2020 • 9 minutes to read • Edit Online

APPLIES TO: Gremlin API

This quickstart shows how to use PHP and the Azure Cosmos DB Gremlin API to build a console app by cloning an example from GitHub. This quickstart also walks you through the creation of an Azure Cosmos DB account by using the web-based Azure portal.

Azure Cosmos DB is Microsoft's globally distributed multi-model database service. You can quickly create and query document, table, key-value, and graph databases, all of which benefit from the global distribution and horizontal scale capabilities at the core of Azure Cosmos DB.

Prerequisites

If you don't have an Azure subscription, create a free account before you begin. Alternatively, you can Try Azure Cosmos DB for free without an Azure subscription, free of charge and commitments.

In addition:

- PHP 5.6 or newer
- Composer

Create a database account

Before you can create a graph database, you need to create a Gremlin (Graph) database account with Azure Cosmos DB.

1. In a new browser window, sign in to the Azure portal.
2. In the left menu, select **Create a resource**.

3. On the **New** page, select **Databases** > **Azure Cosmos DB**.

TIP

If you don't see Azure Cosmos DB in the list, just type it in the search box at the top of the page, and press *Enter* key.

4. On the **Create Azure Cosmos DB Account** page, enter the settings for the new Azure Cosmos DB account.

SETTING	VALUE	DESCRIPTION
Subscription	Your subscription	Select the Azure subscription that you want to use for this Azure Cosmos DB account.
Resource Group	Create new Then enter the same name as Account Name	Select **Create new**. Then enter a new resource group name for your account. For simplicity, use the same name as your Azure Cosmos DB account name.

SETTING	VALUE	DESCRIPTION
Account Name	Enter a unique name	Enter a unique name to identify your Azure Cosmos DB account. Your account URI will be *gremlin.azure.com* appended to your unique account name. The account name can use only lowercase letters, numbers, and hyphens (-), and must be between 3 and 31 characters long.
API	Gremlin (graph)	The API determines the type of account to create. Azure Cosmos DB provides five APIs: Core (SQL) for document databases, Gremlin for graph databases, MongoDB for document databases, Azure Table, and Cassandra. You must create a separate account for each API. Select **Gremlin (graph)**, because in this quickstart you are creating a table that works with the Gremlin API. Learn more about the Gremlin API.
Location	Select the region closest to your users	Select a geographic location to host your Azure Cosmos DB account. Use the location that's closest to your users to give them the fastest access to the data.
Capacity mode	Provisioned throughput or Serverless	Select **Provisioned throughput** to create an account in provisioned throughput mode. Select **Serverless** to create an account in serverless mode.

Select **Review+Create**. You can skip the **Network** and **Tags** section.

5. The account creation takes a few minutes. Wait for the portal to display the **Congratulations! Your Azure Cosmos DB account was created** page.

Add a graph

You can now use the Data Explorer tool in the Azure portal to create a graph database.

1. Select **Data Explorer** > **New Graph**.

   The **Add Graph** area is displayed on the far right, you may need to scroll right to see it.

2. In the **Add graph** page, enter the settings for the new graph.

SETTING	SUGGESTED VALUE	DESCRIPTION
Database ID	sample-database	Enter *sample-database* as the name for the new database. Database names must be between 1 and 255 characters, and cannot contain `/ \ # ?` or a trailing space.
Throughput	400 RUs	Change the throughput to 400 request units per second (RU/s). If you want to reduce latency, you can scale up the throughput later.
Graph ID	sample-graph	Enter *sample-graph* as the name for your new collection. Graph names have the same character requirements as database IDs.
Partition Key	/pk	All Cosmos DB accounts need a partition key to horizontally scale. Learn how to select an appropriate partition key in the Graph Data Partitioning article.

3. Once the form is filled out, select **OK**.

Clone the sample application

Now let's switch to working with code. Let's clone a Gremlin API app from GitHub, set the connection string, and run it. You'll see how easy it is to work with data programmatically.

1. Open a command prompt, create a new folder named git-samples, then close the command prompt.

```
md "C:\git-samples"
```

2. Open a git terminal window, such as git bash, and use the `cd` command to change to a folder to install the sample app.

```
cd "C:\git-samples"
```

3. Run the following command to clone the sample repository. This command creates a copy of the sample app on your computer.

```
git clone https://github.com/Azure-Samples/azure-cosmos-db-graph-php-getting-started.git
```

Review the code

This step is optional. If you're interested in learning how the database resources are created in the code, you can review the following snippets. The snippets are all taken from the connect.php file in the C:\git-samples\azure-cosmos-db-graph-php-getting-started\ folder. Otherwise, you can skip ahead to Update your connection string.

- The Gremlin `connection` is initialized in the beginning of the `connect.php` file using the `$db` object.

```
$db = new Connection([
    'host' => '<your_server_address>.graphs.azure.com',
    'username' => '/dbs/<db>/colls/<coll>',
    'password' => 'your_primary_key'
    ,'port' => '443'

    // Required parameter
    ,'ssl' => TRUE
]);
```

- A series of Gremlin steps are executed using the `$db->send($query);` method.

```
$query = "g.V().drop()";
...
$result = $db->send($query);
$errors = array_filter($result);
}
```

Update your connection information

Now go back to the Azure portal to get your connection information and copy it into the app. These settings enable your app to communicate with your hosted database.

1. In the Azure portal, click **Keys**.

   Copy the first portion of the URI value.

2. Open the `connect.php` file and in line 8 paste the URI value over `your_server_address`.

   The connection object initialization should now look similar to the following code:

```
$db = new Connection([
    'host' => 'testgraphacct.gremlin.cosmosdb.azure.com',
    'username' => '/dbs/<db>/colls/<coll>',
    'password' => 'your_primary_key'
    ,'port' => '443'

    // Required parameter
    ,'ssl' => TRUE
]);
```

3. Change `username` parameter in the Connection object with your database and graph name. If you used the recommended values of `sample-database` and `sample-graph`, it should look like the following code:

   `'username' => '/dbs/sample-database/colls/sample-graph'`

   The entire Connection object should look like the following code snippet at this time:

```
$db = new Connection([
    'host' => 'testgraphacct.gremlin.cosmosdb.azure.com',
    'username' => '/dbs/sample-database/colls/sample-graph',
    'password' => 'your_primary_key',
    'port' => '443'

    // Required parameter
    ,'ssl' => TRUE
]);
```

4. In the Azure portal, use the copy button to copy the PRIMARY KEY and paste it over `your_primary_key` in the password parameter.

   The Connection object initialization should now look like the following code:

```
$db = new Connection([
    'host' => 'testgraphacct.graphs.azure.com',
    'username' => '/dbs/sample-database/colls/sample-graph',
    'password' => '2Ggkr662ifxz2Mg==',
    'port' => '443'

    // Required parameter
    ,'ssl' => TRUE
]);
```

5. Save the `connect.php` file.

Run the console app

1. In the git terminal window, `cd` to the azure-cosmos-db-graph-php-getting-started folder.

```
cd "C:\git-samples\azure-cosmos-db-graph-php-getting-started"
```

2. In the git terminal window, use the following command to install the required PHP dependencies.

```
composer install
```

3. In the git terminal window, use the following command to start the PHP application.

```
php connect.php
```

   The terminal window displays the vertices being added to the graph.

   If you experience timeout errors, check that you updated the connection information correctly in Update your connection information, and also try running the last command again.

   Once the program stops, press Enter, then switch back to the Azure portal in your internet browser.

Review and add sample data

You can now go back to Data Explorer and see the vertices added to the graph, and add additional data points.

1. Click **Data Explorer**, expand **sample-graph**, click **Graph**, and then click **Apply Filter**.

2. In the **Results** list, notice the new users added to the graph. Select **ben** and notice that they're connected to robin. You can move the vertices around by dragging and dropping, zoom in and out by scrolling the wheel of your mouse, and expand the size of the graph with the double-arrow.

3. Let's add a few new users. Click the **New Vertex** button to add data to your graph.

4. Enter a label of *person*.

5. Click **Add property** to add each of the following properties. Notice that you can create unique properties for each person in your graph. Only the **id** key is required.

KEY	VALUE	NOTES
id	ashley	The unique identifier for the vertex. If you don't specify an id, one is generated for you.
gender	female	
tech	java	

> **NOTE**
> In this quickstart you create a non-partitioned collection. However, if you create a partitioned collection by specifying a partition key during the collection creation, then you need to include the partition key as a key in each new vertex.

6. Click **OK**. You may need to expand your screen to see **OK** on the bottom of the screen.

7. Click **New Vertex** again and add an additional new user.

8. Enter a label of *person*.

9. Click **Add property** to add each of the following properties:

KEY	VALUE	NOTES
id	rakesh	The unique identifier for the vertex. If you don't specify an id, one is generated for you.

KEY	VALUE	NOTES
gender	male	
school	MIT	

10. Click **OK**.

11. Click the **Apply Filter** button with the default `g.V()` filter to display all the values in the graph. All of the users now show in the **Results** list.

    As you add more data, you can use filters to limit your results. By default, Data Explorer uses `g.V()` to retrieve all vertices in a graph. You can change it to a different graph query, such as `g.V().count()`, to return a count of all the vertices in the graph in JSON format. If you changed the filter, change the filter back to `g.V()` and click **Apply Filter** to display all the results again.

12. Now you can connect rakesh and ashley. Ensure **ashley** is selected in the **Results** list, then click the edit button next to **Targets** on lower right side. You may need to widen your window to see the **Properties** area.

13. In the **Target** box type *rakesh*, and in the **Edge label** box type *knows*, and then click the check.

14. Now select **rakesh** from the results list and see that ashley and rakesh are connected.

That completes the resource creation part of this quickstart. You can continue to add vertexes to your graph, modify the existing vertexes, or change the queries. Now let's review the metrics Azure Cosmos DB provides, and then clean up the resources.

Review SLAs in the Azure portal

The Azure portal monitors your Cosmos DB account throughput, storage, availability, latency, and consistency. Charts for metrics associated with an Azure Cosmos DB Service Level Agreement (SLA) show the SLA value compared to actual performance. This suite of metrics makes monitoring your SLAs transparent.

To review metrics and SLAs:

1. Select **Metrics** in your Cosmos DB account's navigation menu.
2. Select a tab such as **Latency**, and select a timeframe on the right. Compare the **Actual** and **SLA** lines on the charts.

3. Review the metrics on the other tabs.

Clean up resources

When you're done with your app and Azure Cosmos DB account, you can delete the Azure resources you created so you don't incur more charges. To delete the resources:

1. In the Azure portal Search bar, search for and select **Resource groups**.
2. From the list, select the resource group you created for this quickstart.

3. On the resource group **Overview** page, select **Delete resource group**.

4. In the next window, enter the name of the resource group to delete, and then select **Delete**.

Next steps

In this quickstart, you've learned how to create an Azure Cosmos DB account, create a graph using the Data Explorer, and run an app. You can now build more complex queries and implement powerful graph traversal logic using Gremlin.

Query using Gremlin

Tutorial: Query Azure Cosmos DB Gremlin API by using Gremlin

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: Gremlin API

The Azure Cosmos DB Gremlin API supports Gremlin queries. This article provides sample documents and queries to get you started. A detailed Gremlin reference is provided in the Gremlin support article.

This article covers the following tasks:

- Querying data with Gremlin

Prerequisites

For these queries to work, you must have an Azure Cosmos DB account and have graph data in the container. Don't have any of those? Complete the 5-minute quickstart or the developer tutorial to create an account and populate your database. You can run the following queries using the Gremlin console, or your favorite Gremlin driver.

Count vertices in the graph

The following snippet shows how to count the number of vertices in the graph:

```
g.V().count()
```

Filters

You can perform filters using Gremlin's `has` and `hasLabel` steps, and combine them using `and`, `or`, and `not` to build more complex filters. Azure Cosmos DB provides schema-agnostic indexing of all properties within your vertices and degrees for fast queries:

```
g.V().hasLabel('person').has('age', gt(40))
```

Projection

You can project certain properties in the query results using the `values` step:

```
g.V().hasLabel('person').values('firstName')
```

Find related edges and vertices

So far, we've only seen query operators that work in any database. Graphs are fast and efficient for traversal operations when you need to navigate to related edges and vertices. Let's find all friends of Thomas. We do this by using Gremlin's `outE` step to find all the out-edges from Thomas, then traversing to the in-vertices from those edges using Gremlin's `inV` step:

```
g.V('thomas').outE('knows').inV().hasLabel('person')
```

The next query performs two hops to find all of Thomas' "friends of friends", by calling `outE` and `inV` two times.

```
g.V('thomas').outE('knows').inV().hasLabel('person').outE('knows').inV().hasLabel('person')
```

You can build more complex queries and implement powerful graph traversal logic using Gremlin, including mixing filter expressions, performing looping using the `loop` step, and implementing conditional navigation using the `choose` step. Learn more about what you can do with Gremlin support!

Next steps

In this tutorial, you've done the following:

- Learned how to query using Graph

You can now proceed to the Concepts section for more information about Cosmos DB.

Global distribution

Frequently asked questions about the Gremlin API in Azure Cosmos DB

12/4/2020 • 4 minutes to read • Edit Online

APPLIES TO: Gremlin API

This article explains answers to some frequently asked questions about Gremlin API in Azure Cosmos DB.

How to evaluate the efficiency of Gremlin queries

The **executionProfile()** preview step can be used to provide an analysis of the query execution plan. This step needs to be added to the end of any Gremlin query as illustrated by the following example:

Query example

```
g.V('mary').out('knows').executionProfile()
```

Example output

```
[
  {
    "gremlin": "g.V('mary').out('knows').executionProfile()",
    "totalTime": 8,
    "metrics": [
      {
        "name": "GetVertices",
        "time": 3,
        "annotations": {
          "percentTime": 37.5
        },
        "counts": {
          "resultCount": 1
        }
      },
      {
        "name": "GetEdges",
        "time": 5,
        "annotations": {
          "percentTime": 62.5
        },
        "counts": {
          "resultCount": 0
        },
        "storeOps": [
          {
            "count": 0,
            "size": 0,
            "time": 0.6
          }
        ]
      },
      {
        "name": "GetNeighborVertices",
        "time": 0,
        "annotations": {
          "percentTime": 0
        },
        "counts": {
          "resultCount": 0
        }
      },
      {
        "name": "ProjectOperator",
        "time": 0,
        "annotations": {
          "percentTime": 0
        },
        "counts": {
          "resultCount": 0
        }
      }
    ]
  }
]
```

The output of the above profile shows how much time is spent obtaining the vertex objects, the edge objects, and the size of the working data set. This is related to the standard cost measurements for Azure Cosmos DB queries.

Other frequently asked questions

How are RU/s charged when running queries on a graph database?

All graph objects, vertices, and edges, are shown as JSON documents in the backend. Since one Gremlin query can modify one or many graph objects at a time, the cost associated with it is directly related to the objects, edges that

are processed by the query. This is the same process that Azure Cosmos DB uses for all other APIs. For more information, see Request Units in Azure Cosmos DB.

The RU charge is based on the working data set of the traversal, and not the result set. For example, if a query aims to obtain a single vertex as a result but needs to traverse more than one other object on the way, then the cost will be based on all the graph objects that it will take to compute the one result vertex.

What's the maximum scale that a graph database can have in Azure Cosmos DB Gremlin API?

Azure Cosmos DB makes use of horizontal partitioning to automatically address increase in storage and throughput requirements. The maximum throughput and storage capacity of a workload is determined by the number of partitions that are associated with a given container. However, a Gremlin API container has a specific set of guidelines to ensure a proper performance experience at scale. For more information about partitioning, and best practices, see partitioning in Azure Cosmos DB article.

For C#/.NET development, should I use the Microsoft.Azure.Graphs package or Gremlin.NET?

Azure Cosmos DB Gremlin API leverages the open-source drivers as the main connectors for the service. So the recommended option is to use drivers that are supported by Apache Tinkerpop.

How can I protect against injection attacks using Gremlin drivers?

Most native Apache Tinkerpop Gremlin drivers allow the option to provide a dictionary of parameters for query execution. This is an example of how to do it in Gremlin.Net and in Gremlin-Javascript.

Why am I getting the "Gremlin Query Compilation Error: Unable to find any method" error?

Azure Cosmos DB Gremlin API implements a subset of the functionality defined in the Gremlin surface area. For supported steps and more information, see Gremlin support article.

The best workaround is to rewrite the required Gremlin steps with the supported functionality, since all essential Gremlin steps are supported by Azure Cosmos DB.

Why am I getting the "WebSocketException: The server returned status code '200' when status code '101' was expected" error?

This error is likely thrown when the wrong endpoint is being used. The endpoint that generates this error has the following pattern:

```
https:// YOUR_DATABASE_ACCOUNT.documents.azure.com:443/
```

This is the documents endpoint for your graph database. The correct endpoint to use is the Gremlin Endpoint, which has the following format:

```
https://YOUR_DATABASE_ACCOUNT.gremlin.cosmosdb.azure.com:443/
```

Why am I getting the "RequestRateIsTooLarge" error?

This error means that the allocated Request Units per second aren't enough to serve the query. This error is usually seen when you run a query that obtains all vertices:

```
// Query example:
g.V()
```

This query will attempt to retrieve all vertices from the graph. So, the cost of this query will be equal to at least the number of vertices in terms of RUs. The RU/s setting should be adjusted to address this query.

Why do my Gremlin driver connections get dropped eventually?

A Gremlin connection is made through a WebSocket connection. Although WebSocket connections don't have a specific time to live, Azure Cosmos DB Gremlin API will terminate idle connections after 30 minutes of inactivity.

Why can't I use fluent API calls in the native Gremlin drivers?

Fluent API calls aren't yet supported by the Azure Cosmos DB Gremlin API. Fluent API calls require an internal formatting feature known as bytecode support that currently isn't supported by Azure Cosmos DB Gremlin API. Due to the same reason, the latest Gremlin-JavaScript driver is also currently not supported.

Next steps

- Azure Cosmos DB Gremlin wire protocol support
- Create, query, and traverse an Azure Cosmos DB graph database using the Gremlin console

Azure Cosmos DB Gremlin server response headers

12/4/2020 • 6 minutes to read • Edit Online

APPLIES TO: Gremlin API

This article covers headers that Cosmos DB Gremlin server returns to the caller upon request execution. These headers are useful for troubleshooting request performance, building application that integrates natively with Cosmos DB service and simplifying customer support.

Keep in mind that taking dependency on these headers you are limiting portability of your application to other Gremlin implementations. In return, you are gaining tighter integration with Cosmos DB Gremlin. These headers are not a TinkerPop standard.

Headers

HEADER	TYPE	SAMPLE VALUE	WHEN INCLUDED	EXPLANATION
x-ms-request-charge	double	11.3243	Success and Failure	Amount of collection or database throughput consumed in request units (RU/s or RUs) for a partial response message. This header is present in every continuation for requests that have multiple chunks. It reflects the charge of a particular response chunk. Only for requests that consist of a single response chunk this header matches total cost of traversal. However, for majority of complex traversals this value represents a partial cost.

HEADER	TYPE	SAMPLE VALUE	WHEN INCLUDED	EXPLANATION
x-ms-total-request-charge	double	423.987	Success and Failure	Amount of collection or database throughput consumed in request units (RU/s or RUs) for entire request. This header is present in every continuation for requests that have multiple chunks. It indicates cumulative charge since the beginning of request. Value of this header in the last chunk indicates complete request charge.
x-ms-server-time-ms	double	13.75	Success and Failure	This header is included for latency troubleshooting purposes. It indicates the amount of time, in milliseconds, that Cosmos DB Gremlin server took to execute and produce a partial response message. Using value of this header and comparing it to overall request latency applications can calculate network latency overhead.
x-ms-total-server-time-ms	double	130.512	Success and Failure	Total time, in milliseconds, that Cosmos DB Gremlin server took to execute entire traversal. This header is included in every partial response. It represents cumulative execution time since the start of request. The last response indicates total execution time. This header is useful to differentiate between client and server as a source of latency. You can compare traversal execution time on the client to the value of this header.

HEADER	TYPE	SAMPLE VALUE	WHEN INCLUDED	EXPLANATION
x-ms-status-code	long	200	Success and Failure	Header indicates internal reason for request completion or termination. Application is advised to look at the value of this header and take corrective action.
x-ms-substatus-code	long	1003	Failure Only	Cosmos DB is a multi-model database that is built on top of unified storage layer. This header contains additional insights about the failure reason when failure occurs within lower layers of high availability stack. Application is advised to store this header and use it when contacting Cosmos DB customer support. Value of this header is useful for Cosmos DB engineer for quick troubleshooting.
x-ms-retry-after-ms	string (TimeSpan)	"00:00:03.9500000"	Failure Only	This header is a string representation of a .NET TimeSpan type. This value will only be included in requests failed due provisioned throughput exhaustion. Application should resubmit traversal again after instructed period of time.

HEADER	TYPE	SAMPLE VALUE	WHEN INCLUDED	EXPLANATION
x-ms-activity-id	string (Guid)	"A9218E01-3A3A-4716-9636-5BD86B056613"	Success and Failure	Header contains a unique server-side identifier of a request. Each request is assigned a unique identifier by the server for tracking purposes. Applications should log activity identifiers returned by the server for requests that customers may want to contact customer support about. Cosmos DB support personnel can find specific requests by these identifiers in Cosmos DB service telemetry.

Status codes

Most common status codes returned by the server are listed below.

STATUS	EXPLANATION
401	Error message `"Unauthorized: Invalid credentials provided"` is returned when authentication password doesn't match Cosmos DB account key. Navigate to your Cosmos DB Gremlin account in the Azure portal and confirm that the key is correct.
404	Concurrent operations that attempt to delete and update the same edge or vertex simultaneously. Error message `"Owner resource does not exist"` indicates that specified database or collection is incorrect in connection parameters in `/dbs/<database name>/colls/<collection or graph name>` format.
408	`"Server timeout"` indicates that traversal took more than **30 seconds** and was canceled by the server. Optimize your traversals to run quickly by filtering vertices or edges on every hop of traversal to narrow down search scope.
409	`"Conflicting request to resource has been attempted. Retry to avoid conflicts."` This usually happens when vertex or an edge with an identifier already exists in the graph.

STATUS	EXPLANATION
412	Status code is complemented with error message `"PreconditionFailedException": One of the specified pre-condition is not met`. This error is indicative of an optimistic concurrency control violation between reading an edge or vertex and writing it back to the store after modification. Most common situations when this error occurs is property modification, for example `g.V('identifier').property('name','value')`. Gremlin engine would read the vertex, modify it, and write it back. If there is another traversal running in parallel trying to write the same vertex or an edge, one of them will receive this error. Application should submit traversal to the server again.
429	Request was throttled and should be retried after value in **x-ms-retry-after-ms**
500	Error message that contains `"NotFoundException: Entity with the specified id does not exist in the system."` indicates that a database and/or collection was re-created with the same name. This error will disappear within 5 minutes as change propagates and invalidates caches in different Cosmos DB components. To avoid this issue, use unique database and collection names every time.
1000	This status code is returned when server successfully parsed a message but wasn't able to execute. It usually indicates a problem with the query.
1001	This code is returned when server completes traversal execution but fails to serialize response back to the client. This error can happen when traversal generates complex result, that is too large or does not conform to TinkerPop protocol specification. Application should simplify the traversal when it encounters this error.
1003	`"Query exceeded memory limit. Bytes Consumed: XXX, Max: YYY"` is returned when traversal exceeds allowed memory limit. Memory limit is **2 GB** per traversal.
1004	This status code indicates malformed graph request. Request can be malformed when it fails deserialization, non-value type is being deserialized as value type or unsupported gremlin operation requested. Application should not retry the request because it will not be successful.
1007	Usually this status code is returned with error message `"Could not process request. Underlying connection has been closed."`. This situation can happen if client driver attempts to use a connection that is being closed by the server. Application should retry the traversal on a different connection.

STATUS	EXPLANATION
1008	Cosmos DB Gremlin server can terminate connections to rebalance traffic in the cluster. Client drivers should handle this situation and use only live connections to send requests to the server. Occasionally client drivers may not detect that connection was closed. When application encounters an error, `"Connection is too busy. Please retry after sometime or open more connections."` it should retry traversal on a different connection.

Samples

A sample client application based on Gremlin.Net that reads one status attribute:

```
// Following example reads a status code and total request charge from server response attributes.
// Variable "server" is assumed to be assigned to an instance of a GremlinServer that is connected to Cosmos DB
account.
using (GremlinClient client = new GremlinClient(server, new GraphSON2Reader(), new GraphSON2Writer(),
GremlinClient.GraphSON2MimeType))
{
  ResultSet<dynamic> responseResultSet = await GremlinClientExtensions.SubmitAsync<dynamic>(client,
requestScript: "g.V().count()");
  long statusCode = (long)responseResultSet.StatusAttributes["x-ms-status-code"];
  double totalRequestCharge = (double)responseResultSet.StatusAttributes["x-ms-total-request-charge"];

  // Status code and request charge are logged into application telemetry.
}
```

An example that demonstrates how to read status attribute from Gremlin java client:

```
try {
  ResultSet resultSet = this.client.submit("g.addV().property('id', '13')");
  List<Result> results = resultSet.all().get();

  // Process and consume results

} catch (ResponseException re) {
  // Check for known errors that need to be retried or skipped
  if (re.getStatusAttributes().isPresent()) {
    Map<String, Object> attributes = re.getStatusAttributes().get();
    int statusCode = (int) attributes.getOrDefault("x-ms-status-code", -1);

    // Now we can check for specific conditions
    if (statusCode == 409) {
        // Handle conflicting writes
      }
    }

    // Check if we need to delay retry
    if (attributes.containsKey("x-ms-retry-after-ms")) {
      // Read the value of the attribute as is
      String retryAfterTimeSpan = (String) attributes.get("x-ms-retry-after-ms"));

      // Convert the value into actionable duration
   LocalTime locaTime = LocalTime.parse(retryAfterTimeSpan);
   Duration duration = Duration.between(LocalTime.MIN, locaTime);

      // Perform a retry after "duration" interval of time has elapsed
    }
  }
}
```

Next steps

- HTTP status codes for Azure Cosmos DB
- Common Azure Cosmos DB REST response headers
- TinkerPop Graph Driver Provider Requirements

Azure Cosmos DB Gremlin limits

11/2/2020 • 2 minutes to read • Edit Online

APPLIES TO: Gremlin API

This article talks about the limits of Azure Cosmos DB Gremlin engine and explains how they may impact customer traversals.

Cosmos DB Gremlin is built on top of Cosmos DB infrastructure. Due to this, all limits explained in Azure Cosmos DB service limits still apply.

Limits

When Gremlin limit is reached, traversal is canceled with a **x-ms-status-code** of 429 indicating a throttling error. See Gremlin server response headers for more information.

RESOURCE	DEFAULT LIMIT	EXPLANATION
Script length	**64 KB**	Maximum length of a Gremlin traversal script per request.
Operator depth	**400**	Total number of unique steps in a traversal. For example, `g.V().out()` has an operator count of 2: V() and out(), `g.V('label').repeat(out()).times(100)` has operator depth of 3: V(), repeat(), and out() because `.times(100)` is a parameter to `.repeat()` operator.
Degree of parallelism	**32**	Maximum number of storage partitions queried in a single request to storage layer. Graphs with hundreds of partitions will be impacted by this limit.
Repeat limit	**32**	Maximum number of iterations a `.repeat()` operator can execute. Each iteration of `.repeat()` step in most cases runs breadth-first traversal, which means that any traversal is limited to at most 32 hops between vertices.
Traversal timeout	**30 seconds**	Traversal will be canceled when it exceeds this time. Cosmos DB Graph is an OLTP database with vast majority of traversals completing within milliseconds. To run OLAP queries on Cosmos DB Graph, use Apache Spark with Graph Data Frames and Cosmos DB Spark Connector.

RESOURCE	DEFAULT LIMIT	EXPLANATION
Idle connection timeout	**1 hour**	Amount of time the Gremlin service will keep idle websocket connections open. TCP keep-alive packets or HTTP keep-alive requests don't extend connection lifespan beyond this limit. Cosmos DB Graph engine considers websocket connections to be idle if there are no active Gremlin requests running on it.
Resource token per hour	**100**	Number of unique resource tokens used by Gremlin clients to connect to Gremlin account in a region. When the application exceeds hourly unique token limit, `"Exceeded allowed resource token limit of 100 that can be used concurrently"` will be returned on the next authentication request.

Next steps

- Azure Cosmos DB Gremlin response headers
- Azure Cosmos DB Resource Tokens with Gremlin

Graph data modeling for Azure Cosmos DB Gremlin API

12/4/2020 • 5 minutes to read • Edit Online

APPLIES TO: Gremlin API

The following document is designed to provide graph data modeling recommendations. This step is vital in order to ensure the scalability and performance of a graph database system as the data evolves. An efficient data model is especially important with large-scale graphs.

Requirements

The process outlined in this guide is based on the following assumptions:

- The **entities** in the problem-space are identified. These entities are meant to be consumed *atomically* for each request. In other words, the database system isn't designed to retrieve a single entity's data in multiple query requests.
- There is an understanding of **read and write requirements** for the database system. These requirements will guide the optimizations needed for the graph data model.
- The principles of the Apache Tinkerpop property graph standard are well understood.

When do I need a graph database?

A graph database solution can be optimally applied if the entities and relationships in a data domain have any of the following characteristics:

- The entities are **highly connected** through descriptive relationships. The benefit in this scenario is the fact that the relationships are persisted in storage.
- There are **cyclic relationships** or **self-referenced entities**. This pattern is often a challenge when using relational or document databases.
- There are **dynamically evolving relationships** between entities. This pattern is especially applicable to hierarchical or tree-structured data with many levels.
- There are **many-to-many relationships** between entities.
- There are **write and read requirements on both entities and relationships**.

If the above criteria is satisfied, it's likely that a graph database approach will provide advantages for **query complexity**, **data model scalability**, and **query performance**.

The next step is to determine if the graph is going to be used for analytic or transactional purposes. If the graph is intended to be used for heavy computation and data processing workloads, it would be worth to explore the Cosmos DB Spark connector and the use of the GraphX library.

How to use graph objects

The Apache Tinkerpop property graph standard defines two types of objects **Vertices** and **Edges**.

The following are the best practices for the properties in the graph objects:

OBJECT	PROPERTY	TYPE	NOTES
Vertex	ID	String	Uniquely enforced per partition. If a value isn't supplied upon insertion, an auto-generated GUID will be stored.
Vertex	label	String	This property is used to define the type of entity that the vertex represents. If a value isn't supplied, a default value "vertex" will be used.
Vertex	properties	String, Boolean, Numeric	A list of separate properties stored as key-value pairs in each vertex.
Vertex	partition key	String, Boolean, Numeric	This property defines where the vertex and its outgoing edges will be stored. Read more about graph partitioning.
Edge	ID	String	Uniquely enforced per partition. Auto-generated by default. Edges usually don't have the need to be uniquely retrieved by an ID.
Edge	label	String	This property is used to define the type of relationship that two vertices have.
Edge	properties	String, Boolean, Numeric	A list of separate properties stored as key-value pairs in each edge.

> **NOTE**
> Edges don't require a partition key value, since its value is automatically assigned based on their source vertex. Learn more in the graph partitioning article.

Entity and relationship modeling guidelines

The following are a set of guidelines to approach data modeling for an Azure Cosmos DB Gremlin API graph database. These guidelines assume that there's an existing definition of a data domain and queries for it.

> **NOTE**
> The steps outlined below are presented as recommendations. The final model should be evaluated and tested before its consideration as production-ready. Additionally, the recommendations below are specific to Azure Cosmos DB's Gremlin API implementation.

Modeling vertices and properties

The first step for a graph data model is to map every identified entity to a **vertex object**. A one to one mapping of all entities to vertices should be an initial step and subject to change.

One common pitfall is to map properties of a single entity as separate vertices. Consider the example below, where the same entity is represented in two different ways:

- **Vertex-based properties**: In this approach, the entity uses three separate vertices and two edges to describe its properties. While this approach might reduce redundancy, it increases model complexity. An increase in model complexity can result in added latency, query complexity, and computation cost. This model can also present challenges in partitioning.

- **Property-embedded vertices**: This approach takes advantage of the key-value pair list to represent all the properties of the entity inside a vertex. This approach provides reduced model complexity, which will lead to simpler queries and more cost-efficient traversals.

> **NOTE**
>
> The above examples show a simplified graph model to only show the comparison between the two ways of dividing entity properties.

The **property-embedded vertices** pattern generally provides a more performant and scalable approach. The default approach to a new graph data model should gravitate towards this pattern.

However, there are scenarios where referencing to a property might provide advantages. For example: if the referenced property is updated frequently. Using a separate vertex to represent a property that is constantly changed would minimize the amount of write operations that the update would require.

Relationship modeling with edge directions

After the vertices are modeled, the edges can be added to denote the relationships between them. The first aspect that needs to be evaluated is the **direction of the relationship**.

Edge objects have a default direction that is followed by a traversal when using the `out()` or `outE()` function. Using this natural direction results in an efficient operation, since all vertices are stored with their outgoing edges.

However, traversing in the opposite direction of an edge, using the `in()` function, will always result in a cross-partition query. Learn more about graph partitioning. If there's a need to constantly traverse using the `in()` function, it's recommended to add edges in both directions.

You can determine the edge direction by using the `.to()` or `.from()` predicates to the `.addE()` Gremlin step. Or by using the bulk executor library for Gremlin API.

> **NOTE**
> Edge objects have a direction by default.

Relationship labeling

Using descriptive relationship labels can improve the efficiency of edge resolution operations. This pattern can be applied in the following ways:

- Use non-generic terms to label a relationship.
- Associate the label of the source vertex to the label of the target vertex with the relationship name.

The more specific the label that the traverser will use to filter the edges, the better. This decision can have a significant impact on query cost as well. You can evaluate the query cost at any time using the executionProfile step.

Next steps:

- Check out the list of supported Gremlin steps.
- Learn about graph database partitioning to deal with large-scale graphs.

- Evaluate your Gremlin queries using the Execution Profile step.

Using a partitioned graph in Azure Cosmos DB

11/2/2020 • 3 minutes to read • Edit Online

APPLIES TO: Gremlin API

One of the key features of the Gremlin API in Azure Cosmos DB is the ability to handle large-scale graphs through horizontal scaling. The containers can scale independently in terms of storage and throughput. You can create containers in Azure Cosmos DB that can be automatically scaled to store a graph data. The data is automatically balanced based on the specified **partition key**.

Partitioning is required if the container is expected to store more than 20 GB in size or if you want to allocate more than 10,000 request units per second (RUs). The same general principles from the Azure Cosmos DB partitioning mechanism apply with a few graph-specific optimizations described below.

Graph partitioning mechanism

The following guidelines describe how the partitioning strategy in Azure Cosmos DB operates:

- **Both vertices and edges are stored as JSON documents**.
- **Vertices require a partition key**. This key will determine in which partition the vertex will be stored through a hashing algorithm. The partition key property name is defined when creating a new container and it has a format: `/partitioning-key-name`.
- **Edges will be stored with their source vertex**. In other words, for each vertex its partition key defines where they are stored along with its outgoing edges. This optimization is done to avoid cross-partition queries when using the `out()` cardinality in graph queries.
- **Edges contain references to the vertices they point to**. All edges are stored with the partition keys and IDs of the vertices that they are pointing to. This computation makes all `out()` direction queries always be a scoped partitioned query, and not a blind cross-partition query.
- **Graph queries need to specify a partition key**. To take full advantage of the horizontal partitioning in Azure Cosmos DB, the partition key should be specified when a single vertex is selected, whenever it's possible. The following are queries for selecting one or multiple vertices in a partitioned graph:
  - `/id` and `/label` are not supported as partition keys for a container in Gremlin API.

- Selecting a vertex by ID, then **using the `.has()` step to specify the partition key property**:

```
g.V('vertex_id').has('partitionKey', 'partitionKey_value')
```

- Selecting a vertex by **specifying a tuple including partition key value and ID**:

```
g.V(['partitionKey_value', 'vertex_id'])
```

- Specifying an **array of tuples of partition key values and IDs**:

```
g.V(['partitionKey_value0', 'verted_id0'], ['partitionKey_value1', 'vertex_id1'], ...)
```

- Selecting a set of vertices with their IDs and **specifying a list of partition key values**:

```
g.V('vertex_id0', 'vertex_id1', 'vertex_id2', …).has('partitionKey',
within('partitionKey_value0', 'partitionKey_value01', 'partitionKey_value02', …)
```

- Using the **Partition strategy** at the beginning of a query and specifying a partition for the scope of the rest of the Gremlin query:

```
g.withStrategies(PartitionStrategy.build().partitionKey('partitionKey').readPartitions('partit
ionKey_value').create()).V()
```

Best practices when using a partitioned graph

Use the following guidelines to ensure performance and scalability when using partitioned graphs with unlimited containers:

- **Always specify the partition key value when querying a vertex**. Getting vertex from a known partition is a way to achieve performance. All subsequent adjacency operations will always be scoped to a partition since Edges contain reference ID and partition key to their target vertices.
- **Use the outgoing direction when querying edges whenever it's possible**. As mentioned above, edges are stored with their source vertices in the outgoing direction. So the chances of resorting to cross-partition queries are minimized when the data and queries are designed with this pattern in mind. On the contrary, the `in()` query will always be an expensive fan-out query.
- **Choose a partition key that will evenly distribute data across partitions**. This decision heavily depends on the data model of the solution. Read more about creating an appropriate partition key in Partitioning and scale in Azure Cosmos DB.
- **Optimize queries to obtain data within the boundaries of a partition**. An optimal partitioning strategy would be aligned to the querying patterns. Queries that obtain data from a single partition provide the best possible performance.

Next steps

Next you can proceed to read the following articles:

- Learn about Partition and scale in Azure Cosmos DB.
- Learn about the Gremlin support in Gremlin API.
- Learn about Introduction to Gremlin API.

Using the graph bulk executor .NET library to perform bulk operations in Azure Cosmos DB Gremlin API

12/4/2020 • 6 minutes to read • Edit Online

APPLIES TO: Gremlin API

This tutorial provides instructions about using Azure CosmosDB's bulk executor .NET library to import and update graph objects into an Azure Cosmos DB Gremlin API container. This process makes use of the Graph class in the bulk executor library to create Vertex and Edge objects programmatically to then insert multiple of them per network request. This behavior is configurable through the bulk executor library to make optimal use of both database and local memory resources.

As opposed to sending Gremlin queries to a database, where the command is evaluated and then executed one at a time, using the bulk executor library will instead require to create and validate the objects locally. After creating the objects, the library allows you to send graph objects to the database service sequentially. Using this method, data ingestion speeds can be increased up to 100x, which makes it an ideal method for initial data migrations or periodical data movement operations. Learn more by visiting the GitHub page of the Azure Cosmos DB Graph bulk executor sample application.

Bulk operations with graph data

The bulk executor library contains a `Microsoft.Azure.CosmosDB.BulkExecutor.Graph` namespace to provide functionality for creating and importing graph objects.

The following process outlines how data migration can be used for a Gremlin API container:

1. Retrieve records from the data source.
2. Construct `GremlinVertex` and `GremlinEdge` objects from the obtained records and add them into an `IEnumerable` data structure. In this part of the application the logic to detect and add relationships should be implemented, in case the data source is not a graph database.
3. Use the Graph BulkImportAsync method to insert the graph objects into the collection.

This mechanism will improve the data migration efficiency as compared to using a Gremlin client. This improvement is experienced because inserting data with Gremlin will require the application send a query at a time that will need to be validated, evaluated, and then executed to create the data. The bulk executor library will handle the validation in the application and send multiple graph objects at a time for each network request.

Creating Vertices and Edges

`GraphBulkExecutor` provides the `BulkImportAsync` method that requires a `IEnumerable` list of `GremlinVertex` or `GremlinEdge` objects, both defined in the `Microsoft.Azure.CosmosDB.BulkExecutor.Graph.Element` namespace. In the sample, we separated the edges and vertices into two BulkExecutor import tasks. See the example below:

```
IBulkExecutor graphbulkExecutor = new GraphBulkExecutor(documentClient, targetCollection);

BulkImportResponse vResponse = null;
BulkImportResponse eResponse = null;

try
{
    // Import a list of GremlinVertex objects
    vResponse = await graphbulkExecutor.BulkImportAsync(
            Utils.GenerateVertices(numberOfDocumentsToGenerate),
            enableUpsert: true,
            disableAutomaticIdGeneration: true,
            maxConcurrencyPerPartitionKeyRange: null,
            maxInMemorySortingBatchSize: null,
            cancellationToken: token);

    // Import a list of GremlinEdge objects
    eResponse = await graphbulkExecutor.BulkImportAsync(
            Utils.GenerateEdges(numberOfDocumentsToGenerate),
            enableUpsert: true,
            disableAutomaticIdGeneration: true,
            maxConcurrencyPerPartitionKeyRange: null,
            maxInMemorySortingBatchSize: null,
            cancellationToken: token);
}
catch (DocumentClientException de)
{
    Trace.TraceError("Document client exception: {0}", de);
}
catch (Exception e)
{
    Trace.TraceError("Exception: {0}", e);
}
```

For more information on the parameters of the bulk executor library, refer to the BulkImportData to Azure Cosmos DB topic.

The payload needs to be instantiated into `GremlinVertex` and `GremlinEdge` objects. Here is how these objects can be created:

Vertices:

```
// Creating a vertex
GremlinVertex v = new GremlinVertex(
    "vertexId",
    "vertexLabel");

// Adding custom properties to the vertex
v.AddProperty("customProperty", "value");

// Partitioning keys must be specified for all vertices
v.AddProperty("partitioningKey", "value");
```

Edges:

```
// Creating an edge
GremlinEdge e = new GremlinEdge(
    "edgeId",
    "edgeLabel",
    "targetVertexId",
    "sourceVertexId",
    "targetVertexLabel",
    "sourceVertexLabel",
    "targetVertexPartitioningKey",
    "sourceVertexPartitioningKey");

// Adding custom properties to the edge
e.AddProperty("customProperty", "value");
```

> **NOTE**
> The bulk executor utility doesn't automatically check for existing Vertices before adding Edges. This needs to be validated in the application before running the BulkImport tasks.

Sample application

Prerequisites

- Visual Studio 2019 with the Azure development workload. You can get started with the Visual Studio 2019 Community Edition for free.
- An Azure subscription. You can create a free Azure account here. Alternatively, you can create a Cosmos database account with Try Azure Cosmos DB for free without an Azure subscription.
- An Azure Cosmos DB Gremlin API database with an **unlimited collection**. This guide shows how to get started with Azure Cosmos DB Gremlin API in .NET.
- Git. For more information check out the Git Downloads page.

Clone the sample application

In this tutorial, we'll follow through the steps for getting started by using the Azure Cosmos DB Graph bulk executor sample hosted on GitHub. This application consists of a .NET solution that randomly generates vertex and edge objects and then executes bulk insertions to the specified graph database account. To get the application, run the `git clone` command below:

```
git clone https://github.com/Azure-Samples/azure-cosmosdb-graph-bulkexecutor-dotnet-getting-started.git
```

This repository contains the GraphBulkExecutor sample with the following files:

FILE	DESCRIPTION
`App.config`	This is where the application and database-specific parameters are specified. This file should be modified first to connect to the destination database and collections.
`Program.cs`	This file contains the logic behind creating the `DocumentClient` collection, handling the cleanups and sending the bulk executor requests.
`Util.cs`	This file contains a helper class that contains the logic behind generating test data, and checking if the database and collections exist.

In the `App.config` file, the following are the configuration values that can be provided:

SETTING	DESCRIPTION
`EndPointUrl`	This is **your .NET SDK endpoint** found in the Overview blade of your Azure Cosmos DB Gremlin API database account. This has the format of `https://your-graph-database-account.documents.azure.com:443/`
`AuthorizationKey`	This is the Primary or Secondary key listed under your Azure Cosmos DB account. Learn more about Securing Access to Azure Cosmos DB data
`DatabaseName`, `CollectionName`	These are the **target database and collection names**. When `ShouldCleanupOnStart` is set to `true` these values, along with `CollectionThroughput`, will be used to drop them and create a new database and collection. Similarly, if `ShouldCleanupOnFinish` is set to `true`, they will be used to delete the database as soon as the ingestion is over. Note that the target collection must be **an unlimited collection**.
`CollectionThroughput`	This is used to create a new collection if the `ShouldCleanupOnStart` option is set to `true`.
`ShouldCleanupOnStart`	This will drop the database account and collections before the program is run, and then create new ones with the `DatabaseName`, `CollectionName` and `CollectionThroughput` values.
`ShouldCleanupOnFinish`	This will drop the database account and collections with the specified `DatabaseName` and `CollectionName` after the program is run.
`NumberOfDocumentsToImport`	This will determine the number of test vertices and edges that will be generated in the sample. This number will apply to both vertices and edges.
`NumberOfBatches`	This will determine the number of test vertices and edges that will be generated in the sample. This number will apply to both vertices and edges.
`CollectionPartitionKey`	This will be used to create the test vertices and edges, where this property will be auto-assigned. This will also be used when re-creating the database and collections if the `ShouldCleanupOnStart` option is set to `true`.

Run the sample application

1. Add your specific database configuration parameters in `App.config`. This will be used to create a DocumentClient instance. If the database and container have not been created yet, they will be created automatically.
2. Run the application. This will call `BulkImportAsync` two times, one to import Vertices and one to import Edges. If any of the objects generates an error when they're inserted, they will be added to either `.\BadVertices.txt` or `.\BadEdges.txt`.
3. Evaluate the results by querying the graph database. If the `ShouldCleanupOnFinish` option is set to true, then the database will automatically be deleted.

Next steps

- To learn about NuGet package details and release notes of bulk executor .NET library, see bulk executor SDK details.
- Check out the Performance Tips to further optimize the usage of bulk executor.
- Review the BulkExecutor.Graph Reference article for more details about the classes and methods defined in this namespace.

How to use the execution profile step to evaluate your Gremlin queries

11/2/2020 • 7 minutes to read • Edit Online

APPLIES TO: Gremlin API

This article provides an overview of how to use the execution profile step for Azure Cosmos DB Gremlin API graph databases. This step provides relevant information for troubleshooting and query optimizations, and it is compatible with any Gremlin query that can be executed against a Cosmos DB Gremlin API account.

To use this step, simply append the `executionProfile()` function call at the end of your Gremlin query. **Your Gremlin query will be executed** and the result of the operation will return a JSON response object with the query execution profile.

For example:

```
// Basic traversal
g.V('mary').out()

// Basic traversal with execution profile call
g.V('mary').out().executionProfile()
```

After calling the `executionProfile()` step, the response will be a JSON object that includes the executed Gremlin step, the total time it took, and an array of the Cosmos DB runtime operators that the statement resulted in.

> **NOTE**
> This implementation for Execution Profile is not defined in the Apache Tinkerpop specification. It is specific to Azure Cosmos DB Gremlin API's implementation.

Response Example

The following is an annotated example of the output that will be returned:

> **NOTE**
> This example is annotated with comments that explain the general structure of the response. An actual executionProfile response won't contain any comments.

```
[
  {
    // The Gremlin statement that was executed.
    "gremlin": "g.V('mary').out().executionProfile()",

    // Amount of time in milliseconds that the entire operation took.
    "totalTime": 28,

    // An array containing metrics for each of the steps that were executed.
    // Each Gremlin step will translate to one or more of these steps.
    // This list is sorted in order of execution.
    "metrics": [
      {
```

```
    // This operation obtains a set of Vertex objects.
    // The metrics include: time, percentTime of total execution time, resultCount,
    // fanoutFactor, count, size (in bytes) and time.
    "name": "GetVertices",
    "time": 24,
    "annotations": {
      "percentTime": 85.71
    },
    "counts": {
      "resultCount": 2
    },
    "storeOps": [
      {
        "fanoutFactor": 1,
        "count": 2,
        "size": 696,
        "time": 0.4
      }
    ]
  },
  {
    // This operation obtains a set of Edge objects.
    // Depending on the query, these might be directly adjacent to a set of vertices,
    // or separate, in the case of an E() query.
    //
    // The metrics include: time, percentTime of total execution time, resultCount,
    // fanoutFactor, count, size (in bytes) and time.
    "name": "GetEdges",
    "time": 4,
    "annotations": {
      "percentTime": 14.29
    },
    "counts": {
      "resultCount": 1
    },
    "storeOps": [
      {
        "fanoutFactor": 1,
        "count": 1,
        "size": 419,
        "time": 0.67
      }
    ]
  },
  {
    // This operation obtains the vertices that a set of edges point at.
    // The metrics include: time, percentTime of total execution time and resultCount.
    "name": "GetNeighborVertices",
    "time": 0,
    "annotations": {
      "percentTime": 0
    },
    "counts": {
      "resultCount": 1
    }
  },
  {
    // This operation represents the serialization and preparation for a result from
    // the preceding graph operations. The metrics include: time, percentTime of total
    // execution time and resultCount.
    "name": "ProjectOperator",
    "time": 0,
    "annotations": {
      "percentTime": 0
    },
    "counts": {
      "resultCount": 1
    }
  }
```

```
      }
    ]
  }
]
```

> **NOTE**
>
> The executionProfile step will execute the Gremlin query. This includes the `addV` or `addE` steps, which will result in the creation and will commit the changes specified in the query. As a result, the Request Units generated by the Gremlin query will also be charged.

Execution profile response objects

The response of an executionProfile() function will yield a hierarchy of JSON objects with the following structure:

- **Gremlin operation object**: Represents the entire Gremlin operation that was executed. Contains the following properties.
  - `gremlin`: The explicit Gremlin statement that was executed.
  - `totalTime`: The time, in milliseconds, that the execution of the step incurred in.
  - `metrics`: An array that contains each of the Cosmos DB runtime operators that were executed to fulfill the query. This list is sorted in order of execution.
- **Cosmos DB runtime operators**: Represents each of the components of the entire Gremlin operation. This list is sorted in order of execution. Each object contains the following properties:
  - `name`: Name of the operator. This is the type of step that was evaluated and executed. Read more in the table below.
  - `time`: Amount of time, in milliseconds, that a given operator took.
  - `annotations`: Contains additional information, specific to the operator that was executed.
  - `annotations.percentTime`: Percentage of the total time that it took to execute the specific operator.
  - `counts`: Number of objects that were returned from the storage layer by this operator. This is contained in the `counts.resultCount` scalar value within.
  - `storeOps`: Represents a storage operation that can span one or multiple partitions.
  - `storeOps.fanoutFactor`: Represents the number of partitions that this specific storage operation accessed.
  - `storeOps.count`: Represents the number of results that this storage operation returned.
  - `storeOps.size`: Represents the size in bytes of the result of a given storage operation.

COSMOS DB GREMLIN RUNTIME OPERATOR	DESCRIPTION
`GetVertices`	This step obtains a predicated set of objects from the persistence layer.
`GetEdges`	This step obtains the edges that are adjacent to a set of vertices. This step can result in one or many storage operations.
`GetNeighborVertices`	This step obtains the vertices that are connected to a set of edges. The edges contain the partition keys and ID's of both their source and target vertices.
`Coalesce`	This step accounts for the evaluation of two operations whenever the `coalesce()` Gremlin step is executed.

COSMOS DB GREMLIN RUNTIME OPERATOR	DESCRIPTION
`CartesianProductOperator`	This step computes a cartesian product between two datasets. Usually executed whenever the predicates `to()` or `from()` are used.
`ConstantSourceOperator`	This step computes an expression to produce a constant value as a result.
`ProjectOperator`	This step prepares and serializes a response using the result of preceding operations.
`ProjectAggregation`	This step prepares and serializes a response for an aggregate operation.

> **NOTE**
> This list will continue to be updated as new operators are added.

Examples on how to analyze an execution profile response

The following are examples of common optimizations that can be spotted using the Execution Profile response:

- Blind fan-out query.
- Unfiltered query.

Blind fan-out query patterns

Assume the following execution profile response from a **partitioned graph**:

```
[
  {
    "gremlin": "g.V('tt0093640').executionProfile()",
    "totalTime": 46,
    "metrics": [
      {
        "name": "GetVertices",
        "time": 46,
        "annotations": {
          "percentTime": 100
        },
        "counts": {
          "resultCount": 1
        },
        "storeOps": [
          {
            "fanoutFactor": 5,
            "count": 1,
            "size": 589,
            "time": 75.61
          }
        ]
      },
      {
        "name": "ProjectOperator",
        "time": 0,
        "annotations": {
          "percentTime": 0
        },
        "counts": {
          "resultCount": 1
        }
      }
    ]
  }
]
```

The following conclusions can be made from it:

- The query is a single ID lookup, since the Gremlin statement follows the pattern `g.V('id')`.
- Judging from the `time` metric, the latency of this query seems to be high since it's more than 10ms for a single point-read operation.
- If we look into the `storeOps` object, we can see that the `fanoutFactor` is `5`, which means that 5 partitions were accessed by this operation.

As a conclusion of this analysis, we can determine that the first query is accessing more partitions than necessary. This can be addressed by specifying the partitioning key in the query as a predicate. This will lead to less latency and less cost per query. Learn more about graph partitioning. A more optimal query would be `g.V('tt0093640').has('partitionKey', 't1001')`.

Unfiltered query patterns

Compare the following two execution profile responses. For simplicity, these examples use a single partitioned graph.

This first query retrieves all vertices with the label `tweet` and then obtains their neighboring vertices:

```
[
  {
    "gremlin": "g.V().hasLabel('tweet').out().executionProfile()",
    "totalTime": 42,
    "metrics": [
      {
```

```
        "name": "GetVertices",
        "time": 31,
        "annotations": {
          "percentTime": 73.81
        },
        "counts": {
          "resultCount": 30
        },
        "storeOps": [
          {
            "fanoutFactor": 1,
            "count": 13,
            "size": 6819,
            "time": 1.02
          }
        ]
      },
      {
        "name": "GetEdges",
        "time": 6,
        "annotations": {
          "percentTime": 14.29
        },
        "counts": {
          "resultCount": 18
        },
        "storeOps": [
          {
            "fanoutFactor": 1,
            "count": 20,
            "size": 7950,
            "time": 1.98
          }
        ]
      },
      {
        "name": "GetNeighborVertices",
        "time": 5,
        "annotations": {
          "percentTime": 11.9
        },
        "counts": {
          "resultCount": 20
        },
        "storeOps": [
          {
            "fanoutFactor": 1,
            "count": 4,
            "size": 1070,
            "time": 1.19
          }
        ]
      },
      {
        "name": "ProjectOperator",
        "time": 0,
        "annotations": {
          "percentTime": 0
        },
        "counts": {
          "resultCount": 20
        }
      }
    ]
  }
]
```

Notice the profile of the same query, but now with an additional filter, `has('lang', 'en')`, before exploring the adjacent vertices:


```
[
  {
    "gremlin": "g.V().hasLabel('tweet').has('lang', 'en').out().executionProfile()",
    "totalTime": 14,
    "metrics": [
      {
        "name": "GetVertices",
        "time": 14,
        "annotations": {
          "percentTime": 58.33
        },
        "counts": {
          "resultCount": 11
        },
        "storeOps": [
          {
            "fanoutFactor": 1,
            "count": 11,
            "size": 4807,
            "time": 1.27
          }
        ]
      },
      {
        "name": "GetEdges",
        "time": 5,
        "annotations": {
          "percentTime": 20.83
        },
        "counts": {
          "resultCount": 18
        },
        "storeOps": [
          {
            "fanoutFactor": 1,
            "count": 18,
            "size": 7159,
            "time": 1.7
          }
        ]
      },
      {
        "name": "GetNeighborVertices",
        "time": 5,
        "annotations": {
          "percentTime": 20.83
        },
        "counts": {
          "resultCount": 18
        },
        "storeOps": [
          {
            "fanoutFactor": 1,
            "count": 4,
            "size": 1070,
            "time": 1.01
          }
        ]
      },
      {
        "name": "ProjectOperator",
        "time": 0,
        "annotations": {
          "percentTime": 0
```

```
          },
          "counts": {
            "resultCount": 18
          }
        }
      ]
    }
  ]
```

These two queries reached the same result, however, the first one will require more Request Units since it needed to iterate a larger initial dataset before querying the adjacent items. We can see indicators of this behavior when comparing the following parameters from both responses:

- The `metrics[0].time` value is higher in the first response, which indicates that this single step took longer to resolve.
- The `metrics[0].counts.resultsCount` value is higher as well in the first response, which indicates that the initial working dataset was larger.

Next steps

- Learn about the supported Gremlin features in Azure Cosmos DB.
- Learn more about the Gremlin API in Azure Cosmos DB.

Create a container in Azure Cosmos DB Gremlin API

11/2/2020 • 2 minutes to read • Edit Online

APPLIES TO: Gremlin API

This article explains the different ways to create a container in Azure Cosmos DB Gremlin API. It shows how to create a container using Azure portal, Azure CLI, PowerShell, or supported SDKs. This article demonstrates how to create a container, specify the partition key, and provision throughput.

This article explains the different ways to create a container in Azure Cosmos DB Gremlin API. If you are using a different API, see API for MongoDB, Cassandra API, Table API, and SQL API articles to create the container.

> **NOTE**
> When creating containers, make sure you don't create two containers with the same name but different casing. That's because some parts of the Azure platform are not case-sensitive, and this can result in confusion/collision of telemetry and actions on containers with such names.

Create using Azure portal

1. Sign in to the Azure portal.
2. Create a new Azure Cosmos account, or selectan existing account.
3. Open the **Data Explorer** pane, and select **New Graph**. Next, provide the following details:
   - Indicate whether you are creating a new database, or using an existing one.
   - Enter a Graph ID.
   - Select **Unlimited** storage capacity.
   - Enter a partition key for vertices.
   - Enter a throughput to be provisioned (for example, 1000 RUs).
   - Select **OK**.

Create using .NET SDK

If you encounter timeout exception when creating a collection, do a read operation to validate if the collection was created successfully. The read operation throws an exception until the collection create operation is successful. For the list of status codes supported by the create operation see the HTTP Status Codes for Azure Cosmos DB article.

```
// Create a container with a partition key and provision 1000 RU/s throughput.
DocumentCollection myCollection = new DocumentCollection();
myCollection.Id = "myContainerName";
myCollection.PartitionKey.Paths.Add("/myPartitionKey");

await client.CreateDocumentCollectionAsync(
    UriFactory.CreateDatabaseUri("myDatabaseName"),
    myCollection,
    new RequestOptions { OfferThroughput = 1000 });
```

Create using Azure CLI

Create a Gremlin graph with Azure CLI. For a listing of all Azure CLI samples across all Azure Cosmos DB APIs see, Azure CLI samples for Azure Cosmos DB.

Create using PowerShell

Create a Gremlin graph with PowerShell. For a listing of all PowerShell samples across all Azure Cosmos DB APIs see, PowerShell Samples

Next steps

- Partitioning in Azure Cosmos DB
- Request Units in Azure Cosmos DB
- Provision throughput on containers and databases
- Work with Azure Cosmos account

Provision database, container or autoscale throughput on Azure Cosmos DB Gremlin API resources

11/2/2020 • 2 minutes to read • Edit Online

APPLIES TO: Gremlin API

This article explains how to provision throughput in Azure Cosmos DB Gremlin API. You can provision standard(manual) or autoscale throughput on a container, or a database and share it among the containers within the database. You can provision throughput using Azure portal, Azure CLI, or Azure Cosmos DB SDKs.

If you are using a different API, see SQL API, Cassandra API, API for MongoDB articles to provision the throughput.

Azure portal

1. Sign in to the Azure portal.
2. Create a new Azure Cosmos account, or select an existing Azure Cosmos account.
3. Open the **Data Explorer** pane, and select **New Graph**. Next, provide the following details:
   - Indicate whether you are creating a new database or using an existing one. Select the **Provision database throughput** option if you want to provision throughput at the database level.
   - Enter a graph ID.
   - Enter a partition key value (for example, `/ItemID`).
   - Enter a throughput that you want to provision (for example, 1000 RUs).
   - Select **OK**.

.NET SDK

> **NOTE**
> Use the Cosmos SDKs for SQL API to provision throughput for all Azure Cosmos DB APIs, except Cassandra and MongoDB API.

Provision container level throughput

- .NET SDK V2
- .NET SDK V3

```
// Create a container with a partition key and provision throughput of 400 RU/s
DocumentCollection myCollection = new DocumentCollection();
myCollection.Id = "myContainerName";
myCollection.PartitionKey.Paths.Add("/myPartitionKey");

await client.CreateDocumentCollectionAsync(
    UriFactory.CreateDatabaseUri("myDatabaseName"),
    myCollection,
    new RequestOptions { OfferThroughput = 400 });
```

Provision database level throughput

- .NET SDK V2
- .NET SDK V3

```
//set the throughput for the database
RequestOptions options = new RequestOptions
{
    OfferThroughput = 500
};

//create the database
await client.CreateDatabaseIfNotExistsAsync(
    new Database {Id = databaseName},
    options);
```

Azure Resource Manager

Azure Resource Manager templates can be used to provision autoscale throughput on database or container-level resources for all Azure Cosmos DB APIs. See Azure Resource Manager templates for Azure Cosmos DB for samples.

Azure CLI

Azure CLI can be used to provision autoscale throughput on a database or container-level resources for all Azure Cosmos DB APIs. For samples see Azure CLI Samples for Azure Cosmos DB.

Azure PowerShell

Azure PowerShell can be used to provision autoscale throughput on a database or container-level resources for all Azure Cosmos DB APIs. For samples see Azure PowerShell samples for Azure Cosmos DB.

Next steps

See the following articles to learn about throughput provisioning in Azure Cosmos DB:

- Request units and throughput in Azure Cosmos DB

Find the request unit charge for operations executed in Azure Cosmos DB Gremlin API

11/2/2020 • 2 minutes to read • Edit Online

APPLIES TO: Gremlin API

Azure Cosmos DB supports many APIs, such as SQL, MongoDB, Cassandra, Gremlin, and Table. Each API has its own set of database operations. These operations range from simple point reads and writes to complex queries. Each database operation consumes system resources based on the complexity of the operation.

The cost of all database operations is normalized by Azure Cosmos DB and is expressed by Request Units (or RUs, for short). You can think of RUs as a performance currency abstracting the system resources such as CPU, IOPS, and memory that are required to perform the database operations supported by Azure Cosmos DB. No matter which API you use to interact with your Azure Cosmos container, costs are always measured by RUs. Whether the database operation is a write, point read, or query, costs are always measured in RUs. To learn more, see the request units and it's considerations article.

This article presents the different ways you can find the request unit (RU) consumption for any operation executed against a container in Azure Cosmos DB Gremlin API. If you are using a different API, see API for MongoDB, Cassandra API, SQL API, and Table API articles to find the RU/s charge.

Headers returned by the Gremlin API are mapped to custom status attributes, which currently are surfaced by the Gremlin .NET and Java SDK. The request charge is available under the `x-ms-request-charge` key. When you use the Gremlin API, you have multiple options for finding the RU consumption for an operation against an Azure Cosmos container.

Use the Azure portal

1. Sign in to the Azure portal.
2. Create a new Azure Cosmos account and feed it with data, or select an existing account that already contains data.
3. Go to the **Data Explorer** pane, and then select the container you want to work on.
4. Enter a valid query, and then select **Execute Gremlin Query**.
5. Select **Query Stats** to display the actual request charge for the request you executed.

Use the .NET SDK driver

When you use the Gremlin.NET SDK, status attributes are available under the `StatusAttributes` property of the `ResultSet<>` object:

```
ResultSet<dynamic> results = client.SubmitAsync<dynamic>("g.V().count()").Result;
double requestCharge = (double)results.StatusAttributes["x-ms-request-charge"];
```

For more information, see Quickstart: Build a .NET Framework or Core application by using an Azure Cosmos DB Gremlin API account.

Use the Java SDK driver

When you use the Gremlin Java SDK, you can retrieve status attributes by calling the `statusAttributes()` method on the `ResultSet` object:

```
ResultSet results = client.submit("g.V().count()");
Double requestCharge = (Double)results.statusAttributes().get().get("x-ms-request-charge");
```

For more information, see Quickstart: Create a graph database in Azure Cosmos DB by using the Java SDK.

Next steps

To learn about optimizing your RU consumption, see these articles:

- Request units and throughput in Azure Cosmos DB
- Optimize provisioned throughput cost in Azure Cosmos DB
- Optimize query cost in Azure Cosmos DB

Use Azure Cosmos DB resource tokens with the Gremlin SDK

12/4/2020 • 3 minutes to read • Edit Online

APPLIES TO: Gremlin API

This article explains how to use Azure Cosmos DB resource tokens to access the Graph database through the Gremlin SDK.

Create a resource token

The Apache TinkerPop Gremlin SDK doesn't have an API to use to create resource tokens. The term *resource token* is an Azure Cosmos DB concept. To create resource tokens, download the Azure Cosmos DB SDK. If your application needs to create resource tokens and use them to access the Graph database, it requires two separate SDKs.

The object model hierarchy above resource tokens is illustrated in the following outline:

- **Azure Cosmos DB account** - The top-level entity that has a DNS associated with it (for example, `contoso.gremlin.cosmos.azure.com`).
  - **Azure Cosmos DB database**
    - **User**
      - **Permission**
        - **Token** - A Permission object property that denotes what actions are allowed or denied.

A resource token uses the following format: `"type=resource&ver=1&sig=<base64 string>;<base64 string>;"`. This string is opaque for the clients and should be used as is, without modification or interpretation.

```
// Notice that document client is created against .NET SDK endpoint, rather than Gremlin.
DocumentClient client = new DocumentClient(
  new Uri("https://contoso.documents.azure.com:443/"),
  "<primary key>",
  new ConnectionPolicy
  {
    EnableEndpointDiscovery = false,
    ConnectionMode = ConnectionMode.Direct
  });

  // Read specific permission to obtain a token.
  // The token isn't returned during the ReadPermissionReedAsync() call.
  // The call succeeds only if database id, user id, and permission id already exist.
  // Note that <database id> is not a database name. It is a base64 string that represents the database
identifier, for example "KalVAA==".
  // Similar comment applies to <user id> and <permission id>.
  Permission permission = await client.ReadPermissionAsync(UriFactory.CreatePermissionUri("<database id>", "
<user id>", "<permission id>"));

  Console.WriteLine("Obtained token {0}", permission.Token);
}
```

Use a resource token

You can use resource tokens directly as a "password" property when you construct the GremlinServer class.

```
// The Gremlin application needs to be given a resource token. It can't discover the token on its own.
// You can obtain the token for a given permission by using the Azure Cosmos DB SDK, or you can pass it into
the application as a command line argument or configuration value.
string resourceToken = GetResourceToken();

// Configure the Gremlin server to use a resource token rather than a primary key.
GremlinServer server = new GremlinServer(
  "contoso.gremlin.cosmosdb.azure.com",
  port: 443,
  enableSsl: true,
  username: "/dbs/<database name>/colls/<collection name>",

  // The format of the token is "type=resource&ver=1&sig=<base64 string>;<base64 string>;".
  password: resourceToken);

  using (GremlinClient gremlinClient = new GremlinClient(server, new GraphSON2Reader(), new GraphSON2Writer(),
GremlinClient.GraphSON2MimeType))
  {
      await gremlinClient.SubmitAsync("g.V().limit(1)");
  }
```

The same approach works in all TinkerPop Gremlin SDKs.

```
Cluster.Builder builder = Cluster.build();

AuthProperties authenticationProperties = new AuthProperties();
authenticationProperties.with(AuthProperties.Property.USERNAME,
    String.format("/dbs/%s/colls/%s", "<database name>", "<collection name>"));

// The format of the token is "type=resource&ver=1&sig=<base64 string>;<base64 string>;".
authenticationProperties.with(AuthProperties.Property.PASSWORD, resourceToken);

builder.authProperties(authenticationProperties);
```

Limit

With a single Gremlin account, you can issue an unlimited number of tokens. However, you can use only up to 100 tokens concurrently within 1 hour. If an application exceeds the token limit per hour, an authentication request is denied, and you receive the following error message: "Exceeded allowed resource token limit of 100 that can be used concurrently." It doesn't work to close active connections that use specific tokens to free up slots for new tokens. The Azure Cosmos DB Gremlin database engine keeps track of unique tokens during the hour immediately prior to the authentication request.

Permission

A common error that applications encounter while they're using resource tokens is, "Insufficient permissions provided in the authorization header for the corresponding request. Please retry with another authorization header." This error is returned when a Gremlin traversal attempts to write an edge or a vertex but the resource token grants *Read* permissions only. Inspect your traversal to see whether it contains any of the following steps: *.addV()*, *.addE()*, *.drop()*, or *.property()*.

Next steps

- Azure role-based access control (Azure RBAC) in Azure Cosmos DB
- Learn how to secure access to data in Azure Cosmos DB

Regional endpoints for Azure Cosmos DB Graph account

12/4/2020 • 3 minutes to read • Edit Online

APPLIES TO: Gremlin API

Azure Cosmos DB Graph database is globally distributed so applications can use multiple read endpoints. Applications that need write access in multiple locations should enable multi-region writes capability.

Reasons to choose more than one region:

1. **Horizontal read scalability** - as application load increases it may be prudent to route read traffic to different Azure regions.
2. **Lower latency** - you can reduce network latency overhead of each traversal by routing read and write traffic to the nearest Azure region.

Data residency requirement is achieved by setting Azure Resource Manager policy on Cosmos DB account. Customer can limit regions into which Cosmos DB replicates data.

Traffic routing

Cosmos DB Graph database engine is running in multiple regions, each of which contains multiple clusters. Each cluster has hundreds of machines. Cosmos DB Graph account DNS CNAME *accountname.gremlin.cosmos.azure.com* resolves to DNS A record of a cluster. A single IP address of a load-balancer hides internal cluster topology.

A regional DNS CNAME record is created for every region of Cosmos DB Graph account. Format of regional endpoint is *accountname-region.gremlin.cosmos.azure.com*. Region segment of regional endpoint is obtained by removing all spaces from Azure region name. For example, `"East US 2"` region for `"contoso"` global database account would have a DNS CNAME *contoso-eastus2.gremlin.cosmos.azure.com*

TinkerPop Gremlin client is designed to work with a single server. Application can use global writable DNS CNAME for read and write traffic. Region-aware applications should use regional endpoint for read traffic. Use regional endpoint for write traffic only if specific region is configured to accept writes.

> **NOTE**
>
> Cosmos DB Graph engine can accept write operation in read region by proxying traffic to write region. It is not recommended to send writes into read-only region as it increases traversal latency and is subject to restrictions in the future.

Global database account CNAME always points to a valid write region. During server-side failover of write region, Cosmos DB updates global database account CNAME to point to new region. If application can't handle traffic rerouting after failover, it should use global database account DNS CNAME.

> **NOTE**
>
> Cosmos DB does not route traffic based on geographic proximity of the caller. It is up to each application to select the right region according to unique application needs.

Portal endpoint discovery

The easiest way to get the list of regions for Azure Cosmos DB Graph account is overview blade in Azure portal. It will work for applications that do not change regions often, or have a way to update the list via application configuration.

Example below demonstrates general principles of accessing regional Gremlin endpoint. Application should consider number of regions to send the traffic to and number of corresponding Gremlin clients to instantiate.

```
// Example value: Central US, West US and UK West. This can be found in the overview blade of you Azure Cosmos
DB Gremlin Account.
// Look for Write Locations in the overview blade. You can click to copy and paste.
string[] gremlinAccountRegions = new string[] {"Central US", "West US" ,"UK West"};
string gremlinAccountName = "PUT-COSMOSDB-ACCOUNT-NAME-HERE";
string gremlinAccountKey = "PUT-ACCOUNT-KEY-HERE";
string databaseName = "PUT-DATABASE-NAME-HERE";
string graphName = "PUT-GRAPH-NAME-HERE";

foreach (string gremlinAccountRegion in gremlinAccountRegions)
{
  // Convert preferred read location to the form "[acountname]-[region].gremlin.cosmos.azure.com".
  string regionalGremlinEndPoint = $"{gremlinAccountName}-{gremlinAccountRegion.ToLowerInvariant().Replace(" ",
string.Empty)}.gremlin.cosmos.azure.com";

  GremlinServer regionalGremlinServer = new GremlinServer(
    hostname: regionalGremlinEndPoint,
    port: 443,
    enableSsl: true,
    username: "/dbs/" + databaseName + "/colls/" + graphName,
    password: gremlinAccountKey);

  GremlinClient regionalGremlinClient = new GremlinClient(
    gremlinServer: regionalGremlinServer,
    graphSONReader: new GraphSON2Reader(),
    graphSONWriter: new GraphSON2Writer(),
    mimeType: GremlinClient.GraphSON2MimeType);
}
```

SDK endpoint discovery

Application can use Azure Cosmos DB SDK to discover read and write locations for Graph account. These locations can change at any time through manual reconfiguration on the server side or automatic failover.

TinkerPop Gremlin SDK doesn't have an API to discover Cosmos DB Graph database account regions. Applications that need runtime endpoint discovery need to host 2 separate SDKs in the process space.

```
// Depending on the version and the language of the SDK (.NET vs Java vs Python)
// the API to get readLocations and writeLocations may vary.
IDocumentClient documentClient = new DocumentClient(
    new Uri(cosmosUrl),
    cosmosPrimaryKey,
    connectionPolicy,
    consistencyLevel);

DatabaseAccount databaseAccount = await cosmosClient.GetDatabaseAccountAsync();

IEnumerable<DatabaseAccountLocation> writeLocations = databaseAccount.WritableLocations;
IEnumerable<DatabaseAccountLocation> readLocations = databaseAccount.ReadableLocations;

// Pick write or read locations to construct regional endpoints for.
foreach (string location in readLocations)
{
  // Convert preferred read location to the form "[acountname]-[region].gremlin.cosmos.azure.com".
  string regionalGremlinEndPoint = location
    .Replace("http:\/\/", string.Empty)
    .Replace("documents.azure.com:443/", "gremlin.cosmos.azure.com");

  // Use code from the previous sample to instantiate Gremlin client.
}
```

Next steps

- How to manage database accounts control in Azure Cosmos DB
- High availability in Azure Cosmos DB
- Global distribution with Azure Cosmos DB - under the hood
- Azure CLI Samples for Azure Cosmos DB

System document properties

11/2/2020 • 2 minutes to read • Edit Online

APPLIES TO: Gremlin API

Azure Cosmos DB has system properties such as `_ts`, `_self`, `_attachments`, `_rid`, and `_etag` on every document. Additionally, Gremlin engine adds `inVPartition` and `outVPartition` properties on edges. By default, these properties are available for traversal. However, it's possible to include specific properties, or all of them, in Gremlin traversal.

```
g.withStrategies(ProjectionStrategy.build().IncludeSystemProperties('_ts').create())
```

E-Tag

This property is used for optimistic concurrency control. If application needs to break operation into a few separate traversals, it can use eTag property to avoid data loss in concurrent writes.

```
g.withStrategies(ProjectionStrategy.build().IncludeSystemProperties('_etag').create()).V('1').has('_etag',
'"00000100-0000-0800-0000-5d03edac0000"').property('test', '1')
```

Time-to-live (TTL)

If collection has document expiration enabled and documents have `ttl` property set on them, then this property will be available in Gremlin traversal as a regular vertex or edge property. `ProjectionStrategy` isn't necessary to enable time-to-live property exposure.

Vertex created with the traversal below will be automatically deleted in `123 seconds`.

```
g.addV('vertex-one').property('ttl', 123)
```

Next steps

- Cosmos DB Optimistic Concurrency
- Time to Live (TTL) in Azure Cosmos DB

Visualize graph data stored in Azure Cosmos DB Gremlin API with data visualization solutions

11/2/2020 • 2 minutes to read • Edit Online

APPLIES TO: Gremlin API

You can visualize data stored in Azure Cosmos DB Gremlin API by using various data visualization solutions. The following solutions are recommended by the Apache Tinkerpop community for graph data visualization.

Linkurious Enterprise

Linkurious Enterprise uses graph technology and data visualization to turn complex datasets into interactive visual networks. The platform connects to your data sources and enables investigators to seamlessly navigate across billions of entities and relationships. The result is a new ability to detect suspicious relationships without juggling with queries or tables.

The interactive interface of Linkurious Enterprise offers an easy way to investigate complex data. You can search for specific entities, expand connections to uncover hidden relationships, and apply layouts of your choice to untangle complex networks. Linkurious Enterprise is now compatible with Azure Cosmos DB Gremlin API. It's suitable for end-to-end graph visualization scenarios and supports read and write capabilities from the user interface. You can request a demo of Linkurious with Azure Cosmos DB

Cambridge Intelligence

Cambridge Intelligence

Cambridge Intelligence's graph visualization toolkits now support Azure Cosmos DB. The following two visualization toolkits are supported by Azure Cosmos DB:

- KeyLines for JavaScript developers
- Re-Graph for React developers

These toolkits let you design high-performance graph visualization and analysis applications for your use case. They harness powerful Web Graphics Library(WebGL) rendering and carefully crafted code to give users a fast and insightful visualization experience. These tools are compatible with any browser, device, server or database, and come with step-by-step tutorials, fully documented APIs, and interactive demos.

Next steps

- Try the toolkits
- KeyLines technology overview
- Re-Graph technology overview
- Graph visualization use cases

Manage Azure Cosmos DB Gremlin API resources using Azure Resource Manager templates

12/4/2020 • 5 minutes to read • Edit Online

APPLIES TO: Gremlin API

In this article, you learn how to use Azure Resource Manager templates to help deploy and manage your Azure Cosmos DB accounts, databases, and graphs.

This article has examples for Gremlin API accounts only, to find examples for other API type accounts see: use Azure Resource Manager templates with Azure Cosmos DB's API for Cassandra, SQL, MongoDB, Table articles.

> **IMPORTANT**
>
> - Account names are limited to 44 characters, all lowercase.
> - To change the throughput values, redeploy the template with updated RU/s.
> - When you add or remove locations to an Azure Cosmos account, you can't simultaneously modify other properties. These operations must be done separately.

To create any of the Azure Cosmos DB resources below, copy the following example template into a new json file. You can optionally create a parameters json file to use when deploying multiple instances of the same resource with different names and values. There are many ways to deploy Azure Resource Manager templates including, Azure portal, Azure CLI, Azure PowerShell and GitHub.

Azure Cosmos DB account for Gremlin with autoscale provisioned throughput

This template will create an Azure Cosmos account for Gremlin API with a database and graph with autoscale throughput. This template is also available for one-click deploy from Azure Quickstart Templates Gallery.

Deploy to Azure

```
{
    "$schema": "https://schema.management.azure.com/schemas/2019-04-01/deploymentTemplate.json#",
    "contentVersion": "1.0.0.0",
    "parameters": {
        "accountName": {
            "type": "string",
            "defaultValue": "[concat('gremlin-', uniqueString(resourceGroup().id))]",
            "metadata": {
                "description": "Cosmos DB account name"
            }
        },
        "location": {
            "type": "string",
            "defaultValue": "[resourceGroup().location]",
            "metadata": {
                "description": "Location for the Cosmos DB account."
            }
        },
        "primaryRegion": {
            "type": "string",
            "metadata": {
```

```
                "description": "The primary replica region for the Cosmos DB account."
            }
        },
        "secondaryRegion": {
            "type": "string",
            "metadata": {
                "description": "The secondary replica region for the Cosmos DB account."
            }
        },
        "defaultConsistencyLevel": {
            "type": "string",
            "defaultValue": "Session",
            "allowedValues": [
                "Eventual",
                "ConsistentPrefix",
                "Session",
                "BoundedStaleness",
                "Strong"
            ],
            "metadata": {
                "description": "The default consistency level of the Cosmos DB account."
            }
        },
        "maxStalenessPrefix": {
            "type": "int",
            "defaultValue": 100000,
            "minValue": 10,
            "maxValue": 2147483647,
            "metadata": {
                "description": "Max stale requests. Required for BoundedStaleness. Valid ranges, Single
Region: 10 to 1000000. Multi Region: 100000 to 1000000."
            }
        },
        "maxIntervalInSeconds": {
            "type": "int",
            "defaultValue": 300,
            "minValue": 5,
            "maxValue": 86400,
            "metadata": {
                "description": "Max lag time (seconds). Required for BoundedStaleness. Valid ranges, Single
Region: 5 to 84600. Multi Region: 300 to 86400."
            }
        },
        "automaticFailover": {
            "type": "bool",
            "defaultValue": true,
            "allowedValues": [
                true,
                false
            ],
            "metadata": {
                "description": "Enable automatic failover for regions"
            }
        },
        "databaseName": {
            "type": "string",
            "metadata": {
                "description": "The name for the Gremlin database"
            }
        },
        "graphName": {
            "type": "string",
            "metadata": {
                "description": "The name for the Gremlin graph"
            }
        },
        "autoscaleMaxThroughput": {
            "type": "int",
            "defaultValue": 4000,
```

```
            "minValue": 4000,
            "maxValue": 1000000,
            "metadata": {
                "description": "Maximum autoscale throughput for the graph"
            }
        }
    },
    "variables": {
        "accountName": "[toLower(parameters('accountName'))]",
        "consistencyPolicy": {
            "Eventual": {
                "defaultConsistencyLevel": "Eventual"
            },
            "ConsistentPrefix": {
                "defaultConsistencyLevel": "ConsistentPrefix"
            },
            "Session": {
                "defaultConsistencyLevel": "Session"
            },
            "BoundedStaleness": {
                "defaultConsistencyLevel": "BoundedStaleness",
                "maxStalenessPrefix": "[parameters('maxStalenessPrefix')]",
                "maxIntervalInSeconds": "[parameters('maxIntervalInSeconds')]"
            },
            "Strong": {
                "defaultConsistencyLevel": "Strong"
            }
        },
        "locations": [
            {
                "locationName": "[parameters('primaryRegion')]",
                "failoverPriority": 0,
                "isZoneRedundant": false
            },
            {
                "locationName": "[parameters('secondaryRegion')]",
                "failoverPriority": 1,
                "isZoneRedundant": false
            }
        ]
    },
    "resources": [
        {
            "type": "Microsoft.DocumentDB/databaseAccounts",
            "name": "[variables('accountName')]",
            "apiVersion": "2020-04-01",
            "location": "[parameters('location')]",
            "kind": "GlobalDocumentDB",
            "properties": {
                "capabilities": [
                    {
                        "name": "EnableGremlin"
                    }
                ],
                "consistencyPolicy": "[variables('consistencyPolicy')
[parameters('defaultConsistencyLevel')]]",
                "locations": "[variables('locations')]",
                "databaseAccountOfferType": "Standard",
                "enableAutomaticFailover": "[parameters('automaticFailover')]"
            }
        },
        {
            "type": "Microsoft.DocumentDB/databaseAccounts/gremlinDatabases",
            "name": "[concat(variables('accountName'), '/', parameters('databaseName'))]",
            "apiVersion": "2020-04-01",
            "dependsOn": [
                "[resourceId('Microsoft.DocumentDB/databaseAccounts/', variables('accountName'))]"
            ],
            "properties": {
```

```
                "resource": {
                    "id": "[parameters('databaseName')]"
                }
            }
        },
        {
            "type": "Microsoft.DocumentDb/databaseAccounts/gremlinDatabases/graphs",
            "name": "[concat(variables('accountName'), '/', parameters('databaseName'), '/',
parameters('graphName'))]",
            "apiVersion": "2020-04-01",
            "dependsOn": [
                "[resourceId('Microsoft.DocumentDB/databaseAccounts/gremlinDatabases',
variables('accountName'),  parameters('databaseName'))]"
            ],
            "properties": {
                "resource": {
                    "id": "[parameters('graphName')]",
                    "indexingPolicy": {
                        "indexingMode": "consistent",
                        "includedPaths": [
                            {
                                "path": "/*"
                            }
                        ],
                        "excludedPaths": [
                            {
                                "path": "/myPathToNotIndex/*"
                            }
                        ]
                    },
                    "partitionKey": {
                        "paths": [
                            "/myPartitionKey"
                        ],
                        "kind": "Hash"
                    }
                },
                "options": {
                    "autoscaleSettings": {
                        "maxThroughput": "[parameters('autoscaleMaxThroughput')]"
                    }
                }
            }
        }
    ]
}
```

Azure Cosmos DB account for Gremlin with standard provisioned throughput

This template will create an Azure Cosmos account for Gremlin API with a database and graph with standard (manual) throughput. This template is also available for one-click deploy from Azure Quickstart Templates Gallery.

Deploy to Azure

```
{
    "$schema": "https://schema.management.azure.com/schemas/2019-04-01/deploymentTemplate.json#",
    "contentVersion": "1.0.0.0",
    "parameters": {
        "accountName": {
            "type": "string",
            "defaultValue": "[uniqueString(resourceGroup().id)]",
            "metadata": {
                "description": "Cosmos DB account name"
```

```
            }
        },
        "location": {
            "type": "string",
            "defaultValue": "[resourceGroup().location]",
            "metadata": {
                "description": "Location for the Cosmos DB account."
            }
        },
        "primaryRegion": {
            "type": "string",
            "metadata": {
                "description": "The primary replica region for the Cosmos DB account."
            }
        },
        "secondaryRegion": {
            "type": "string",
            "metadata": {
                "description": "The secondary replica region for the Cosmos DB account."
            }
        },
        "defaultConsistencyLevel": {
            "type": "string",
            "defaultValue": "Session",
            "allowedValues": [
                "Eventual",
                "ConsistentPrefix",
                "Session",
                "BoundedStaleness",
                "Strong"
            ],
            "metadata": {
                "description": "The default consistency level of the Cosmos DB account."
            }
        },
        "maxStalenessPrefix": {
            "type": "int",
            "defaultValue": 100000,
            "minValue": 10,
            "maxValue": 2147483647,
            "metadata": {
                "description": "Max stale requests. Required for BoundedStaleness. Valid ranges, Single
Region: 10 to 1000000. Multi Region: 100000 to 1000000."
            }
        },
        "maxIntervalInSeconds": {
            "type": "int",
            "defaultValue": 300,
            "minValue": 5,
            "maxValue": 86400,
            "metadata": {
                "description": "Max lag time (seconds). Required for BoundedStaleness. Valid ranges, Single
Region: 5 to 84600. Multi Region: 300 to 86400."
            }
        },
        "automaticFailover": {
            "type": "bool",
            "defaultValue": true,
            "allowedValues": [
                true,
                false
            ],
            "metadata": {
                "description": "Enable automatic failover for regions"
            }
        },
        "databaseName": {
            "type": "string",
            "defaultValue": "database1",
```

```
            "metadata": {
                "description": "The name for the Gremlin database"
            }
        },
        "graphName": {
            "type": "string",
            "defaultValue": "graph1",
            "metadata": {
                "description": "The name for the Gremlin graph"
            }
        },
        "throughput": {
            "type": "int",
            "defaultValue": 400,
            "minValue": 400,
            "maxValue": 1000000,
            "metadata": {
                "description": "Throughput for the Gremlin graph"
            }
        }
    },
    "variables": {
        "accountName": "[toLower(parameters('accountName'))]",
        "consistencyPolicy": {
            "Eventual": {
                "defaultConsistencyLevel": "Eventual"
            },
            "ConsistentPrefix": {
                "defaultConsistencyLevel": "ConsistentPrefix"
            },
            "Session": {
                "defaultConsistencyLevel": "Session"
            },
            "BoundedStaleness": {
                "defaultConsistencyLevel": "BoundedStaleness",
                "maxStalenessPrefix": "[parameters('maxStalenessPrefix')]",
                "maxIntervalInSeconds": "[parameters('maxIntervalInSeconds')]"
            },
            "Strong": {
                "defaultConsistencyLevel": "Strong"
            }
        },
        "locations": [
            {
                "locationName": "[parameters('primaryRegion')]",
                "failoverPriority": 0,
                "isZoneRedundant": false
            },
            {
                "locationName": "[parameters('secondaryRegion')]",
                "failoverPriority": 1,
                "isZoneRedundant": false
            }
        ]
    },
    "resources": [
        {
            "type": "Microsoft.DocumentDB/databaseAccounts",
            "name": "[variables('accountName')]",
            "apiVersion": "2020-03-01",
            "location": "[parameters('location')]",
            "kind": "GlobalDocumentDB",
            "properties": {
                "capabilities": [
                    {
                        "name": "EnableGremlin"
                    }
                ],
                "consistencyPolicy": "[variables('consistencyPolicy')
```

```
                "consistencyPolicy": "[variables('consistencyPolicy')]",
[parameters('defaultConsistencyLevel')]]",
                "locations": "[variables('locations')]",
                "databaseAccountOfferType": "Standard",
                "enableAutomaticFailover": "[parameters('automaticFailover')]"
            }
        },
        {
            "type": "Microsoft.DocumentDB/databaseAccounts/gremlinDatabases",
            "name": "[concat(variables('accountName'), '/', parameters('databaseName'))]",
            "apiVersion": "2020-03-01",
            "dependsOn": [
                "[resourceId('Microsoft.DocumentDB/databaseAccounts/', variables('accountName'))]"
            ],
            "properties": {
                "resource": {
                    "id": "[parameters('databaseName')]"
                }
            }
        },
        {
            "type": "Microsoft.DocumentDb/databaseAccounts/gremlinDatabases/graphs",
            "name": "[concat(variables('accountName'), '/', parameters('databaseName'), '/',
parameters('graphName'))]",
            "apiVersion": "2020-03-01",
            "dependsOn": [
                "[resourceId('Microsoft.DocumentDB/databaseAccounts/gremlinDatabases',
variables('accountName'),  parameters('databaseName'))]"
            ],
            "properties": {
                "resource": {
                    "id": "[parameters('graphName')]",
                    "indexingPolicy": {
                        "indexingMode": "consistent",
                        "includedPaths": [
                            {
                                "path": "/*"
                            }
                        ],
                        "excludedPaths": [
                            {
                                "path": "/myPathToNotIndex/*"
                            }
                        ]
                    },
                    "partitionKey": {
                        "paths": [
                            "/myPartitionKey"
                        ],
                        "kind": "Hash"
                    },
                    "options": {
                        "throughput": "[parameters('throughput')]"
                    }
                }
            }
        }
    ]
}
```

Next steps

Here are some additional resources:

- Azure Resource Manager documentation
- Azure Cosmos DB resource provider schema

- Azure Cosmos DB Quickstart templates
- Troubleshoot common Azure Resource Manager deployment errors

Azure PowerShell samples for Azure Cosmos DB Gremlin API

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: Gremlin API

The following table includes links to commonly used Azure PowerShell scripts for Azure Cosmos DB. Use the links on the right to navigate to API specific samples. Common samples are the same across all APIs. Reference pages for all Azure Cosmos DB PowerShell cmdlets are available in the Azure PowerShell Reference. Please check for updates to `Az.CosmosDB` regularly. You can also fork these PowerShell samples for Cosmos DB from our GitHub repository, Cosmos DB PowerShell Samples on GitHub.

Common Samples

TASK	DESCRIPTION
Update an account	Update a Cosmos DB account's default consistency level.
Update an account's regions	Update a Cosmos DB account's regions.
Change failover priority or trigger failover	Change the regional failover priority of an Azure Cosmos account or trigger a manual failover.
Account keys or connection strings	Get primary and secondary keys, connection strings or regenerate an account key of an Azure Cosmos DB account.
Create a Cosmos Account with IP Firewall	Create an Azure Cosmos DB account with IP Firewall enabled.

Gremlin API Samples

TASK	DESCRIPTION
Create an account, database and graph	Creates an Azure Cosmos account, database and graph.
Create an account, database and graph with autoscale	Creates an Azure Cosmos account, database and graph with autoscale.
List or get databases or graphs	List or get database or graph.
Throughput operations	Throughput operations for a database or graph including get, update and migrate between autoscale and standard throughput.
Lock resources from deletion	Prevent resources from being deleted with resource locks.

Azure CLI samples for Azure Cosmos DB Gremlin API

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: Gremlin API

The following table includes links to sample Azure CLI scripts for Azure Cosmos DB. Use the links on the right to navigate to API specific samples. Common samples are the same across all APIs. Reference pages for all Azure Cosmos DB CLI commands are available in the Azure CLI Reference. Azure Cosmos DB CLI script samples can also be found in the Azure Cosmos DB CLI GitHub Repository.

These samples require Azure CLI version 2.12.1 or later. Run `az --version` to find the version. If you need to install or upgrade, see Install Azure CLI

Common Samples

These samples apply to all Azure Cosmos DB APIs

TASK	DESCRIPTION
Add or failover regions	Add a region, change failover priority, trigger a manual failover.
Account keys and connection strings	List account keys, read-only keys, regenerate keys and list connection strings.
Secure with IP firewall	Create a Cosmos account with IP firewall configured.
Secure new account with service endpoints	Create a Cosmos account and secure with service-endpoints.
Secure existing account with service endpoints	Update a Cosmos account to secure with service-endpoints when the subnet is eventually configured.

Gremlin API Samples

TASK	DESCRIPTION
Create an Azure Cosmos account, database and graph	Creates an Azure Cosmos DB account, database, and graph for Gremlin API.
Create an Azure Cosmos account, database and graph with autoscale	Creates an Azure Cosmos DB account, database, and graph with autoscale for Gremlin API.
Throughput operations	Read, update and migrate between autoscale and standard throughput on a database and graph.
Lock resources from deletion	Prevent resources from being deleted with resource locks.

Azure Cosmos DB Gremlin server response headers

12/4/2020 • 6 minutes to read • Edit Online

APPLIES TO: Gremlin API

This article covers headers that Cosmos DB Gremlin server returns to the caller upon request execution. These headers are useful for troubleshooting request performance, building application that integrates natively with Cosmos DB service and simplifying customer support.

Keep in mind that taking dependency on these headers you are limiting portability of your application to other Gremlin implementations. In return, you are gaining tighter integration with Cosmos DB Gremlin. These headers are not a TinkerPop standard.

Headers

HEADER	TYPE	SAMPLE VALUE	WHEN INCLUDED	EXPLANATION
x-ms-request-charge	double	11.3243	Success and Failure	Amount of collection or database throughput consumed in request units (RU/s or RUs) for a partial response message. This header is present in every continuation for requests that have multiple chunks. It reflects the charge of a particular response chunk. Only for requests that consist of a single response chunk this header matches total cost of traversal. However, for majority of complex traversals this value represents a partial cost.

HEADER	TYPE	SAMPLE VALUE	WHEN INCLUDED	EXPLANATION
x-ms-total-request-charge	double	423.987	Success and Failure	Amount of collection or database throughput consumed in request units (RU/s or RUs) for entire request. This header is present in every continuation for requests that have multiple chunks. It indicates cumulative charge since the beginning of request. Value of this header in the last chunk indicates complete request charge.
x-ms-server-time-ms	double	13.75	Success and Failure	This header is included for latency troubleshooting purposes. It indicates the amount of time, in milliseconds, that Cosmos DB Gremlin server took to execute and produce a partial response message. Using value of this header and comparing it to overall request latency applications can calculate network latency overhead.
x-ms-total-server-time-ms	double	130.512	Success and Failure	Total time, in milliseconds, that Cosmos DB Gremlin server took to execute entire traversal. This header is included in every partial response. It represents cumulative execution time since the start of request. The last response indicates total execution time. This header is useful to differentiate between client and server as a source of latency. You can compare traversal execution time on the client to the value of this header.

HEADER	TYPE	SAMPLE VALUE	WHEN INCLUDED	EXPLANATION
x-ms-status-code	long	200	Success and Failure	Header indicates internal reason for request completion or termination. Application is advised to look at the value of this header and take corrective action.
x-ms-substatus-code	long	1003	Failure Only	Cosmos DB is a multi-model database that is built on top of unified storage layer. This header contains additional insights about the failure reason when failure occurs within lower layers of high availability stack. Application is advised to store this header and use it when contacting Cosmos DB customer support. Value of this header is useful for Cosmos DB engineer for quick troubleshooting.
x-ms-retry-after-ms	string (TimeSpan)	"00:00:03.9500000"	Failure Only	This header is a string representation of a .NET TimeSpan type. This value will only be included in requests failed due provisioned throughput exhaustion. Application should resubmit traversal again after instructed period of time.

HEADER	TYPE	SAMPLE VALUE	WHEN INCLUDED	EXPLANATION
x-ms-activity-id	string (Guid)	"A9218E01-3A3A-4716-9636-5BD86B056613"	Success and Failure	Header contains a unique server-side identifier of a request. Each request is assigned a unique identifier by the server for tracking purposes. Applications should log activity identifiers returned by the server for requests that customers may want to contact customer support about. Cosmos DB support personnel can find specific requests by these identifiers in Cosmos DB service telemetry.

Status codes

Most common status codes returned by the server are listed below.

STATUS	EXPLANATION
401	Error message `"Unauthorized: Invalid credentials provided"` is returned when authentication password doesn't match Cosmos DB account key. Navigate to your Cosmos DB Gremlin account in the Azure portal and confirm that the key is correct.
404	Concurrent operations that attempt to delete and update the same edge or vertex simultaneously. Error message `"Owner resource does not exist"` indicates that specified database or collection is incorrect in connection parameters in `/dbs/<database name>/colls/<collection or graph name>` format.
408	`"Server timeout"` indicates that traversal took more than **30 seconds** and was canceled by the server. Optimize your traversals to run quickly by filtering vertices or edges on every hop of traversal to narrow down search scope.
409	`"Conflicting request to resource has been attempted. Retry to avoid conflicts."` This usually happens when vertex or an edge with an identifier already exists in the graph.

STATUS	EXPLANATION
412	Status code is complemented with error message `"PreconditionFailedException": One of the specified pre-condition is not met`. This error is indicative of an optimistic concurrency control violation between reading an edge or vertex and writing it back to the store after modification. Most common situations when this error occurs is property modification, for example `g.V('identifier').property('name','value')`. Gremlin engine would read the vertex, modify it, and write it back. If there is another traversal running in parallel trying to write the same vertex or an edge, one of them will receive this error. Application should submit traversal to the server again.
429	Request was throttled and should be retried after value in **x-ms-retry-after-ms**
500	Error message that contains `"NotFoundException: Entity with the specified id does not exist in the system."` indicates that a database and/or collection was re-created with the same name. This error will disappear within 5 minutes as change propagates and invalidates caches in different Cosmos DB components. To avoid this issue, use unique database and collection names every time.
1000	This status code is returned when server successfully parsed a message but wasn't able to execute. It usually indicates a problem with the query.
1001	This code is returned when server completes traversal execution but fails to serialize response back to the client. This error can happen when traversal generates complex result, that is too large or does not conform to TinkerPop protocol specification. Application should simplify the traversal when it encounters this error.
1003	`"Query exceeded memory limit. Bytes Consumed: XXX, Max: YYY"` is returned when traversal exceeds allowed memory limit. Memory limit is **2 GB** per traversal.
1004	This status code indicates malformed graph request. Request can be malformed when it fails deserialization, non-value type is being deserialized as value type or unsupported gremlin operation requested. Application should not retry the request because it will not be successful.
1007	Usually this status code is returned with error message `"Could not process request. Underlying connection has been closed."`. This situation can happen if client driver attempts to use a connection that is being closed by the server. Application should retry the traversal on a different connection.

STATUS	EXPLANATION
1008	Cosmos DB Gremlin server can terminate connections to rebalance traffic in the cluster. Client drivers should handle this situation and use only live connections to send requests to the server. Occasionally client drivers may not detect that connection was closed. When application encounters an error, `"Connection is too busy. Please retry after sometime or open more connections."` it should retry traversal on a different connection.

Samples

A sample client application based on Gremlin.Net that reads one status attribute:

```
// Following example reads a status code and total request charge from server response attributes.
// Variable "server" is assumed to be assigned to an instance of a GremlinServer that is connected to Cosmos
DB account.
using (GremlinClient client = new GremlinClient(server, new GraphSON2Reader(), new GraphSON2Writer(),
GremlinClient.GraphSON2MimeType))
{
  ResultSet<dynamic> responseResultSet = await GremlinClientExtensions.SubmitAsync<dynamic>(client,
requestScript: "g.V().count()");
  long statusCode = (long)responseResultSet.StatusAttributes["x-ms-status-code"];
  double totalRequestCharge = (double)responseResultSet.StatusAttributes["x-ms-total-request-charge"];

  // Status code and request charge are logged into application telemetry.
}
```

An example that demonstrates how to read status attribute from Gremlin java client:

```
try {
  ResultSet resultSet = this.client.submit("g.addV().property('id', '13')");
  List<Result> results = resultSet.all().get();

  // Process and consume results

} catch (ResponseException re) {
  // Check for known errors that need to be retried or skipped
  if (re.getStatusAttributes().isPresent()) {
    Map<String, Object> attributes = re.getStatusAttributes().get();
    int statusCode = (int) attributes.getOrDefault("x-ms-status-code", -1);

    // Now we can check for specific conditions
    if (statusCode == 409) {
        // Handle conflicting writes
      }
    }

    // Check if we need to delay retry
    if (attributes.containsKey("x-ms-retry-after-ms")) {
      // Read the value of the attribute as is
      String retryAfterTimeSpan = (String) attributes.get("x-ms-retry-after-ms"));

      // Convert the value into actionable duration
   LocalTime locaTime = LocalTime.parse(retryAfterTimeSpan);
   Duration duration = Duration.between(LocalTime.MIN, locaTime);

      // Perform a retry after "duration" interval of time has elapsed
    }
  }
}
```

Next steps

- HTTP status codes for Azure Cosmos DB
- Common Azure Cosmos DB REST response headers
- TinkerPop Graph Driver Provider Requirements

Introduction to Azure Cosmos DB: Table API

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: Table API

Azure Cosmos DB provides the Table API for applications that are written for Azure Table storage and that need premium capabilities like:

- Turnkey global distribution.
- Dedicated throughput worldwide.
- Single-digit millisecond latencies at the 99th percentile.
- Guaranteed high availability.
- Automatic secondary indexing.

Applications written for Azure Table storage can migrate to Azure Cosmos DB by using the Table API with no code changes and take advantage of premium capabilities. The Table API has client SDKs available for .NET, Java, Python, and Node.js.

> **NOTE**
>
> The serverless capacity mode is now available on Azure Cosmos DB's Table API.

> **IMPORTANT**
>
> The .NET Framework SDK Microsoft.Azure.CosmosDB.Table is in maintenance mode and it will be deprecated soon. Please upgrade to the new .NET Standard library Microsoft.Azure.Cosmos.Table to continue to get the latest features supported by the Table API.

Table offerings

If you currently use Azure Table Storage, you gain the following benefits by moving to the Azure Cosmos DB Table API:

FEATURE	AZURE TABLE STORAGE	AZURE COSMOS DB TABLE API
Latency	Fast, but no upper bounds on latency.	Single-digit millisecond latency for reads and writes, backed with <10 ms latency for reads and writes at the 99th percentile, at any scale, anywhere in the world.
Throughput	Variable throughput model. Tables have a scalability limit of 20,000 operations/s.	Highly scalable with dedicated reserved throughput per table that's backed by SLAs. Accounts have no upper limit on throughput and support >10 million operations/s per table.

FEATURE	AZURE TABLE STORAGE	AZURE COSMOS DB TABLE API
Global distribution	Single region with one optional readable secondary read region for high availability.	Turnkey global distribution from one to any number of regions. Support for automatic and manual failovers at any time, anywhere in the world. Multiple write regions to let any region accept write operations.
Indexing	Only primary index on PartitionKey and RowKey. No secondary indexes.	Automatic and complete indexing on all properties by default, with no index management.
Query	Query execution uses index for primary key, and scans otherwise.	Queries can take advantage of automatic indexing on properties for fast query times.
Consistency	Strong within primary region. Eventual within secondary region.	Five well-defined consistency levels to trade off availability, latency, throughput, and consistency based on your application needs.
Pricing	Storage-optimized.	Throughput-optimized.
SLAs	99.9% to 99.99% availability, depending on the replication strategy.	99.999% read availability, 99.99% write availability on a single-region account and 99.999% write availability on multi-region accounts. Comprehensive SLAs covering availability, latency, throughput and consistency.

Get started

Create an Azure Cosmos DB account in the Azure portal. Then get started with our Quick Start for Table API by using .NET.

> **IMPORTANT**
>
> If you created a Table API account during the preview, please create a new Table API account to work with the generally available Table API SDKs.

Next steps

Here are a few pointers to get you started:

- Build a .NET application by using the Table API
- Develop with the Table API in .NET
- Query table data by using the Table API
- Learn how to set up Azure Cosmos DB global distribution by using the Table API
- Azure Cosmos DB Table .NET Standard SDK
- Azure Cosmos DB Table .NET SDK
- Azure Cosmos DB Table Java SDK
- Azure Cosmos DB Table Node.js SDK
- Azure Cosmos DB Table SDK for Python

Quickstart: Build a Table API app with .NET SDK and Azure Cosmos DB

11/2/2020 • 9 minutes to read • Edit Online

APPLIES TO: Table API

This quickstart shows how to use .NET and the Azure Cosmos DB Table API to build an app by cloning an example from GitHub. This quickstart also shows you how to create an Azure Cosmos DB account and how to use Data Explorer to create tables and entities in the web-based Azure portal.

Prerequisites

If you don't already have Visual Studio 2019 installed, you can download and use the **free** Visual Studio 2019 Community Edition. Make sure that you enable **Azure development** during the Visual Studio setup.

If you don't have an Azure subscription, create a free account before you begin.

Create a database account

1. In a new browser window, sign in to the Azure portal.
2. In the left menu, select **Create a resource**.

3. On the **New** page, select **Databases** > **Azure Cosmos DB**.

4. On the **Create Azure Cosmos DB Account** page, enter the settings for the new Azure Cosmos DB account.

SETTING	VALUE	DESCRIPTION
Subscription	Your subscription	Select the Azure subscription that you want to use for this Azure Cosmos DB account.
Resource Group	**Create new**, then Account Name	Select **Create new**. Then enter a new resource group name for your account. For simplicity, use the same name as your Azure Cosmos DB account name.
Account Name	A unique name	Enter a unique name to identify your Azure Cosmos DB account. The account name can use only lowercase letters, numbers, and hyphens (-), and must be between 3 and 31 characters long.

SETTING	VALUE	DESCRIPTION
API	Table	The API determines the type of account to create. Azure Cosmos DB provides five APIs: Core (SQL) for document databases, Gremlin for graph databases, MongoDB for document databases, Azure Table, and Cassandra. You must create a separate account for each API. Select **Azure Table**, because in this quickstart you are creating a table that works with the Table API. Learn more about the Table API.
Location	The region closest to your users	Select a geographic location to host your Azure Cosmos DB account. Use the location that's closest to your users to give them the fastest access to the data.
Capacity mode	Provisioned throughput or Serverless	Select **Provisioned throughput** to create an account in provisioned throughput mode. Select **Serverless** to create an account in serverless mode.

You can leave the **Geo-Redundancy** and **Multi-region Writes** options at **Disable** to avoid additional charges, and skip the **Network** and **Tags** sections.

5. Select **Review+Create**. After the validation is complete, select **Create** to create the account.

6. It takes a few minutes to create the account. You'll see a message that states **Your deployment is underway**. Wait for the deployment to finish, and then select **Go to resource**.

Add a table

You can now use the Data Explorer tool in the Azure portal to create a database and table.

1. Select **Data Explorer** > **New Table**.

   The **Add Table** area is displayed on the far right, you may need to scroll right to see it.

2. In the **Add Table** page, enter the settings for the new table.

SETTING	SUGGESTED VALUE	DESCRIPTION
Table Id	sample-table	The ID for your new table. Table names have the same character requirements as database ids. Database names must be between 1 and 255 characters, and cannot contain `/ \ # ?` or a trailing space.
Throughput	400 RUs	Change the throughput to 400 request units per second (RU/s). If you want to reduce latency, you can scale up the throughput later.

3. Select **OK**.

4. Data Explorer displays the new database and table.

Add sample data

You can now add data to your new table using Data Explorer.

1. In Data Explorer, expand **sample-table**, select **Entities**, and then select **Add Entity**.

2. Now add data to the PartitionKey value box and RowKey value box, and select **Add Entity**.

You can now add more entities to your table, edit your entities, or query your data in Data Explorer. Data Explorer is also where you can scale your throughput and add stored procedures, user-defined functions, and triggers to your table.

Clone the sample application

Now let's clone a Table app from GitHub, set the connection string, and run it. You'll see how easy it is to work with data programmatically.

1. Open a command prompt, create a new folder named git-samples, then close the command prompt.

```
md "C:\git-samples"
```

2. Open a git terminal window, such as git bash, and use the `cd` command to change to the new folder to install the sample app.

```
cd "C:\git-samples"
```

3. Run the following command to clone the sample repository. This command creates a copy of the sample app on your computer.

```
git clone https://github.com/Azure-Samples/azure-cosmos-table-dotnet-core-getting-started.git
```

> **TIP**
>
> For a more detailed walkthrough of similar code, see the Cosmos DB Table API sample article.

Open the sample application in Visual Studio

1. In Visual Studio, from the **File** menu, choose **Open**, then choose **Project/Solution**.

2. Navigate to the folder where you cloned the sample application and open the TableStorage.sln file.

Review the code

This step is optional. If you're interested in learning how the database resources are created in the code, you can review the following snippets. Otherwise, you can skip ahead to update the connection string section of this doc.

- The following code shows how to create a table within the Azure Storage:

```
public static async Task<CloudTable> CreateTableAsync(string tableName)
{
    string storageConnectionString = AppSettings.LoadAppSettings().StorageConnectionString;

    // Retrieve storage account information from connection string.
    CloudStorageAccount storageAccount =
CreateStorageAccountFromConnectionString(storageConnectionString);

    // Create a table client for interacting with the table service
    CloudTableClient tableClient = storageAccount.CreateCloudTableClient(new
TableClientConfiguration());

    Console.WriteLine("Create a Table for the demo");

    // Create a table client for interacting with the table service
    CloudTable table = tableClient.GetTableReference(tableName);
    if (await table.CreateIfNotExistsAsync())
    {
        Console.WriteLine("Created Table named: {0}", tableName);
    }
    else
    {
        Console.WriteLine("Table {0} already exists", tableName);
    }

    Console.WriteLine();
    return table;
}
```

- The following code shows how to insert data into the table:

```
public static async Task<CustomerEntity> InsertOrMergeEntityAsync(CloudTable table, CustomerEntity
entity)
{
    if (entity == null)
    {
        throw new ArgumentNullException("entity");
    }

    try
    {
        // Create the InsertOrReplace table operation
        TableOperation insertOrMergeOperation = TableOperation.InsertOrMerge(entity);

        // Execute the operation.
        TableResult result = await table.ExecuteAsync(insertOrMergeOperation);
        CustomerEntity insertedCustomer = result.Result as CustomerEntity;

        if (result.RequestCharge.HasValue)
        {
            Console.WriteLine("Request Charge of InsertOrMerge Operation: " + result.RequestCharge);
        }

        return insertedCustomer;
    }
    catch (StorageException e)
    {
        Console.WriteLine(e.Message);
        Console.ReadLine();
        throw;
    }
}
```

- The following code shows how to query data from the table:

```
public static async Task<CustomerEntity> RetrieveEntityUsingPointQueryAsync(CloudTable table, string
partitionKey, string rowKey)
{
    try
    {
        TableOperation retrieveOperation = TableOperation.Retrieve<CustomerEntity>(partitionKey,
rowKey);
        TableResult result = await table.ExecuteAsync(retrieveOperation);
        CustomerEntity customer = result.Result as CustomerEntity;
        if (customer != null)
        {
            Console.WriteLine("\t{0}\t{1}\t{2}\t{3}", customer.PartitionKey, customer.RowKey,
customer.Email, customer.PhoneNumber);
        }

        if (result.RequestCharge.HasValue)
        {
            Console.WriteLine("Request Charge of Retrieve Operation: " + result.RequestCharge);
        }

        return customer;
    }
    catch (StorageException e)
    {
        Console.WriteLine(e.Message);
        Console.ReadLine();
        throw;
    }
}
```

- The following code shows how to delete data from the table:

```
public static async Task DeleteEntityAsync(CloudTable table, CustomerEntity deleteEntity)
{
    try
    {
        if (deleteEntity == null)
        {
            throw new ArgumentNullException("deleteEntity");
        }

        TableOperation deleteOperation = TableOperation.Delete(deleteEntity);
        TableResult result = await table.ExecuteAsync(deleteOperation);

        if (result.RequestCharge.HasValue)
        {
            Console.WriteLine("Request Charge of Delete Operation: " + result.RequestCharge);
        }

    }
    catch (StorageException e)
    {
        Console.WriteLine(e.Message);
        Console.ReadLine();
        throw;
    }
}
```

Update your connection string

Now go back to the Azure portal to get your connection string information and copy it into the app. This enables your app to communicate with your hosted database.

1. In the Azure portal, click **Connection String**. Use the copy button on the right side of the window to copy the **PRIMARY CONNECTION STRING**.

2. In Visual Studio, open the **Settings.json** file.

3. Paste the **PRIMARY CONNECTION STRING** from the portal into the StorageConnectionString value. Paste the string inside the quotes.

```
{
   "StorageConnectionString": "<Primary connection string from Azure portal>"
}
```

4. Press CTRL+S to save the **Settings.json** file.

You've now updated your app with all the info it needs to communicate with Azure Cosmos DB.

Build and deploy the app

1. In Visual Studio, right-click on the **CosmosTableSamples** project in **Solution Explorer** and then click **Manage NuGet Packages**.

2. In the NuGet **Browse** box, type Microsoft.Azure.Cosmos.Table. This will find the Cosmos DB Table API client library. Note that this library is currently available for .NET Framework and .NET Standard.

3. Click **Install** to install the **Microsoft.Azure.Cosmos.Table** library. This installs the Azure Cosmos DB Table API package and all dependencies.

4. When you run the entire app, sample data is inserted into the table entity and deleted at the end so you won't see any data inserted if you run the whole sample. However you can insert some breakpoints to view the data. Open BasicSamples.cs file and right-click on line 52, select **Breakpoint**, then select **Insert Breakpoint**. Insert another breakpoint on line 55.

5. Press F5 to run the application. The console window displays the name of the new table database (in this case, demoa13b1) in Azure Cosmos DB.

When you hit the first breakpoint, go back to Data Explorer in the Azure portal. Click the **Refresh** button, expand the demo* table, and click **Entities**. The **Entities** tab on the right shows the new entity that was added for Walter Harp. Note that the phone number for the new entity is 425-555-0101.

If you receive an error that says Settings.json file can't be found when running the project, you can resolve it by adding the following XML entry to the project settings. Right click on CosmosTableSamples, select Edit CosmosTableSamples.csproj and add the following itemGroup:

```
<ItemGroup>
  <None Update="Settings.json">
    <CopyToOutputDirectory>PreserveNewest</CopyToOutputDirectory>
  </None>
</ItemGroup>
```

6. Close the **Entities** tab in Data Explorer.

7. Press F5 to run the app to the next breakpoint.

   When you hit the breakpoint, switch back to the Azure portal, click **Entities** again to open the **Entities** tab, and note that the phone number has been updated to 425-555-0105.

8. Press F5 to run the app.

   The app adds entities for use in an advanced sample app that the Table API currently does not support. The app then deletes the table created by the sample app.

9. In the console window, press Enter to end the execution of the app.

Review SLAs in the Azure portal

The Azure portal monitors your Cosmos DB account throughput, storage, availability, latency, and consistency. Charts for metrics associated with an Azure Cosmos DB Service Level Agreement (SLA) show the SLA value compared to actual performance. This suite of metrics makes monitoring your SLAs transparent.

To review metrics and SLAs:

1. Select **Metrics** in your Cosmos DB account's navigation menu.

2. Select a tab such as **Latency**, and select a timeframe on the right. Compare the **Actual** and **SLA** lines on the charts.

3. Review the metrics on the other tabs.

Clean up resources

When you're done with your app and Azure Cosmos DB account, you can delete the Azure resources you created so you don't incur more charges. To delete the resources:

1. In the Azure portal Search bar, search for and select **Resource groups**.

2. From the list, select the resource group you created for this quickstart.

3. On the resource group **Overview** page, select **Delete resource group**.

4. In the next window, enter the name of the resource group to delete, and then select **Delete**.

Next steps

In this quickstart, you've learned how to create an Azure Cosmos DB account, create a table using the Data Explorer, and run an app. Now you can query your data using the Table API.

Import table data to the Table API

Quickstart: Build a Java app to manage Azure Cosmos DB Table API data

11/2/2020 • 7 minutes to read • Edit Online

APPLIES TO: Table API

In this quickstart, you create an Azure Cosmos DB Table API account, and use Data Explorer and a Java app cloned from GitHub to create tables and entities. Azure Cosmos DB is a multi-model database service that lets you quickly create and query document, table, key-value, and graph databases with global distribution and horizontal scale capabilities.

Prerequisites

- An Azure account with an active subscription. Create one for free. Or try Azure Cosmos DB for free without an Azure subscription. You can also use the Azure Cosmos DB Emulator with a URI of `https://localhost:8081` and the key `C2y6yDjf5/R+ob0N8A7Cgv30VRDJIWEHLM+4QDU5DE2nQ9nDuVTqobD4b8mGGyPMbIZnqyMsEcaGQy67XIw/Jw==`.
- Java Development Kit (JDK) 8. Point your `JAVA_HOME` environment variable to the folder where the JDK is installed.
- A Maven binary archive.
- Git.

Create a database account

> **IMPORTANT**
> You need to create a new Table API account to work with the generally available Table API SDKs. Table API accounts created during preview are not supported by the generally available SDKs.

1. In a new browser window, sign in to the Azure portal.
2. In the left menu, select **Create a resource**.

3. On the **New** page, select **Databases** > **Azure Cosmos DB**.

4. On the **Create Azure Cosmos DB Account** page, enter the settings for the new Azure Cosmos DB account.

SETTING	VALUE	DESCRIPTION
Subscription	Your subscription	Select the Azure subscription that you want to use for this Azure Cosmos DB account.
Resource Group	**Create new**, then Account Name	Select **Create new**. Then enter a new resource group name for your account. For simplicity, use the same name as your Azure Cosmos DB account name.
Account Name	A unique name	Enter a unique name to identify your Azure Cosmos DB account. The account name can use only lowercase letters, numbers, and hyphens (-), and must be between 3 and 31 characters long.

SETTING	VALUE	DESCRIPTION
API	Table	The API determines the type of account to create. Azure Cosmos DB provides five APIs: Core (SQL) for document databases, Gremlin for graph databases, MongoDB for document databases, Azure Table, and Cassandra. You must create a separate account for each API. Select **Azure Table**, because in this quickstart you are creating a table that works with the Table API. Learn more about the Table API.
Location	The region closest to your users	Select a geographic location to host your Azure Cosmos DB account. Use the location that's closest to your users to give them the fastest access to the data.
Capacity mode	Provisioned throughput or Serverless	Select **Provisioned throughput** to create an account in provisioned throughput mode. Select **Serverless** to create an account in serverless mode.

You can leave the **Geo-Redundancy** and **Multi-region Writes** options at **Disable** to avoid additional charges, and skip the **Network** and **Tags** sections.

5. Select **Review+Create**. After the validation is complete, select **Create** to create the account.

6. It takes a few minutes to create the account. You'll see a message that states **Your deployment is underway**. Wait for the deployment to finish, and then select **Go to resource**.

Add a table

You can now use the Data Explorer tool in the Azure portal to create a database and table.

1. Select **Data Explorer** > **New Table**.

   The **Add Table** area is displayed on the far right, you may need to scroll right to see it.

2. In the **Add Table** page, enter the settings for the new table.

SETTING	SUGGESTED VALUE	DESCRIPTION
Table Id	sample-table	The ID for your new table. Table names have the same character requirements as database ids. Database names must be between 1 and 255 characters, and cannot contain `/ \ # ?` or a trailing space.
Throughput	400 RUs	Change the throughput to 400 request units per second (RU/s). If you want to reduce latency, you can scale up the throughput later.

3. Select **OK**.

4. Data Explorer displays the new database and table.

Add sample data

You can now add data to your new table using Data Explorer.

1. In Data Explorer, expand **sample-table**, select **Entities**, and then select **Add Entity**.

2. Now add data to the PartitionKey value box and RowKey value box, and select **Add Entity**.

You can now add more entities to your table, edit your entities, or query your data in Data Explorer. Data Explorer is also where you can scale your throughput and add stored procedures, user-defined functions, and triggers to your table.

Clone the sample application

Now let's clone a Table app from GitHub, set the connection string, and run it. You'll see how easy it is to work with data programmatically.

1. Open a command prompt, create a new folder named git-samples, then close the command prompt.

```
md "C:\git-samples"
```

2. Open a git terminal window, such as git bash, and use the `cd` command to change to the new folder to install the sample app.

```
cd "C:\git-samples"
```

3. Run the following command to clone the sample repository. This command creates a copy of the sample app on your computer.

```
git clone https://github.com/Azure-Samples/storage-table-java-getting-started.git
```

> **TIP**
> For a more detailed walkthrough of similar code, see the Cosmos DB Table API sample article.

Review the code

This step is optional. If you're interested in learning how the database resources are created in the code, you can review the following snippets. Otherwise, you can skip ahead to update the connection string section of this doc.

- The following code shows how to create a table within the Azure Storage:

```
private static CloudTable createTable(CloudTableClient tableClient, String tableName) throws
StorageException, RuntimeException, IOException, InvalidKeyException,   IllegalArgumentException,
URISyntaxException, IllegalStateException {

  // Create a new table
  CloudTable table = tableClient.getTableReference(tableName);
  try {
      if (table.createIfNotExists() == false) {
          throw new IllegalStateException(String.format("Table with name \"%s\" already exists.",
tableName));
      }
  }
  catch (StorageException s) {
      if (s.getCause() instanceof java.net.ConnectException) {
          System.out.println("Caught connection exception from the client. If running with the default
configuration please make sure you have started the storage emulator.");
      }
      throw s;
  }

  return table;
}
```

- The following code shows how to insert data into the table:

```
private static void batchInsertOfCustomerEntities(CloudTable table) throws StorageException {

// Create the batch operation
TableBatchOperation batchOperation1 = new TableBatchOperation();
for (int i = 1; i <= 50; i++) {
    CustomerEntity entity = new CustomerEntity("Smith", String.format("%04d", i));
    entity.setEmail(String.format("smith%04d@contoso.com", i));
    entity.setHomePhoneNumber(String.format("425-555-%04d", i));
    entity.setWorkPhoneNumber(String.format("425-556-%04d", i));
    batchOperation1.insertOrMerge(entity);
}

// Execute the batch operation
table.execute(batchOperation1);
}
```

- The following code shows how to query data from the table:

```
private static void partitionScan(CloudTable table, String partitionKey) throws StorageException {

    // Create the partition scan query
    TableQuery<CustomerEntity> partitionScanQuery = TableQuery.from(CustomerEntity.class).where(
        (TableQuery.generateFilterCondition("PartitionKey", QueryComparisons.EQUAL, partitionKey)));

    // Iterate through the results
    for (CustomerEntity entity : table.execute(partitionScanQuery)) {
        System.out.println(String.format("\tCustomer: %s,%s\t%s\t%s\t%s", entity.getPartitionKey(),
entity.getRowKey(), entity.getEmail(), entity.getHomePhoneNumber(), entity.  getWorkPhoneNumber()));
    }
}
```

- The following code shows how to delete data from the table:

```
System.out.print("\nDelete any tables that were created.");

if (table1 != null && table1.deleteIfExists() == true) {
    System.out.println(String.format("\tSuccessfully deleted the table: %s", table1.getName()));
}

if (table2 != null && table2.deleteIfExists() == true) {
    System.out.println(String.format("\tSuccessfully deleted the table: %s", table2.getName()));
}
```

Update your connection string

Now go back to the Azure portal to get your connection string information and copy it into the app. This enables your app to communicate with your hosted database.

1. In your Azure Cosmos DB account in the Azure portal, select **Connection String**.

2. Copy the PRIMARY CONNECTION STRING using the copy button on the right.

3. Open *config.properties* from the *C:\git-samples\storage-table-java-getting-started\src\main\resources* folder.

4. Comment out line one and uncomment line two. The first two lines should now look like this.

```
#StorageConnectionString = UseDevelopmentStorage=true
StorageConnectionString = DefaultEndpointsProtocol=https;AccountName=[ACCOUNTNAME];AccountKey=
[ACCOUNTKEY]
```

5. Paste your PRIMARY CONNECTION STRING from the portal into the StorageConnectionString value in line 2.

> **IMPORTANT**
>
> If your Endpoint uses documents.azure.com, that means you have a preview account, and you need to create a new Table API account to work with the generally available Table API SDK.

6. Save the *config.properties* file.

You've now updated your app with all the info it needs to communicate with Azure Cosmos DB.

Run the app

1. In the git terminal window, `cd` to the storage-table-java-getting-started folder.

```
cd "C:\git-samples\storage-table-java-getting-started"
```

2. In the git terminal window, run the following commands to run the Java application.

```
mvn compile exec:java
```

   The console window displays the table data being added to the new table database in Azure Cosmos DB.

   You can now go back to Data Explorer and see, query, modify, and work with this new data.

Review SLAs in the Azure portal

The Azure portal monitors your Cosmos DB account throughput, storage, availability, latency, and consistency. Charts for metrics associated with an Azure Cosmos DB Service Level Agreement (SLA) show the SLA value compared to actual performance. This suite of metrics makes monitoring your SLAs transparent.

To review metrics and SLAs:

1. Select **Metrics** in your Cosmos DB account's navigation menu.

2. Select a tab such as **Latency**, and select a timeframe on the right. Compare the **Actual** and **SLA** lines on the charts.

3. Review the metrics on the other tabs.

Clean up resources

When you're done with your app and Azure Cosmos DB account, you can delete the Azure resources you created so you don't incur more charges. To delete the resources:

1. In the Azure portal Search bar, search for and select **Resource groups**.

2. From the list, select the resource group you created for this quickstart.

3. On the resource group **Overview** page, select **Delete resource group**.

4. In the next window, enter the name of the resource group to delete, and then select **Delete**.

Next steps

In this quickstart, you learned how to create an Azure Cosmos DB account, create a table using the Data Explorer, and run a Java app to add table data. Now you can query your data using the Table API.

Import table data to the Table API

Quickstart: Build a Table API app with Node.js and Azure Cosmos DB

11/2/2020 • 7 minutes to read • Edit Online

APPLIES TO: Table API

In this quickstart, you create an Azure Cosmos DB Table API account, and use Data Explorer and a Node.js app cloned from GitHub to create tables and entities. Azure Cosmos DB is a multi-model database service that lets you quickly create and query document, table, key-value, and graph databases with global distribution and horizontal scale capabilities.

Prerequisites

- An Azure account with an active subscription. Create one for free. Or try Azure Cosmos DB for free without an Azure subscription. You can also use the Azure Cosmos DB Emulator with a URI of `https://localhost:8081` and the key `C2y6yDjf5/R+ob0N8A7Cgv30VRDJIWEHLM+4QDU5DE2nQ9nDuVTqobD4b8mGGyPMbIZnqyMsEcaGQy67XIw/Jw==`.
- Node.js 0.10.29+ .
- Git.

Create a database account

> **IMPORTANT**
>
> You need to create a new Table API account to work with the generally available Table API SDKs. Table API accounts created during preview are not supported by the generally available SDKs.

1. In a new browser window, sign in to the Azure portal.
2. In the left menu, select **Create a resource**.

3. On the **New** page, select **Databases** > **Azure Cosmos DB**.

4. On the **Create Azure Cosmos DB Account** page, enter the settings for the new Azure Cosmos DB account.

SETTING	VALUE	DESCRIPTION
Subscription	Your subscription	Select the Azure subscription that you want to use for this Azure Cosmos DB account.
Resource Group	**Create new**, then Account Name	Select **Create new**. Then enter a new resource group name for your account. For simplicity, use the same name as your Azure Cosmos DB account name.
Account Name	A unique name	Enter a unique name to identify your Azure Cosmos DB account. The account name can use only lowercase letters, numbers, and hyphens (-), and must be between 3 and 31 characters long.

SETTING	VALUE	DESCRIPTION
API	Table	The API determines the type of account to create. Azure Cosmos DB provides five APIs: Core (SQL) for document databases, Gremlin for graph databases, MongoDB for document databases, Azure Table, and Cassandra. You must create a separate account for each API. Select **Azure Table**, because in this quickstart you are creating a table that works with the Table API. Learn more about the Table API.
Location	The region closest to your users	Select a geographic location to host your Azure Cosmos DB account. Use the location that's closest to your users to give them the fastest access to the data.
Capacity mode	Provisioned throughput or Serverless	Select **Provisioned throughput** to create an account in provisioned throughput mode. Select **Serverless** to create an account in serverless mode.

You can leave the **Geo-Redundancy** and **Multi-region Writes** options at **Disable** to avoid additional charges, and skip the **Network** and **Tags** sections.

5. Select **Review+Create**. After the validation is complete, select **Create** to create the account.

6. It takes a few minutes to create the account. You'll see a message that states **Your deployment is underway**. Wait for the deployment to finish, and then select **Go to resource**.

Add a table

You can now use the Data Explorer tool in the Azure portal to create a database and table.

1. Select **Data Explorer** > **New Table**.

   The **Add Table** area is displayed on the far right, you may need to scroll right to see it.

2. In the **Add Table** page, enter the settings for the new table.

SETTING	SUGGESTED VALUE	DESCRIPTION
Table Id	sample-table	The ID for your new table. Table names have the same character requirements as database ids. Database names must be between 1 and 255 characters, and cannot contain `/ \ # ?` or a trailing space.
Throughput	400 RUs	Change the throughput to 400 request units per second (RU/s). If you want to reduce latency, you can scale up the throughput later.

3. Select **OK**.

4. Data Explorer displays the new database and table.

Add sample data

You can now add data to your new table using Data Explorer.

1. In Data Explorer, expand **sample-table**, select **Entities**, and then select **Add Entity**.

2. Now add data to the PartitionKey value box and RowKey value box, and select **Add Entity**.

You can now add more entities to your table, edit your entities, or query your data in Data Explorer. Data Explorer is also where you can scale your throughput and add stored procedures, user-defined functions, and triggers to your table.

Clone the sample application

Now let's clone a Table app from GitHub, set the connection string, and run it. You'll see how easy it is to work with data programmatically.

1. Open a command prompt, create a new folder named git-samples, then close the command prompt.

```
md "C:\git-samples"
```

2. Open a git terminal window, such as git bash, and use the `cd` command to change to the new folder to install the sample app.

```
cd "C:\git-samples"
```

3. Run the following command to clone the sample repository. This command creates a copy of the sample app on your computer.

```
git clone https://github.com/Azure-Samples/storage-table-node-getting-started.git
```

> **TIP**
> For a more detailed walkthrough of similar code, see the Cosmos DB Table API sample article.

Review the code

This step is optional. If you're interested in learning how the database resources are created in the code, you can review the following snippets. Otherwise, you can skip ahead to update the connection string section of this doc.

- The following code shows how to create a table within the Azure Storage:

```
storageClient.createTableIfNotExists(tableName, function (error, createResult) {
  if (error) return callback(error);

  if (createResult.isSuccessful) {
    console.log("1. Create Table operation executed successfully for: ", tableName);
  }
}
```

- The following code shows how to insert data into the table:

```
var customer = createCustomerEntityDescriptor("Harp", "Walter", "Walter@contoso.com", "425-555-0101");

storageClient.insertOrMergeEntity(tableName, customer, function (error, result, response) {
  if (error) return callback(error);

  console.log("   insertOrMergeEntity succeeded.");
}
```

- The following code shows how to query data from the table:

```
console.log("6. Retrieving entities with surname of Smith and first names > 1 and <= 75");

var storageTableQuery = storage.TableQuery;
var segmentSize = 10;

// Demonstrate a partition range query whereby we are searching within a partition for a set of
entities that are within a specific range.
var tableQuery = new storageTableQuery()
    .top(segmentSize)
    .where('PartitionKey eq ?', lastName)
    .and('RowKey gt ?', "0001").and('RowKey le ?', "0075");

runPageQuery(tableQuery, null, function (error, result) {

    if (error) return callback(error);
```

- The following code shows how to delete data from the table:

```
storageClient.deleteEntity(tableName, customer, function entitiesQueried(error, result) {
    if (error) return callback(error);

    console.log("   deleteEntity succeeded.");
}
```

Update your connection string

Now go back to the Azure portal to get your connection string information and copy it into the app. This enables your app to communicate with your hosted database.

1. In your Azure Cosmos DB account in the Azure portal, select **Connection String**.

2. Copy the PRIMARY CONNECTION STRING using the copy button on the right side.

3. Open the *app.config* file, and paste the value into the connectionString on line three.

> **IMPORTANT**
>
> If your Endpoint uses documents.azure.com, that means you have a preview account, and you need to create a new Table API account to work with the generally available Table API SDK.

4. Save the *app.config* file.

You've now updated your app with all the info it needs to communicate with Azure Cosmos DB.

Run the app

1. In the git terminal window, `cd` to the storage-table-java-getting-started folder.

```
cd "C:\git-samples\storage-table-node-getting-started"
```

2. Run the following command to install the [azure], [node-uuid], [nconf] and [async] modules locally as well as to save an entry for them to the *package.json* file.

```
npm install azure-storage node-uuid async nconf --save
```

3. In the git terminal window, run the following commands to run the Node.js application.

```
node ./tableSample.js
```

The console window displays the table data being added to the new table database in Azure Cosmos DB.

You can now go back to Data Explorer and see, query, modify, and work with this new data.

Review SLAs in the Azure portal

The Azure portal monitors your Cosmos DB account throughput, storage, availability, latency, and consistency. Charts for metrics associated with an Azure Cosmos DB Service Level Agreement (SLA) show the SLA value compared to actual performance. This suite of metrics makes monitoring your SLAs transparent.

To review metrics and SLAs:

1. Select **Metrics** in your Cosmos DB account's navigation menu.
2. Select a tab such as **Latency**, and select a timeframe on the right. Compare the **Actual** and **SLA** lines on the charts.

3. Review the metrics on the other tabs.

Clean up resources

When you're done with your app and Azure Cosmos DB account, you can delete the Azure resources you created so you don't incur more charges. To delete the resources:

1. In the Azure portal Search bar, search for and select **Resource groups**.
2. From the list, select the resource group you created for this quickstart.

3. On the resource group **Overview** page, select **Delete resource group**.

4. In the next window, enter the name of the resource group to delete, and then select **Delete**.

Next steps

In this quickstart, you learned how to create an Azure Cosmos DB account, create a table using the Data Explorer, and run a Node.js app to add table data. Now you can query your data using the Table API.

Import table data to the Table API

Get started with Azure Table storage and the Azure Cosmos DB Table API using Python

11/2/2020 • 7 minutes to read • Edit Online

APPLIES TO: Table API

> **TIP**
>
> The content in this article applies to Azure Table storage and the Azure Cosmos DB Table API. The Azure Cosmos DB Table API is a premium offering for table storage that offers throughput-optimized tables, global distribution, and automatic secondary indexes.

The Azure Table storage and the Azure Cosmos DB are services that store structured NoSQL data in the cloud, providing a key/attribute store with a schemaless design. Because Table storage and Azure Cosmos DB are schemaless, it's easy to adapt your data as the needs of your application evolve. Access to the table storage and table API data is fast and cost-effective for many types of applications, and is typically lower in cost than traditional SQL for similar volumes of data.

You can use the Table storage or the Azure Cosmos DB to store flexible datasets like user data for web applications, address books, device information, or other types of metadata your service requires. You can store any number of entities in a table, and a storage account may contain any number of tables, up to the capacity limit of the storage account.

About this sample

This sample shows you how to use the Azure Cosmos DB Table SDK for Python in common Azure Table storage scenarios. The name of the SDK indicates it is for use with Azure Cosmos DB, but it works with both Azure Cosmos DB and Azure Tables storage, each service just has a unique endpoint. These scenarios are explored using Python examples that illustrate how to:

- Create and delete tables
- Insert and query entities
- Modify entities

While working through the scenarios in this sample, you may want to refer to the Azure Cosmos DB SDK for Python API reference.

Prerequisites

You need the following to complete this sample successfully:

- Python 2.7, 3.3, 3.4, 3.5, or 3.6
- Azure Cosmos DB Table SDK for Python. This SDK connects with both Azure Table storage and the Azure Cosmos DB Table API.
- Azure Storage account or Azure Cosmos DB account

Create an Azure service account

You can work with tables using the Azure Table storage or the Azure Cosmos DB. To learn more about the differences between table offerings in these two services, see the Table offerings article. You'll need to create an

account for the service you're going to use. The following sections show how to create both Azure Table storage and the Azure Cosmos DB account, however you can just use one of them.

Create an Azure storage account

The easiest way to create an Azure storage account is by using the Azure portal. To learn more, see Create a storage account.

You can also create an Azure storage account by using Azure PowerShell or Azure CLI.

If you prefer not to create a storage account at this time, you can also use the Azure Storage Emulator to run and test your code in a local environment. For more information, see Use the Azure Storage Emulator for development and testing.

Create an Azure Cosmos DB Table API account

For instructions on creating an Azure Cosmos DB Table API account, see Create a database account.

Install the Azure Cosmos DB Table SDK for Python

After you've created a Storage account, your next step is to install the Microsoft Azure Cosmos DB Table SDK for Python. For details on installing the SDK, refer to the README.rst file in the Cosmos DB Table SDK for Python repository on GitHub.

Import the TableService and Entity classes

To work with entities in the Azure Table service in Python, you use the TableService and Entity classes. Add this code near the top your Python file to import both:

```
from azure.cosmosdb.table.tableservice import TableService
from azure.cosmosdb.table.models import Entity
```

Connect to Azure Table service

To connect to Azure Storage Table service, create a TableService object, and pass in your Storage account name and account key. Replace `myaccount` and `mykey` with your account name and key.

```
table_service = TableService(account_name='myaccount', account_key='mykey')
```

Connect to Azure Cosmos DB

To connect to Azure Cosmos DB, copy your primary connection string from the Azure portal, and create a TableService object using your copied connection string:

```
table_service =
TableService(connection_string='DefaultEndpointsProtocol=https;AccountName=myaccount;AccountKey=mykey;TableEndp
oint=myendpoint;')
```

Create a table

Call create_table to create the table.

```
table_service.create_table('tasktable')
```

Add an entity to a table

To add an entity, you first create an object that represents your entity, then pass the object to the TableService.insert_entity method. The entity object can be a dictionary or an object of type Entity, and defines your entity's property names and values. Every entity must include the required PartitionKey and RowKey properties, in addition to any other properties you define for the entity.

This example creates a dictionary object representing an entity, then passes it to the insert_entity method to add it to the table:

```
task = {'PartitionKey': 'tasksSeattle', 'RowKey': '001',
        'description': 'Take out the trash', 'priority': 200}
table_service.insert_entity('tasktable', task)
```

This example creates an Entity object, then passes it to the insert_entity method to add it to the table:

```
task = Entity()
task.PartitionKey = 'tasksSeattle'
task.RowKey = '002'
task.description = 'Wash the car'
task.priority = 100
table_service.insert_entity('tasktable', task)
```

PartitionKey and RowKey

You must specify both a **PartitionKey** and a **RowKey** property for every entity. These are the unique identifiers of your entities, as together they form the primary key of an entity. You can query using these values much faster than you can query any other entity properties because only these properties are indexed.

The Table service uses **PartitionKey** to intelligently distribute table entities across storage nodes. Entities that have the same **PartitionKey** are stored on the same node. **RowKey** is the unique ID of the entity within the partition it belongs to.

Update an entity

To update all of an entity's property values, call the update_entity method. This example shows how to replace an existing entity with an updated version:

```
task = {'PartitionKey': 'tasksSeattle', 'RowKey': '001',
        'description': 'Take out the garbage', 'priority': 250}
table_service.update_entity('tasktable', task)
```

If the entity that is being updated doesn't already exist, then the update operation will fail. If you want to store an entity whether it exists or not, use insert_or_replace_entity. In the following example, the first call will replace the existing entity. The second call will insert a new entity, since no entity with the specified PartitionKey and RowKey exists in the table.

```
# Replace the entity created earlier
task = {'PartitionKey': 'tasksSeattle', 'RowKey': '001',
        'description': 'Take out the garbage again', 'priority': 250}
table_service.insert_or_replace_entity('tasktable', task)

# Insert a new entity
task = {'PartitionKey': 'tasksSeattle', 'RowKey': '003',
        'description': 'Buy detergent', 'priority': 300}
table_service.insert_or_replace_entity('tasktable', task)
```

> **TIP**
> The update_entity method replaces all properties and values of an existing entity, which you can also use to remove properties from an existing entity. You can use the merge_entity method to update an existing entity with new or modified property values without completely replacing the entity.

Modify multiple entities

To ensure the atomic processing of a request by the Table service, you can submit multiple operations together in a batch. First, use the TableBatch class to add multiple operations to a single batch. Next, call TableService.commit_batch to submit the operations in an atomic operation. All entities to be modified in batch must be in the same partition.

This example adds two entities together in a batch:

```
from azure.cosmosdb.table.tablebatch import TableBatch
batch = TableBatch()
task004 = {'PartitionKey': 'tasksSeattle', 'RowKey': '004',
           'description': 'Go grocery shopping', 'priority': 400}
task005 = {'PartitionKey': 'tasksSeattle', 'RowKey': '005',
           'description': 'Clean the bathroom', 'priority': 100}
batch.insert_entity(task004)
batch.insert_entity(task005)
table_service.commit_batch('tasktable', batch)
```

Batches can also be used with the context manager syntax:

```
task006 = {'PartitionKey': 'tasksSeattle', 'RowKey': '006',
           'description': 'Go grocery shopping', 'priority': 400}
task007 = {'PartitionKey': 'tasksSeattle', 'RowKey': '007',
           'description': 'Clean the bathroom', 'priority': 100}

with table_service.batch('tasktable') as batch:
    batch.insert_entity(task006)
    batch.insert_entity(task007)
```

Query for an entity

To query for an entity in a table, pass its PartitionKey and RowKey to the TableService.get_entity method.

```
task = table_service.get_entity('tasktable', 'tasksSeattle', '001')
print(task.description)
print(task.priority)
```

Query a set of entities

You can query for a set of entities by supplying a filter string with the **filter** parameter. This example finds all tasks in Seattle by applying a filter on PartitionKey:

```
tasks = table_service.query_entities(
    'tasktable', filter="PartitionKey eq 'tasksSeattle'")
for task in tasks:
    print(task.description)
    print(task.priority)
```

Query a subset of entity properties

You can also restrict which properties are returned for each entity in a query. This technique, called *projection*, reduces bandwidth and can improve query performance, especially for large entities or result sets. Use the **select** parameter and pass the names of the properties you want returned to the client.

The query in the following code returns only the descriptions of entities in the table.

> **NOTE**
>
> The following snippet works only against the Azure Storage. It is not supported by the Storage Emulator.

```
tasks = table_service.query_entities(
    'tasktable', filter="PartitionKey eq 'tasksSeattle'", select='description')
for task in tasks:
    print(task.description)
```

Delete an entity

Delete an entity by passing its **PartitionKey** and **RowKey** to the delete_entity method.

```
table_service.delete_entity('tasktable', 'tasksSeattle', '001')
```

Delete a table

If you no longer need a table or any of the entities within it, call the delete_table method to permanently delete the table from Azure Storage.

```
table_service.delete_table('tasktable')
```

Next steps

- FAQ - Develop with the Table API
- Azure Cosmos DB SDK for Python API reference
- Python Developer Center
- Microsoft Azure Storage Explorer: A free, cross-platform application for working visually with Azure Storage data on Windows, macOS, and Linux.
- Working with Python in Visual Studio (Windows)

Get started with Azure Cosmos DB Table API and Azure Table storage using the .NET SDK

12/4/2020 • 12 minutes to read • Edit Online

APPLIES TO: Table API

> **TIP**
>
> The content in this article applies to Azure Table storage and the Azure Cosmos DB Table API. The Azure Cosmos DB Table API is a premium offering for table storage that offers throughput-optimized tables, global distribution, and automatic secondary indexes.

You can use the Azure Cosmos DB Table API or Azure Table storage to store structured NoSQL data in the cloud, providing a key/attribute store with a schema less design. Because Azure Cosmos DB Table API and Table storage are schema less, it's easy to adapt your data as the needs of your application evolve. You can use Azure Cosmos DB Table API or the Table storage to store flexible datasets such as user data for web applications, address books, device information, or other types of metadata your service requires.

This tutorial describes a sample that shows you how to use the Microsoft Azure Cosmos DB Table Library for .NET with Azure Cosmos DB Table API and Azure Table storage scenarios. You must use the connection specific to the Azure service. These scenarios are explored using C# examples that illustrate how to create tables, insert/ update data, query data and delete the tables.

Prerequisites

You need the following to complete this sample successfully:

- Microsoft Visual Studio
- Microsoft Azure CosmosDB Table Library for .NET - This library is currently available for .NET Standard and .NET framework.
- Azure Cosmos DB Table API account.

Create an Azure Cosmos DB Table API account

1. In a new browser window, sign in to the Azure portal.
2. In the left menu, select **Create a resource**.

3. On the **New** page, select **Databases** > **Azure Cosmos DB**.

4. On the **Create Azure Cosmos DB Account** page, enter the settings for the new Azure Cosmos DB account.

SETTING	VALUE	DESCRIPTION
Subscription	Your subscription	Select the Azure subscription that you want to use for this Azure Cosmos DB account.
Resource Group	**Create new**, then Account Name	Select **Create new**. Then enter a new resource group name for your account. For simplicity, use the same name as your Azure Cosmos DB account name.
Account Name	A unique name	Enter a unique name to identify your Azure Cosmos DB account. The account name can use only lowercase letters, numbers, and hyphens (-), and must be between 3 and 31 characters long.

SETTING	VALUE	DESCRIPTION
API	Table	The API determines the type of account to create. Azure Cosmos DB provides five APIs: Core (SQL) for document databases, Gremlin for graph databases, MongoDB for document databases, Azure Table, and Cassandra. You must create a separate account for each API. Select **Azure Table**, because in this quickstart you are creating a table that works with the Table API. Learn more about the Table API.
Location	The region closest to your users	Select a geographic location to host your Azure Cosmos DB account. Use the location that's closest to your users to give them the fastest access to the data.
Capacity mode	Provisioned throughput or Serverless	Select **Provisioned throughput** to create an account in provisioned throughput mode. Select **Serverless** to create an account in serverless mode.

You can leave the **Geo-Redundancy** and **Multi-region Writes** options at **Disable** to avoid additional charges, and skip the **Network** and **Tags** sections.

5. Select **Review+Create**. After the validation is complete, select **Create** to create the account.

6. It takes a few minutes to create the account. You'll see a message that states **Your deployment is underway**. Wait for the deployment to finish, and then select **Go to resource**.

Create a .NET console project

In Visual Studio, create a new .NET console application. The following steps show you how to create a console application in Visual Studio 2019. You can use the Azure Cosmos DB Table Library in any type of .NET application, including an Azure cloud service or web app, and desktop and mobile applications. In this guide, we use a console application for simplicity.

1. Select **File** > **New** > **Project**.
2. Choose **Console App (.NET Core)**, and then select **Next**.
3. In the **Project name** field, enter a name for your application, such as **CosmosTableSamples**. (You can provide a different name as needed.)
4. Select **Create**.

All code examples in this sample can be added to the Main() method of your console application's **Program.cs** file.

Install the required NuGet package

To obtain the NuGet package, follow these steps:

1. Right-click your project in **Solution Explorer** and choose **Manage NuGet Packages**.
2. Search online for `Microsoft.Azure.Cosmos.Table`, `Microsoft.Extensions.Configuration`, `Microsoft.Extensions.Configuration.Json`, `Microsoft.Extensions.Configuration.Binder` and select **Install** to install the Microsoft Azure Cosmos DB Table Library.

Configure your storage connection string

1. From the Azure portal, navigate to your Azure Cosmos account or the Table Storage account.
2. Open the **Connection String** or **Access keys** pane. Use the copy button on the right side of the window to copy the **PRIMARY CONNECTION STRING**.

3. To configure your connection string, from visual studio right click on your project **CosmosTableSamples**.
4. Select **Add** and then **New Item**. Create a new file **Settings.json** with file type as **TypeScript JSON Configuration** File.
5. Replace the code in Settings.json file with the following code and assign your primary connection string:

```
{
"StorageConnectionString": <Primary connection string of your Azure Cosmos DB account>
}
```

6. Right click on your project **CosmosTableSamples**. Select **Add**, **New Item** and add a class named **AppSettings.cs**.

7. Add the following code to the AppSettings.cs file. This file reads the connection string from Settings.json file and assigns it to the configuration parameter:

```
namespace CosmosTableSamples
{
    using Microsoft.Extensions.Configuration;

    public class AppSettings
    {
        public string StorageConnectionString { get; set; }

        public static AppSettings LoadAppSettings()
        {
            IConfigurationRoot configRoot = new ConfigurationBuilder()
                .AddJsonFile("Settings.json")
                .Build();
            AppSettings appSettings = configRoot.Get<AppSettings>();
            return appSettings;
        }
    }
}
```

Parse and validate the connection details

1. Right click on your project **CosmosTableSamples**. Select **Add**, **New Item** and add a class named **Common.cs**. You will write code to validate the connection details and create a table within this class.

2. Define a method `CreateStorageAccountFromConnectionString` as shown below. This method will parse the connection string details and validate that the account name and account key details provided in the "Settings.json" file are valid.

```
using System;

namespace CosmosTableSamples
{
    using System.Threading.Tasks;
    using Microsoft.Azure.Cosmos.Table;
    using Microsoft.Azure.Documents;

    public class Common
    {
        public static CloudStorageAccount CreateStorageAccountFromConnectionString(string
storageConnectionString)
        {
            CloudStorageAccount storageAccount;
            try
            {
                storageAccount = CloudStorageAccount.Parse(storageConnectionString);
            }
            catch (FormatException)
            {
                Console.WriteLine("Invalid storage account information provided. Please confirm the
AccountName and AccountKey are valid in the app.config file - then restart the application.");
                throw;
            }
            catch (ArgumentException)
            {
                Console.WriteLine("Invalid storage account information provided. Please confirm the
AccountName and AccountKey are valid in the app.config file - then restart the sample.");
                Console.ReadLine();
                throw;
            }

            return storageAccount;
        }
```

Create a Table

The CloudTableClient class enables you to retrieve tables and entities stored in Table storage. Because we don't have any tables in the Cosmos DB Table API account, let's add the `CreateTableAsync` method to the **Common.cs** class to create a table:

```
public static async Task<CloudTable> CreateTableAsync(string tableName)
{
    string storageConnectionString = AppSettings.LoadAppSettings().StorageConnectionString;

    // Retrieve storage account information from connection string.
    CloudStorageAccount storageAccount = CreateStorageAccountFromConnectionString(storageConnectionString);

    // Create a table client for interacting with the table service
    CloudTableClient tableClient = storageAccount.CreateCloudTableClient(new TableClientConfiguration());

    Console.WriteLine("Create a Table for the demo");

    // Create a table client for interacting with the table service
    CloudTable table = tableClient.GetTableReference(tableName);
    if (await table.CreateIfNotExistsAsync())
    {
        Console.WriteLine("Created Table named: {0}", tableName);
    }
    else
    {
        Console.WriteLine("Table {0} already exists", tableName);
    }

    Console.WriteLine();
    return table;
}
```

If you get a "503 service unavailable exception" error, it's possible that the required ports for the connectivity mode are blocked by a firewall. To fix this issue, either open the required ports or use the gateway mode connectivity as shown in the following code:

```
tableClient.TableClientConfiguration.UseRestExecutorForCosmosEndpoint = true;
```

Define the entity

Entities map to C# objects by using a custom class derived from TableEntity. To add an entity to a table, create a class that defines the properties of your entity.

Right click on your project **CosmosTableSamples**. Select **Add**, **New Folder** and name it as **Model**. Within the Model folder add a class named **CustomerEntity.cs** and add the following code to it.

```
namespace CosmosTableSamples.Model
{
    using Microsoft.Azure.Cosmos.Table;

    public class CustomerEntity : TableEntity
    {
        public CustomerEntity()
        {
        }

        public CustomerEntity(string lastName, string firstName)
        {
            PartitionKey = lastName;
            RowKey = firstName;
        }

        public string Email { get; set; }

        public string PhoneNumber { get; set; }
    }
}
```

This code defines an entity class that uses the customer's first name as the row key and last name as the partition key. Together, an entity's partition and row key uniquely identify it in the table. Entities with the same partition key can be queried faster than entities with different partition keys but using diverse partition keys allows for greater scalability of parallel operations. Entities to be stored in tables must be of a supported type, for example derived from the TableEntity class. Entity properties you'd like to store in a table must be public properties of the type, and support both getting and setting of values. Also, your entity type must expose a parameter-less constructor.

Insert or merge an entity

The following code example creates an entity object and adds it to the table. The InsertOrMerge method within the TableOperation class is used to insert or merge an entity. The CloudTable.ExecuteAsync method is called to execute the operation.

Right click on your project **CosmosTableSamples**. Select **Add**, **New Item** and add a class named **SamplesUtils.cs**. This class stores all the code required to perform CRUD operations on the entities.

```
public static async Task<CustomerEntity> InsertOrMergeEntityAsync(CloudTable table, CustomerEntity entity)
{
    if (entity == null)
    {
        throw new ArgumentNullException("entity");
    }

    try
    {
        // Create the InsertOrReplace table operation
        TableOperation insertOrMergeOperation = TableOperation.InsertOrMerge(entity);

        // Execute the operation.
        TableResult result = await table.ExecuteAsync(insertOrMergeOperation);
        CustomerEntity insertedCustomer = result.Result as CustomerEntity;

        if (result.RequestCharge.HasValue)
        {
            Console.WriteLine("Request Charge of InsertOrMerge Operation: " + result.RequestCharge);
        }

        return insertedCustomer;
    }
    catch (StorageException e)
    {
        Console.WriteLine(e.Message);
        Console.ReadLine();
        throw;
    }
}
```

Get an entity from a partition

You can get entity from a partition by using the Retrieve method under the TableOperation class. The following code example gets the partition key row key, email and phone number of a customer entity. This example also prints out the request units consumed to query for the entity. To query for an entity, append the following code to **SamplesUtils.cs** file:

```
public static async Task<CustomerEntity> RetrieveEntityUsingPointQueryAsync(CloudTable table, string
partitionKey, string rowKey)
{
    try
    {
        TableOperation retrieveOperation = TableOperation.Retrieve<CustomerEntity>(partitionKey, rowKey);
        TableResult result = await table.ExecuteAsync(retrieveOperation);
        CustomerEntity customer = result.Result as CustomerEntity;
        if (customer != null)
        {
            Console.WriteLine("\t{0}\t{1}\t{2}\t{3}", customer.PartitionKey, customer.RowKey, customer.Email,
customer.PhoneNumber);
        }

        if (result.RequestCharge.HasValue)
        {
            Console.WriteLine("Request Charge of Retrieve Operation: " + result.RequestCharge);
        }

        return customer;
    }
    catch (StorageException e)
    {
        Console.WriteLine(e.Message);
        Console.ReadLine();
        throw;
    }
}
```

Delete an entity

You can easily delete an entity after you have retrieved it by using the same pattern shown for updating an entity. The following code retrieves and deletes a customer entity. To delete an entity, append the following code to **SamplesUtils.cs** file:

```
public static async Task DeleteEntityAsync(CloudTable table, CustomerEntity deleteEntity)
{
    try
    {
        if (deleteEntity == null)
        {
            throw new ArgumentNullException("deleteEntity");
        }

        TableOperation deleteOperation = TableOperation.Delete(deleteEntity);
        TableResult result = await table.ExecuteAsync(deleteOperation);

        if (result.RequestCharge.HasValue)
        {
            Console.WriteLine("Request Charge of Delete Operation: " + result.RequestCharge);
        }

    }
    catch (StorageException e)
    {
        Console.WriteLine(e.Message);
        Console.ReadLine();
        throw;
    }
}
```

Execute the CRUD operations on sample data

After you define the methods to create table, insert or merge entities, run these methods on the sample data. To do so, right click on your project **CosmosTableSamples**. Select **Add**, **New Item** and add a class named **BasicSamples.cs** and add the following code to it. This code creates a table, adds entities to it.

If don't want to delete the entity and table at the end of the project, comment the `await table.DeleteIfExistsAsync()` and `SamplesUtils.DeleteEntityAsync(table, customer)` methods from the following code. It's best to comment out these methods and validate the data before you delete the table.

```
using System;

namespace CosmosTableSamples
{
    using System.Threading.Tasks;
    using Microsoft.Azure.Cosmos.Table;
    using Model;

    class BasicSamples
    {
        public async Task RunSamples()
        {
            Console.WriteLine("Azure Cosmos DB Table - Basic Samples\n");
            Console.WriteLine();

            string tableName = "demo" + Guid.NewGuid().ToString().Substring(0, 5);

            // Create or reference an existing table
            CloudTable table = await Common.CreateTableAsync(tableName);

            try
            {
                // Demonstrate basic CRUD functionality
                await BasicDataOperationsAsync(table);
            }
            finally
            {
                // Delete the table
                await table.DeleteIfExistsAsync();
            }
        }

        private static async Task BasicDataOperationsAsync(CloudTable table)
        {
            // Create an instance of a customer entity. See the Model\CustomerEntity.cs for a description of
the entity.
            CustomerEntity customer = new CustomerEntity("Harp", "Walter")
            {
                Email = "Walter@contoso.com",
                PhoneNumber = "425-555-0101"
            };

            // Demonstrate how to insert the entity
            Console.WriteLine("Insert an Entity.");
            customer = await SamplesUtils.InsertOrMergeEntityAsync(table, customer);

            // Demonstrate how to Update the entity by changing the phone number
            Console.WriteLine("Update an existing Entity using the InsertOrMerge Upsert Operation.");
            customer.PhoneNumber = "425-555-0105";
            await SamplesUtils.InsertOrMergeEntityAsync(table, customer);
            Console.WriteLine();

            // Demonstrate how to Read the updated entity using a point query
            Console.WriteLine("Reading the updated Entity.");
            customer = await SamplesUtils.RetrieveEntityUsingPointQueryAsync(table, "Harp", "Walter");
            Console.WriteLine();

            // Demonstrate how to Delete an entity
            Console.WriteLine("Delete the entity. ");
            await SamplesUtils.DeleteEntityAsync(table, customer);
            Console.WriteLine();
        }
    }
}
```

The previous code creates a table that starts with “demo” and the generated GUID is appended to the table name. It

then adds a customer entity with first and last name as "Harp Walter" and later updates the phone number of this user.

In this tutorial, you built code to perform basic CRUD operations on the data stored in Table API account. You can also perform advanced operations such as – batch inserting data, query all the data within a partition, query a range of data within a partition, Lists tables in the account whose names begin with the specified prefix. You can download the complete sample form azure-cosmos-table-dotnet-core-getting-started GitHub repository. The AdvancedSamples.cs class has more operations that you can perform on the data.

Run the project

From your project **CosmosTableSamples**. Open the class named **Program.cs** and add the following code to it for calling BasicSamples when the project runs.

```
using System;

namespace CosmosTableSamples
{
    class Program
    {
        public static void Main(string[] args)
        {
            Console.WriteLine("Azure Cosmos Table Samples");
            BasicSamples basicSamples = new BasicSamples();
            basicSamples.RunSamples().Wait();

            AdvancedSamples advancedSamples = new AdvancedSamples();
            advancedSamples.RunSamples().Wait();

            Console.WriteLine();
            Console.WriteLine("Press any key to exit");
            Console.Read();
        }
    }
}
```

Now build the solution and press F5 to run the project. When the project is run, you will see the following output in the command prompt:

If you receive an error that says Settings.json file can't be found when running the project, you can resolve it by adding the following XML entry to the project settings. Right click on CosmosTableSamples, select Edit CosmosTableSamples.csproj and add the following itemGroup:

```
<ItemGroup>
  <None Update="Settings.json">
    <CopyToOutputDirectory>PreserveNewest</CopyToOutputDirectory>
  </None>
</ItemGroup>
```

Now you can sign into the Azure portal and verify that the data exists in the table.

Next steps

You can now proceed to the next tutorial and learn how to migrate data to Azure Cosmos DB Table API account.

Migrate data to Azure Comsos DB Table API

Migrate your data to Azure Cosmos DB Table API account

11/2/2020 • 6 minutes to read • Edit Online

APPLIES TO: Table API

This tutorial provides instructions on importing data for use with the Azure Cosmos DB Table API. If you have data stored in Azure Table storage, you can use either the Data Migration Tool or AzCopy to import your data to Azure Cosmos DB Table API. If you have data stored in an Azure Cosmos DB Table API (preview) account, you must use the Data Migration tool to migrate your data.

This tutorial covers the following tasks:

- Importing data with the Data Migration tool
- Importing data with AzCopy
- Migrating from Table API (preview) to Table API

Prerequisites

- **Increase throughput:** The duration of your data migration depends on the amount of throughput you set up for an individual container or a set of containers. Be sure to increase the throughput for larger data migrations. After you've completed the migration, decrease the throughput to save costs. For more information about increasing throughput in the Azure portal, see Performance levels and pricing tiers in Azure Cosmos DB.

- **Create Azure Cosmos DB resources:** Before you start the migrating data, pre-create all your tables from the Azure portal. If you are migrating to an Azure Cosmos DB account that has database level throughput, make sure to provide a partition key when creating the Azure Cosmos DB tables.

Data Migration tool

The command-line Azure Cosmos DB Data Migration tool (dt.exe) can be used to import your existing Azure Table storage data to a Table API GA account, or migrate data from a Table API (preview) account into a Table API GA account. Other sources are not currently supported. The UI based Data Migration tool (dtui.exe) is not currently supported for Table API accounts.

To perform a migration of table data, complete the following tasks:

1. Download the migration tool from GitHub.

2. Run `dt.exe` using the command-line arguments for your scenario. `dt.exe` takes a command in the following format:

```
dt.exe [/<option>:<value>] /s:<source-name> [/s.<source-option>:<value>] /t:<target-name> [/t.
<target-option>:<value>]
```

The supported options for this command are:

- **/ErrorLog:** Optional. Name of the CSV file to redirect data transfer failures
- **/OverwriteErrorLog:** Optional. Overwrite error log file
- **/ProgressUpdateInterval:** Optional, default is 00:00:01. Time interval to refresh on-screen data transfer

progress

- **/ErrorDetails:** Optional, default is None. Specifies that detailed error information should be displayed for the following errors: None, Critical, All
- **/EnableCosmosTableLog:** Optional. Direct the log to a cosmos table account. If set, this defaults to destination account connection string unless /CosmosTableLogConnectionString is also provided. This is useful if multiple instances of DT are being run simultaneously.
- **/CosmosTableLogConnectionString:** Optional. ConnectionString to direct the log to a remote cosmos table account.

Command-line source settings

Use the following source options when defining Azure Table Storage or Table API preview as the source of the migration.

- **/s:AzureTable:** Reads data from Azure Table storage
- **/s.ConnectionString:** Connection string for the table endpoint. This can be retrieved from the Azure portal
- **/s.LocationMode:** Optional, default is PrimaryOnly. Specifies which location mode to use when connecting to Azure Table storage: PrimaryOnly, PrimaryThenSecondary, SecondaryOnly, SecondaryThenPrimary
- **/s.Table:** Name of the Azure Table
- **/s.InternalFields:** Set to All for table migration as RowKey and PartitionKey are required for import.
- **/s.Filter:** Optional. Filter string to apply
- **/s.Projection:** Optional. List of columns to select

To retrieve the source connection string when importing from Azure Table storage, open the Azure portal and click **Storage accounts** > **Account** > **Access keys**, and then use the copy button to copy the **Connection string**.

To retrieve the source connection string when importing from an Azure Cosmos DB Table API (preview) account, open the Azure portal, click **Azure Cosmos DB** > **Account** > **Connection String** and use the copy button to copy the **Connection String**.

Sample Azure Table Storage command

Sample Azure Cosmos DB Table API (preview) command

Command-line target settings

Use the following target options when defining Azure Cosmos DB Table API as the target of the migration.

- **/t:TableAPIBulk**: Uploads data into Azure CosmosDB Table in batches
- **/t.ConnectionString**: Connection string for the table endpoint
- **/t.TableName**: Specifies the name of the table to write to
- **/t.Overwrite**: Optional, default is false. Specifies if existing values should be overwritten
- **/t.MaxInputBufferSize**: Optional, default is 1GB. Approximate estimate of input bytes to buffer before flushing data to sink
- **/t.Throughput**: Optional, service defaults if not specified. Specifies throughput to configure for table
- **/t.MaxBatchSize**: Optional, default is 2MB. Specify the batch size in bytes

Sample command: Source is Azure Table storage

Here is a command-line sample showing how to import from Azure Table storage to Table API:

```
dt /s:AzureTable /s.ConnectionString:DefaultEndpointsProtocol=https;AccountName=<Azure Table storage account
name>;AccountKey=<Account Key>;EndpointSuffix=core.windows.net /s.Table:<Table name> /t:TableAPIBulk
/t.ConnectionString:DefaultEndpointsProtocol=https;AccountName=<Azure Cosmos DB account name>;AccountKey=
<Azure Cosmos DB account key>;TableEndpoint=https://<Account name>.table.cosmosdb.azure.com:443 /t.TableName:
<Table name> /t.Overwrite
```

Sample command: Source is Azure Cosmos DB Table API (preview)

Here is a command-line sample to import from Table API preview to Table API GA:

```
dt /s:AzureTable /s.ConnectionString:DefaultEndpointsProtocol=https;AccountName=<Table API preview account
name>;AccountKey=<Table API preview account key>;TableEndpoint=https://<Account Name>.documents.azure.com;
/s.Table:<Table name> /t:TableAPIBulk /t.ConnectionString:DefaultEndpointsProtocol=https;AccountName=<Azure
Cosmos DB account name>;AccountKey=<Azure Cosmos DB account key>;TableEndpoint=https://<Account
name>.table.cosmosdb.azure.com:443 /t.TableName:<Table name> /t.Overwrite
```

Migrate data by using AzCopy

Using the AzCopy command-line utility is the other option for migrating data from Azure Table storage to the Azure Cosmos DB Table API. To use AzCopy, you first export your data as described in Export data from Table

storage, then import the data to Azure Cosmos DB as described in Azure Cosmos DB Table API.

When performing the import into Azure Cosmos DB, refer to the following sample. Note that the /Dest value uses cosmosdb, not core.

Example import command:

```
AzCopy /Source:C:\myfolder\ /Dest:https://myaccount.table.cosmosdb.windows.net/mytable1/ /DestKey:key
/Manifest:"myaccount_mytable_20140103T112020.manifest" /EntityOperation:InsertOrReplace
```

Migrate from Table API (preview) to Table API

> **WARNING**
> If you want to immediately enjoy the benefits of the generally available tables then please migrate your existing preview tables as specified in this section, otherwise we will be performing auto-migrations for existing preview customers in the coming weeks, note however that auto-migrated preview tables will have certain restrictions to them that newly created tables will not.

The Table API is now generally available (GA). There are differences between the preview and GA versions of tables both in the code that runs in the cloud as well as in code that runs at the client. Therefore it is not advised to try to mix a preview SDK client with a GA Table API account, and vice versa. Table API preview customers who want to continue to use their existing tables but in a production environment need to migrate from the preview to the GA environment, or wait for auto-migration. If you wait for auto-migration, you will be notified of the restrictions on the migrated tables. After migration, you will be able to create new tables on your existing account without restrictions (only migrated tables will have restrictions).

To migrate from Table API (preview) to the generally available Table API:

1. Create a new Azure Cosmos DB account and set its API type to Azure Table as described in Create a database account.
2. Change clients to use a GA release of the Table API SDKs.
3. Migrate the client data from preview tables to GA tables by using the Data Migration tool. Instructions on using the data migration tool for this purpose are described in Data Migration tool.

Next steps

In this tutorial you learned how to:

- Import data with the Data Migration tool
- Import data with AzCopy
- Migrate from Table API (preview) to Table API

You can now proceed to the next tutorial and learn how to query data using the Azure Cosmos DB Table API.

How to query data?

Tutorial: Query Azure Cosmos DB by using the Table API

11/2/2020 • 2 minutes to read • Edit Online

APPLIES TO: Table API

The Azure Cosmos DB Table API supports OData and LINQ queries against key/value (table) data.

This article covers the following tasks:

- Querying data with the Table API

The queries in this article use the following sample `People` table:

PARTITIONKEY	ROWKEY	EMAIL	PHONENUMBER
Harp	Walter	Walter@contoso.com	425-555-0101
Smith	Ben	Ben@contoso.com	425-555-0102
Smith	Jeff	Jeff@contoso.com	425-555-0104

See Querying Tables and Entities for details on how to query by using the Table API.

For more information on the premium capabilities that Azure Cosmos DB offers, see Azure Cosmos DB Table API and Develop with the Table API in .NET.

Prerequisites

For these queries to work, you must have an Azure Cosmos DB account and have entity data in the container. Don't have any of those? Complete the five-minute quickstart or the developer tutorial to create an account and populate your database.

Query on PartitionKey and RowKey

Because the PartitionKey and RowKey properties form an entity's primary key, you can use the following special syntax to identify the entity:

Query

```
https://<mytableendpoint>/People(PartitionKey='Harp',RowKey='Walter')
```

Results

PARTITIONKEY	ROWKEY	EMAIL	PHONENUMBER
Harp	Walter	Walter@contoso.com	425-555-0104

Alternatively, you can specify these properties as part of the `$filter` option, as shown in the following section. Note that the key property names and constant values are case-sensitive. Both the PartitionKey and RowKey properties are of type String.

Query by using an OData filter

When you're constructing a filter string, keep these rules in mind:

- Use the logical operators defined by the OData Protocol Specification to compare a property to a value. Note that you can't compare a property to a dynamic value. One side of the expression must be a constant.
- The property name, operator, and constant value must be separated by URL-encoded spaces. A space is URL-encoded as `%20`.
- All parts of the filter string are case-sensitive.
- The constant value must be of the same data type as the property in order for the filter to return valid results. For more information about supported property types, see Understanding the Table Service Data Model.

Here's an example query that shows how to filter by the PartitionKey and Email properties by using an OData `$filter`.

Query

```
https://<mytableapi-endpoint>/People()?
$filter=PartitionKey%20eq%20'Smith'%20and%20Email%20eq%20'Ben@contoso.com'
```

For more information on how to construct filter expressions for various data types, see Querying Tables and Entities.

Results

PARTITIONKEY	ROWKEY	EMAIL	PHONENUMBER
Smith	Ben	Ben@contoso.com	425-555-0102

The queries on datetime properties don't return any data when executed in Azure Cosmos DB's Table API. While the Azure Table storage stores date values with time granularity of ticks, the Table API in Azure Cosmos DB uses the `_ts` property. The `_ts` property is at a second level of granularity, which isn't an OData filter. So, the queries on timestamp properties are blocked by Azure Cosmos DB. As a workaround, you can define a custom datetime or long data type property and set the date value from the client.

Query by using LINQ

You can also query by using LINQ, which translates to the corresponding OData query expressions. Here's an example of how to build queries by using the .NET SDK:

```
IQueryable<CustomerEntity> linqQuery = table.CreateQuery<CustomerEntity>()
            .Where(x => x.PartitionKey == "4")
            .Select(x => new CustomerEntity() { PartitionKey = x.PartitionKey, RowKey = x.RowKey, Email =
x.Email });
```

Next steps

In this tutorial, you've done the following:

- Learned how to query by using the Table API

You can now proceed to the next tutorial to learn how to distribute your data globally.

Distribute your data globally

Set up Azure Cosmos DB global distribution using the Table API

11/2/2020 • 3 minutes to read • Edit Online

APPLIES TO: Table API

This article covers the following tasks:

- Configure global distribution using the Azure portal
- Configure global distribution using the Table API

Add global database regions using the Azure portal

Azure Cosmos DB is available in all Azure regions worldwide. After selecting the default consistency level for your database account, you can associate one or more regions (depending on your choice of default consistency level and global distribution needs).

1. In the Azure portal, in the left bar, click **Azure Cosmos DB**.
2. In the **Azure Cosmos DB** page, select the database account to modify.
3. In the account page, click **Replicate data globally** from the menu.
4. In the **Replicate data globally** page, select the regions to add or remove by clicking regions in the map, and then click **Save**. There is a cost to adding regions, see the pricing page or the Distribute data globally with Azure Cosmos DB article for more information.

Once you add a second region, the **Manual Failover** option is enabled on the **Replicate data globally** page in the portal. You can use this option to test the failover process or change the primary write region. Once you add a third region, the **Failover Priorities** option is enabled on the same page so that you can change the failover order for reads.

Selecting global database regions

There are two common scenarios for configuring two or more regions:

1. Delivering low-latency access to data to end users no matter where they are located around the globe
2. Adding regional resiliency for business continuity and disaster recovery (BCDR)

For delivering low-latency to end users, it is recommended that you deploy both the application and Azure Cosmos DB in the regions that correspond to where the application's users are located.

For BCDR, it is recommended to add regions based on the region pairs described in the Business continuity and disaster recovery (BCDR): Azure Paired Regions article.

Connecting to a preferred region using the Table API

In order to take advantage of the global distribution, client applications should specify the current location where their application is running. This is done by setting the `CosmosExecutorConfiguration.CurrentRegion` property. The `CurrentRegion` property should contain a single location. Each client instance can specify their own region for low latency reads. The region must be named by using their display names such as "West US".

The Azure Cosmos DB Table API SDK automatically picks the best endpoint to communicate with based on the account configuration and current regional availability. It prioritizes the closest region to provide better latency to clients. After you set the current `CurrentRegion` property, read and write requests are directed as follows:

- **Read requests:** All read requests are sent to the configured `CurrentRegion`. Based on the proximity, the SDK automatically selects a fallback geo-replicated region for high availability.
- **Write requests:** The SDK automatically sends all write requests to the current write region. In an account with multi-region writes, current region will serve the writes requests as well. Based on the proximity, the SDK automatically selects a fallback geo-replicated region for high availability.

If you don't specify the `CurrentRegion` property, the SDK uses the current write region for all operations.

For example, if an Azure Cosmos account is in "West US" and "East US" regions. If "West US" is the write region and the application is present in "East US". If the CurrentRegion property is not configured, all the read and write requests are always directed to the "West US" region. If the CurrentRegion property is configured, all the read requests are served from "East US" region.

Next steps

In this tutorial, you've done the following:

- Configure global distribution using the Azure portal
- Configure global distribution using the Azure Cosmos DB Table APIs

Get started with Azure Cosmos DB Table API and Azure Table storage using the .NET SDK

12/4/2020 • 12 minutes to read • Edit Online

APPLIES TO: Table API

> **TIP**
>
> The content in this article applies to Azure Table storage and the Azure Cosmos DB Table API. The Azure Cosmos DB Table API is a premium offering for table storage that offers throughput-optimized tables, global distribution, and automatic secondary indexes.

You can use the Azure Cosmos DB Table API or Azure Table storage to store structured NoSQL data in the cloud, providing a key/attribute store with a schema less design. Because Azure Cosmos DB Table API and Table storage are schema less, it's easy to adapt your data as the needs of your application evolve. You can use Azure Cosmos DB Table API or the Table storage to store flexible datasets such as user data for web applications, address books, device information, or other types of metadata your service requires.

This tutorial describes a sample that shows you how to use the Microsoft Azure Cosmos DB Table Library for .NET with Azure Cosmos DB Table API and Azure Table storage scenarios. You must use the connection specific to the Azure service. These scenarios are explored using C# examples that illustrate how to create tables, insert/update data, query data and delete the tables.

Prerequisites

You need the following to complete this sample successfully:

- Microsoft Visual Studio
- Microsoft Azure CosmosDB Table Library for .NET - This library is currently available for .NET Standard and .NET framework.
- Azure Cosmos DB Table API account.

Create an Azure Cosmos DB Table API account

1. In a new browser window, sign in to the Azure portal.
2. In the left menu, select **Create a resource**.

3. On the **New** page, select **Databases** > **Azure Cosmos DB**.

4. On the **Create Azure Cosmos DB Account** page, enter the settings for the new Azure Cosmos DB account.

SETTING	VALUE	DESCRIPTION
Subscription	Your subscription	Select the Azure subscription that you want to use for this Azure Cosmos DB account.
Resource Group	**Create new**, then Account Name	Select **Create new**. Then enter a new resource group name for your account. For simplicity, use the same name as your Azure Cosmos DB account name.
Account Name	A unique name	Enter a unique name to identify your Azure Cosmos DB account. The account name can use only lowercase letters, numbers, and hyphens (-), and must be between 3 and 31 characters long.

SETTING	VALUE	DESCRIPTION
API	Table	The API determines the type of account to create. Azure Cosmos DB provides five APIs: Core (SQL) for document databases, Gremlin for graph databases, MongoDB for document databases, Azure Table, and Cassandra. You must create a separate account for each API. Select **Azure Table**, because in this quickstart you are creating a table that works with the Table API. Learn more about the Table API.
Location	The region closest to your users	Select a geographic location to host your Azure Cosmos DB account. Use the location that's closest to your users to give them the fastest access to the data.
Capacity mode	Provisioned throughput or Serverless	Select **Provisioned throughput** to create an account in provisioned throughput mode. Select **Serverless** to create an account in serverless mode.

You can leave the **Geo-Redundancy** and **Multi-region Writes** options at **Disable** to avoid additional charges, and skip the **Network** and **Tags** sections.

5. Select **Review+Create**. After the validation is complete, select **Create** to create the account.

6. It takes a few minutes to create the account. You'll see a message that states **Your deployment is underway**. Wait for the deployment to finish, and then select **Go to resource**.

Create a .NET console project

In Visual Studio, create a new .NET console application. The following steps show you how to create a console application in Visual Studio 2019. You can use the Azure Cosmos DB Table Library in any type of .NET application, including an Azure cloud service or web app, and desktop and mobile applications. In this guide, we use a console application for simplicity.

1. Select **File** > **New** > **Project**.

2. Choose **Console App (.NET Core)**, and then select **Next**.

3. In the **Project name** field, enter a name for your application, such as **CosmosTableSamples**. (You can provide a different name as needed.)

4. Select **Create**.

All code examples in this sample can be added to the Main() method of your console application's **Program.cs** file.

Install the required NuGet package

To obtain the NuGet package, follow these steps:

1. Right-click your project in **Solution Explorer** and choose **Manage NuGet Packages**.

2. Search online for `Microsoft.Azure.Cosmos.Table`, `Microsoft.Extensions.Configuration`, `Microsoft.Extensions.Configuration.Json`, `Microsoft.Extensions.Configuration.Binder` and select **Install** to install the Microsoft Azure Cosmos DB Table Library.

Configure your storage connection string

1. From the Azure portal, navigate to your Azure Cosmos account or the Table Storage account.

2. Open the **Connection String** or **Access keys** pane. Use the copy button on the right side of the window to copy the **PRIMARY CONNECTION STRING**.

3. To configure your connection string, from visual studio right click on your project **CosmosTableSamples**.

4. Select **Add** and then **New Item**. Create a new file **Settings.json** with file type as **TypeScript JSON Configuration** File.

5. Replace the code in Settings.json file with the following code and assign your primary connection string:

```
{
"StorageConnectionString": <Primary connection string of your Azure Cosmos DB account>
}
```

6. Right click on your project **CosmosTableSamples**. Select **Add**, **New Item** and add a class named **AppSettings.cs**.

7. Add the following code to the AppSettings.cs file. This file reads the connection string from Settings.json file and assigns it to the configuration parameter:

```
namespace CosmosTableSamples
{
    using Microsoft.Extensions.Configuration;

    public class AppSettings
    {
        public string StorageConnectionString { get; set; }

        public static AppSettings LoadAppSettings()
        {
            IConfigurationRoot configRoot = new ConfigurationBuilder()
                .AddJsonFile("Settings.json")
                .Build();
            AppSettings appSettings = configRoot.Get<AppSettings>();
            return appSettings;
        }
    }
}
```

Parse and validate the connection details

1. Right click on your project **CosmosTableSamples**. Select **Add**, **New Item** and add a class named **Common.cs**. You will write code to validate the connection details and create a table within this class.

2. Define a method `CreateStorageAccountFromConnectionString` as shown below. This method will parse the connection string details and validate that the account name and account key details provided in the "Settings.json" file are valid.

```
using System;

namespace CosmosTableSamples
{
    using System.Threading.Tasks;
    using Microsoft.Azure.Cosmos.Table;
    using Microsoft.Azure.Documents;

    public class Common
    {
        public static CloudStorageAccount CreateStorageAccountFromConnectionString(string
storageConnectionString)
        {
            CloudStorageAccount storageAccount;
            try
            {
                storageAccount = CloudStorageAccount.Parse(storageConnectionString);
            }
            catch (FormatException)
            {
                Console.WriteLine("Invalid storage account information provided. Please confirm the
AccountName and AccountKey are valid in the app.config file - then restart the application.");
                throw;
            }
            catch (ArgumentException)
            {
                Console.WriteLine("Invalid storage account information provided. Please confirm the
AccountName and AccountKey are valid in the app.config file - then restart the sample.");
                Console.ReadLine();
                throw;
            }

            return storageAccount;
        }
```

Create a Table

The CloudTableClient class enables you to retrieve tables and entities stored in Table storage. Because we don't have any tables in the Cosmos DB Table API account, let's add the `CreateTableAsync` method to the **Common.cs** class to create a table:

```
public static async Task<CloudTable> CreateTableAsync(string tableName)
{
    string storageConnectionString = AppSettings.LoadAppSettings().StorageConnectionString;

    // Retrieve storage account information from connection string.
    CloudStorageAccount storageAccount = CreateStorageAccountFromConnectionString(storageConnectionString);

    // Create a table client for interacting with the table service
    CloudTableClient tableClient = storageAccount.CreateCloudTableClient(new TableClientConfiguration());

    Console.WriteLine("Create a Table for the demo");

    // Create a table client for interacting with the table service
    CloudTable table = tableClient.GetTableReference(tableName);
    if (await table.CreateIfNotExistsAsync())
    {
        Console.WriteLine("Created Table named: {0}", tableName);
    }
    else
    {
        Console.WriteLine("Table {0} already exists", tableName);
    }

    Console.WriteLine();
    return table;
}
```

If you get a "503 service unavailable exception" error, it's possible that the required ports for the connectivity mode are blocked by a firewall. To fix this issue, either open the required ports or use the gateway mode connectivity as shown in the following code:

```
tableClient.TableClientConfiguration.UseRestExecutorForCosmosEndpoint = true;
```

Define the entity

Entities map to C# objects by using a custom class derived from TableEntity. To add an entity to a table, create a class that defines the properties of your entity.

Right click on your project **CosmosTableSamples**. Select **Add**, **New Folder** and name it as **Model**. Within the Model folder add a class named **CustomerEntity.cs** and add the following code to it.

```
namespace CosmosTableSamples.Model
{
    using Microsoft.Azure.Cosmos.Table;

    public class CustomerEntity : TableEntity
    {
        public CustomerEntity()
        {
        }

        public CustomerEntity(string lastName, string firstName)
        {
            PartitionKey = lastName;
            RowKey = firstName;
        }

        public string Email { get; set; }

        public string PhoneNumber { get; set; }
    }
}
```

This code defines an entity class that uses the customer's first name as the row key and last name as the partition key. Together, an entity's partition and row key uniquely identify it in the table. Entities with the same partition key can be queried faster than entities with different partition keys but using diverse partition keys allows for greater scalability of parallel operations. Entities to be stored in tables must be of a supported type, for example derived from the TableEntity class. Entity properties you'd like to store in a table must be public properties of the type, and support both getting and setting of values. Also, your entity type must expose a parameter-less constructor.

Insert or merge an entity

The following code example creates an entity object and adds it to the table. The InsertOrMerge method within the TableOperation class is used to insert or merge an entity. The CloudTable.ExecuteAsync method is called to execute the operation.

Right click on your project **CosmosTableSamples**. Select **Add**, **New Item** and add a class named **SamplesUtils.cs**. This class stores all the code required to perform CRUD operations on the entities.

```
public static async Task<CustomerEntity> InsertOrMergeEntityAsync(CloudTable table, CustomerEntity entity)
{
    if (entity == null)
    {
        throw new ArgumentNullException("entity");
    }

    try
    {
        // Create the InsertOrReplace table operation
        TableOperation insertOrMergeOperation = TableOperation.InsertOrMerge(entity);

        // Execute the operation.
        TableResult result = await table.ExecuteAsync(insertOrMergeOperation);
        CustomerEntity insertedCustomer = result.Result as CustomerEntity;

        if (result.RequestCharge.HasValue)
        {
            Console.WriteLine("Request Charge of InsertOrMerge Operation: " + result.RequestCharge);
        }

        return insertedCustomer;
    }
    catch (StorageException e)
    {
        Console.WriteLine(e.Message);
        Console.ReadLine();
        throw;
    }
}
```

Get an entity from a partition

You can get entity from a partition by using the Retrieve method under the TableOperation class. The following code example gets the partition key row key, email and phone number of a customer entity. This example also prints out the request units consumed to query for the entity. To query for an entity, append the following code to **SamplesUtils.cs** file:

```
public static async Task<CustomerEntity> RetrieveEntityUsingPointQueryAsync(CloudTable table, string
partitionKey, string rowKey)
{
    try
    {
        TableOperation retrieveOperation = TableOperation.Retrieve<CustomerEntity>(partitionKey, rowKey);
        TableResult result = await table.ExecuteAsync(retrieveOperation);
        CustomerEntity customer = result.Result as CustomerEntity;
        if (customer != null)
        {
            Console.WriteLine("\t{0}\t{1}\t{2}\t{3}", customer.PartitionKey, customer.RowKey,
customer.Email, customer.PhoneNumber);
        }

        if (result.RequestCharge.HasValue)
        {
            Console.WriteLine("Request Charge of Retrieve Operation: " + result.RequestCharge);
        }

        return customer;
    }
    catch (StorageException e)
    {
        Console.WriteLine(e.Message);
        Console.ReadLine();
        throw;
    }
}
```

Delete an entity

You can easily delete an entity after you have retrieved it by using the same pattern shown for updating an entity. The following code retrieves and deletes a customer entity. To delete an entity, append the following code to **SamplesUtils.cs** file:

```
public static async Task DeleteEntityAsync(CloudTable table, CustomerEntity deleteEntity)
{
    try
    {
        if (deleteEntity == null)
        {
            throw new ArgumentNullException("deleteEntity");
        }

        TableOperation deleteOperation = TableOperation.Delete(deleteEntity);
        TableResult result = await table.ExecuteAsync(deleteOperation);

        if (result.RequestCharge.HasValue)
        {
            Console.WriteLine("Request Charge of Delete Operation: " + result.RequestCharge);
        }

    }
    catch (StorageException e)
    {
        Console.WriteLine(e.Message);
        Console.ReadLine();
        throw;
    }
}
```

Execute the CRUD operations on sample data

After you define the methods to create table, insert or merge entities, run these methods on the sample data. To do so, right click on your project **CosmosTableSamples**. Select **Add**, **New Item** and add a class named **BasicSamples.cs** and add the following code to it. This code creates a table, adds entities to it.

If don't want to delete the entity and table at the end of the project, comment the `await table.DeleteIfExistsAsync()` and `SamplesUtils.DeleteEntityAsync(table, customer)` methods from the following code. It's best to comment out these methods and validate the data before you delete the table.

```
using System;

namespace CosmosTableSamples
{
    using System.Threading.Tasks;
    using Microsoft.Azure.Cosmos.Table;
    using Model;

    class BasicSamples
    {
        public async Task RunSamples()
        {
            Console.WriteLine("Azure Cosmos DB Table - Basic Samples\n");
            Console.WriteLine();

            string tableName = "demo" + Guid.NewGuid().ToString().Substring(0, 5);

            // Create or reference an existing table
            CloudTable table = await Common.CreateTableAsync(tableName);

            try
            {
                // Demonstrate basic CRUD functionality
                await BasicDataOperationsAsync(table);
            }
            finally
            {
                // Delete the table
                await table.DeleteIfExistsAsync();
            }
        }

        private static async Task BasicDataOperationsAsync(CloudTable table)
        {
            // Create an instance of a customer entity. See the Model\CustomerEntity.cs for a description
of the entity.
            CustomerEntity customer = new CustomerEntity("Harp", "Walter")
            {
                Email = "Walter@contoso.com",
                PhoneNumber = "425-555-0101"
            };

            // Demonstrate how to insert the entity
            Console.WriteLine("Insert an Entity.");
            customer = await SamplesUtils.InsertOrMergeEntityAsync(table, customer);

            // Demonstrate how to Update the entity by changing the phone number
            Console.WriteLine("Update an existing Entity using the InsertOrMerge Upsert Operation.");
            customer.PhoneNumber = "425-555-0105";
            await SamplesUtils.InsertOrMergeEntityAsync(table, customer);
            Console.WriteLine();

            // Demonstrate how to Read the updated entity using a point query
            Console.WriteLine("Reading the updated Entity.");
            customer = await SamplesUtils.RetrieveEntityUsingPointQueryAsync(table, "Harp", "Walter");
            Console.WriteLine();

            // Demonstrate how to Delete an entity
            Console.WriteLine("Delete the entity. ");
            await SamplesUtils.DeleteEntityAsync(table, customer);
            Console.WriteLine();
        }
    }
}
```

The previous code creates a table that starts with “demo” and the generated GUID is appended to the table

name. It then adds a customer entity with first and last name as "Harp Walter" and later updates the phone number of this user.

In this tutorial, you built code to perform basic CRUD operations on the data stored in Table API account. You can also perform advanced operations such as – batch inserting data, query all the data within a partition, query a range of data within a partition, Lists tables in the account whose names begin with the specified prefix. You can download the complete sample form azure-cosmos-table-dotnet-core-getting-started GitHub repository. The AdvancedSamples.cs class has more operations that you can perform on the data.

Run the project

From your project **CosmosTableSamples**. Open the class named **Program.cs** and add the following code to it for calling BasicSamples when the project runs.

```
using System;

namespace CosmosTableSamples
{
    class Program
    {
        public static void Main(string[] args)
        {
            Console.WriteLine("Azure Cosmos Table Samples");
            BasicSamples basicSamples = new BasicSamples();
            basicSamples.RunSamples().Wait();

            AdvancedSamples advancedSamples = new AdvancedSamples();
            advancedSamples.RunSamples().Wait();

            Console.WriteLine();
            Console.WriteLine("Press any key to exit");
            Console.Read();
        }
    }
}
```

Now build the solution and press F5 to run the project. When the project is run, you will see the following output in the command prompt:

If you receive an error that says Settings.json file can't be found when running the project, you can resolve it by adding the following XML entry to the project settings. Right click on CosmosTableSamples, select Edit CosmosTableSamples.csproj and add the following itemGroup:

```
<ItemGroup>
  <None Update="Settings.json">
    <CopyToOutputDirectory>PreserveNewest</CopyToOutputDirectory>
  </None>
</ItemGroup>
```

Now you can sign into the Azure portal and verify that the data exists in the table.

Next steps

You can now proceed to the next tutorial and learn how to migrate data to Azure Cosmos DB Table API account.

Migrate data to Azure Comsos DB Table API

How to use Azure Table storage or Azure Cosmos DB Table API from Java

11/2/2020 • 17 minutes to read • Edit Online

APPLIES TO: Table API

> **TIP**
> The content in this article applies to Azure Table storage and the Azure Cosmos DB Table API. The Azure Cosmos DB Table API is a premium offering for table storage that offers throughput-optimized tables, global distribution, and automatic secondary indexes.

This article shows you how to create tables, store your data, and perform CRUD operations on the data. Choose either the Azure Table service or the Azure Cosmos DB Table API. The samples are written in Java and use the Azure Storage SDK for Java. The scenarios covered include **creating**, **listing**, and **deleting** tables, as well as **inserting**, **querying**, **modifying**, and **deleting** entities in a table. For more information on tables, see the Next steps section.

> **NOTE**
> An SDK is available for developers who are using Azure Storage on Android devices. For more information, see the Azure Storage SDK for Android.

Create an Azure service account

You can work with tables using the Azure Table storage or the Azure Cosmos DB. To learn more about the differences between table offerings in these two services, see the Table offerings article. You'll need to create an account for the service you're going to use. The following sections show how to create both Azure Table storage and the Azure Cosmos DB account, however you can just use one of them.

Create an Azure storage account

The easiest way to create an Azure storage account is by using the Azure portal. To learn more, see Create a storage account.

You can also create an Azure storage account by using Azure PowerShell or Azure CLI.

If you prefer not to create a storage account at this time, you can also use the Azure Storage Emulator to run and test your code in a local environment. For more information, see Use the Azure Storage Emulator for development and testing.

Create an Azure Cosmos DB account

For instructions on creating an Azure Cosmos DB Table API account, see Create a database account.

Create a Java application

In this guide, you will use storage features that you can run in a Java application locally, or in code running in a web role or worker role in Azure.

To use the samples in this article, install the Java Development Kit (JDK), then create an Azure storage account or

Azure Cosmos DB account in your Azure subscription. Once you have done so, verify that your development system meets the minimum requirements and dependencies that are listed in the Azure Storage SDK for Java repository on GitHub. If your system meets those requirements, you can follow the instructions to download and install the Azure Storage Libraries for Java on your system from that repository. After you complete those tasks, you can create a Java application that uses the examples in this article.

Configure your application to access table storage

Add the following import statements to the top of the Java file where you want to use Azure storage APIs or the Azure Cosmos DB Table API to access tables:

```
// Include the following imports to use table APIs
import com.microsoft.azure.storage.*;
import com.microsoft.azure.storage.table.*;
import com.microsoft.azure.storage.table.TableQuery.*;
```

Add your connection string

You can either connect to the Azure storage account or the Azure Cosmos DB Table API account. Get the connection string based on the type of account you are using.

Add an Azure storage connection string

An Azure storage client uses a storage connection string to store endpoints and credentials for accessing data management services. When running in a client application, you must provide the storage connection string in the following format, using the name of your storage account and the Primary access key for the storage account listed in the Azure portal for the **AccountName** and **AccountKey** values.

This example shows how you can declare a static field to hold the connection string:

```
// Define the connection-string with your values.
public static final String storageConnectionString =
    "DefaultEndpointsProtocol=http;" +
    "AccountName=your_storage_account;" +
    "AccountKey=your_storage_account_key";
```

Add an Azure Cosmos DB Table API connection string

An Azure Cosmos DB account uses a connection string to store the table endpoint and your credentials. When running in a client application, you must provide the Azure Cosmos DB connection string in the following format, using the name of your Azure Cosmos DB account and the primary access key for the account listed in the Azure portal for the **AccountName** and **AccountKey** values.

This example shows how you can declare a static field to hold the Azure Cosmos DB connection string:

```
public static final String storageConnectionString =
    "DefaultEndpointsProtocol=https;" +
    "AccountName=your_cosmosdb_account;" +
    "AccountKey=your_account_key;" +
    "TableEndpoint=https://your_endpoint;" ;
```

In an application running within a role in Azure, you can store this string in the service configuration file, *ServiceConfiguration.cscfg*, and you can access it with a call to the **RoleEnvironment.getConfigurationSettings** method. Here's an example of getting the connection string from a **Setting** element named *StorageConnectionString* in the service configuration file:

```
// Retrieve storage account from connection-string.
String storageConnectionString =
    RoleEnvironment.getConfigurationSettings().get("StorageConnectionString");
```

You can also store your connection string in your project's config.properties file:

```
StorageConnectionString =
DefaultEndpointsProtocol=https;AccountName=your_account;AccountKey=your_account_key;TableEndpoint=https://you
r_table_endpoint/
```

The following samples assume that you have used one of these methods to get the storage connection string.

Create a table

A `CloudTableClient` object lets you get reference objects for tables and entities. The following code creates a `CloudTableClient` object and uses it to create a new `CloudTable` object, which represents a table named "people".

> **NOTE**
> There are other ways to create `CloudStorageAccount` objects; for more information, see `CloudStorageAccount` in the Azure Storage Client SDK Reference).

```
try
{
    // Retrieve storage account from connection-string.
    CloudStorageAccount storageAccount =
        CloudStorageAccount.parse(storageConnectionString);

    // Create the table client.
    CloudTableClient tableClient = storageAccount.createCloudTableClient();

    // Create the table if it doesn't exist.
    String tableName = "people";
    CloudTable cloudTable = tableClient.getTableReference(tableName);
    cloudTable.createIfNotExists();
}
catch (Exception e)
{
    // Output the stack trace.
    e.printStackTrace();
}
```

List the tables

To get a list of tables, call the **CloudTableClient.listTables()** method to retrieve an iterable list of table names.

```
try
{
    // Retrieve storage account from connection-string.
    CloudStorageAccount storageAccount =
        CloudStorageAccount.parse(storageConnectionString);

    // Create the table client.
    CloudTableClient tableClient = storageAccount.createCloudTableClient();

    // Loop through the collection of table names.
    for (String table : tableClient.listTables())
    {
        // Output each table name.
        System.out.println(table);
    }
}
catch (Exception e)
{
    // Output the stack trace.
    e.printStackTrace();
}
```

Add an entity to a table

Entities map to Java objects using a custom class implementing `TableEntity`. For convenience, the `TableServiceEntity` class implements `TableEntity` and uses reflection to map properties to getter and setter methods named for the properties. To add an entity to a table, first create a class that defines the properties of your entity. The following code defines an entity class that uses the customer's first name as the row key, and last name as the partition key. Together, an entity's partition and row key uniquely identify the entity in the table. Entities with the same partition key can be queried faster than those with different partition keys.

```
public class CustomerEntity extends TableServiceEntity {
    public CustomerEntity(String lastName, String firstName) {
        this.partitionKey = lastName;
        this.rowKey = firstName;
    }

    public CustomerEntity() { }

    String email;
    String phoneNumber;

    public String getEmail() {
        return this.email;
    }

    public void setEmail(String email) {
        this.email = email;
    }

    public String getPhoneNumber() {
        return this.phoneNumber;
    }

    public void setPhoneNumber(String phoneNumber) {
        this.phoneNumber = phoneNumber;
    }
}
```

Table operations involving entities require a `TableOperation` object. This object defines the operation to be performed on an entity, which can be executed with a `CloudTable` object. The following code creates a new

instance of the `CustomerEntity` class with some customer data to be stored. The code next calls `TableOperation` .insertOrReplace** to create a `TableOperation` object to insert an entity into a table, and associates the new `CustomerEntity` with it. Finally, the code calls the `execute` method on the `CloudTable` object, specifying the "people" table and the new `TableOperation`, which then sends a request to the storage service to insert the new customer entity into the "people" table, or replace the entity if it already exists.

```
try
{
    // Retrieve storage account from connection-string.
    CloudStorageAccount storageAccount =
        CloudStorageAccount.parse(storageConnectionString);

    // Create the table client.
    CloudTableClient tableClient = storageAccount.createCloudTableClient();

    // Create a cloud table object for the table.
    CloudTable cloudTable = tableClient.getTableReference("people");

    // Create a new customer entity.
    CustomerEntity customer1 = new CustomerEntity("Harp", "Walter");
    customer1.setEmail("Walter@contoso.com");
    customer1.setPhoneNumber("425-555-0101");

    // Create an operation to add the new customer to the people table.
    TableOperation insertCustomer1 = TableOperation.insertOrReplace(customer1);

    // Submit the operation to the table service.
    cloudTable.execute(insertCustomer1);
}
catch (Exception e)
{
    // Output the stack trace.
    e.printStackTrace();
}
```

Insert a batch of entities

You can insert a batch of entities to the table service in one write operation. The following code creates a `TableBatchOperation` object, then adds three insert operations to it. Each insert operation is added by creating a new entity object, setting its values, and then calling the `insert` method on the `TableBatchOperation` object to associate the entity with a new insert operation. Then the code calls `execute` on the `CloudTable` object, specifying the "people" table and the `TableBatchOperation` object, which sends the batch of table operations to the storage service in a single request.

```
try
{
    // Retrieve storage account from connection-string.
    CloudStorageAccount storageAccount =
        CloudStorageAccount.parse(storageConnectionString);

    // Create the table client.
    CloudTableClient tableClient = storageAccount.createCloudTableClient();

    // Define a batch operation.
    TableBatchOperation batchOperation = new TableBatchOperation();

    // Create a cloud table object for the table.
    CloudTable cloudTable = tableClient.getTableReference("people");

    // Create a customer entity to add to the table.
    CustomerEntity customer = new CustomerEntity("Smith", "Jeff");
    customer.setEmail("Jeff@contoso.com");
    customer.setPhoneNumber("425-555-0104");
    batchOperation.insertOrReplace(customer);

    // Create another customer entity to add to the table.
    CustomerEntity customer2 = new CustomerEntity("Smith", "Ben");
    customer2.setEmail("Ben@contoso.com");
    customer2.setPhoneNumber("425-555-0102");
    batchOperation.insertOrReplace(customer2);

    // Create a third customer entity to add to the table.
    CustomerEntity customer3 = new CustomerEntity("Smith", "Denise");
    customer3.setEmail("Denise@contoso.com");
    customer3.setPhoneNumber("425-555-0103");
    batchOperation.insertOrReplace(customer3);

    // Execute the batch of operations on the "people" table.
    cloudTable.execute(batchOperation);
}
catch (Exception e)
{
    // Output the stack trace.
    e.printStackTrace();
}
```

Some things to note on batch operations:

- You can perform up to 100 insert, delete, merge, replace, insert or merge, and insert or replace operations in any combination in a single batch.
- A batch operation can have a retrieve operation, if it is the only operation in the batch.
- All entities in a single batch operation must have the same partition key.
- A batch operation is limited to a 4-MB data payload.

Retrieve all entities in a partition

To query a table for entities in a partition, you can use a `TableQuery`. Call `TableQuery.from` to create a query on a particular table that returns a specified result type. The following code specifies a filter for entities where 'Smith' is the partition key. `TableQuery.generateFilterCondition` is a helper method to create filters for queries. Call `where` on the reference returned by the `TableQuery.from` method to apply the filter to the query. When the query is executed with a call to `execute` on the `CloudTable` object, it returns an `Iterator` with the `CustomerEntity` result type specified. You can then use the `Iterator` returned in a "ForEach" loop to consume the results. This code prints the fields of each entity in the query results to the console.

```
try
{
    // Define constants for filters.
    final String PARTITION_KEY = "PartitionKey";
    final String ROW_KEY = "RowKey";
    final String TIMESTAMP = "Timestamp";

    // Retrieve storage account from connection-string.
    CloudStorageAccount storageAccount =
        CloudStorageAccount.parse(storageConnectionString);

    // Create the table client.
    CloudTableClient tableClient = storageAccount.createCloudTableClient();

    // Create a cloud table object for the table.
    CloudTable cloudTable = tableClient.getTableReference("people");

    // Create a filter condition where the partition key is "Smith".
    String partitionFilter = TableQuery.generateFilterCondition(
        PARTITION_KEY,
        QueryComparisons.EQUAL,
        "Smith");

    // Specify a partition query, using "Smith" as the partition key filter.
    TableQuery<CustomerEntity> partitionQuery =
        TableQuery.from(CustomerEntity.class)
        .where(partitionFilter);

    // Loop through the results, displaying information about the entity.
    for (CustomerEntity entity : cloudTable.execute(partitionQuery)) {
        System.out.println(entity.getPartitionKey() +
            " " + entity.getRowKey() +
            "\t" + entity.getEmail() +
            "\t" + entity.getPhoneNumber());
    }
}
catch (Exception e)
{
    // Output the stack trace.
    e.printStackTrace();
}
```

Retrieve a range of entities in a partition

If you don't want to query all the entities in a partition, you can specify a range by using comparison operators in a filter. The following code combines two filters to get all entities in partition "Smith" where the row key (first name) starts with a letter up to 'E' in the alphabet. Then it prints the query results. If you use the entities added to the table in the batch insert section of this guide, only two entities are returned this time (Ben and Denise Smith); Jeff Smith is not included.

```
try
{
    // Define constants for filters.
    final String PARTITION_KEY = "PartitionKey";
    final String ROW_KEY = "RowKey";
    final String TIMESTAMP = "Timestamp";

    // Retrieve storage account from connection-string.
    CloudStorageAccount storageAccount =
        CloudStorageAccount.parse(storageConnectionString);

    // Create the table client.
    CloudTableClient tableClient = storageAccount.createCloudTableClient();

    // Create a cloud table object for the table.
    CloudTable cloudTable = tableClient.getTableReference("people");

    // Create a filter condition where the partition key is "Smith".
    String partitionFilter = TableQuery.generateFilterCondition(
        PARTITION_KEY,
        QueryComparisons.EQUAL,
        "Smith");

    // Create a filter condition where the row key is less than the letter "E".
    String rowFilter = TableQuery.generateFilterCondition(
        ROW_KEY,
        QueryComparisons.LESS_THAN,
        "E");

    // Combine the two conditions into a filter expression.
    String combinedFilter = TableQuery.combineFilters(partitionFilter,
        Operators.AND, rowFilter);

    // Specify a range query, using "Smith" as the partition key,
    // with the row key being up to the letter "E".
    TableQuery<CustomerEntity> rangeQuery =
        TableQuery.from(CustomerEntity.class)
        .where(combinedFilter);

    // Loop through the results, displaying information about the entity
    for (CustomerEntity entity : cloudTable.execute(rangeQuery)) {
        System.out.println(entity.getPartitionKey() +
            " " + entity.getRowKey() +
            "\t" + entity.getEmail() +
            "\t" + entity.getPhoneNumber());
    }
}
catch (Exception e)
{
    // Output the stack trace.
    e.printStackTrace();
}
```

Retrieve a single entity

You can write a query to retrieve a single, specific entity. The following code calls `TableOperation.retrieve` with partition key and row key parameters to specify the customer "Jeff Smith", instead of creating a `Table Query` and using filters to do the same thing. When executed, the retrieve operation returns just one entity, rather than a collection. The `getResultAsType` method casts the result to the type of the assignment target, a `CustomerEntity` object. If this type is not compatible with the type specified for the query, an exception is thrown. A null value is returned if no entity has an exact partition and row key match. Specifying both partition and row keys in a query is the fastest way to retrieve a single entity from the Table service.

```
try
{
    // Retrieve storage account from connection-string.
    CloudStorageAccount storageAccount =
        CloudStorageAccount.parse(storageConnectionString);

    // Create the table client.
    CloudTableClient tableClient = storageAccount.createCloudTableClient();

    // Create a cloud table object for the table.
    CloudTable cloudTable = tableClient.getTableReference("people");

    // Retrieve the entity with partition key of "Smith" and row key of "Jeff"
    TableOperation retrieveSmithJeff =
        TableOperation.retrieve("Smith", "Jeff", CustomerEntity.class);

    // Submit the operation to the table service and get the specific entity.
    CustomerEntity specificEntity =
        cloudTable.execute(retrieveSmithJeff).getResultAsType();

    // Output the entity.
    if (specificEntity != null)
    {
        System.out.println(specificEntity.getPartitionKey() +
            " " + specificEntity.getRowKey() +
            "\t" + specificEntity.getEmail() +
            "\t" + specificEntity.getPhoneNumber());
    }
}
catch (Exception e)
{
    // Output the stack trace.
    e.printStackTrace();
}
```

Modify an entity

To modify an entity, retrieve it from the table service, make changes to the entity object, and save the changes back to the table service with a replace or merge operation. The following code changes an existing customer's phone number. Instead of calling **TableOperation.insert** as we did to insert, this code calls **TableOperation.replace**. The **CloudTable.execute** method calls the table service, and the entity is replaced, unless another application changed it in the time since this application retrieved it. When that happens, an exception is thrown, and the entity must be retrieved, modified, and saved again. This optimistic concurrency retry pattern is common in a distributed storage system.

```
try
{
    // Retrieve storage account from connection-string.
    CloudStorageAccount storageAccount =
        CloudStorageAccount.parse(storageConnectionString);

    // Create the table client.
    CloudTableClient tableClient = storageAccount.createCloudTableClient();

    // Create a cloud table object for the table.
    CloudTable cloudTable = tableClient.getTableReference("people");

    // Retrieve the entity with partition key of "Smith" and row key of "Jeff".
    TableOperation retrieveSmithJeff =
        TableOperation.retrieve("Smith", "Jeff", CustomerEntity.class);

    // Submit the operation to the table service and get the specific entity.
    CustomerEntity specificEntity =
        cloudTable.execute(retrieveSmithJeff).getResultAsType();

    // Specify a new phone number.
    specificEntity.setPhoneNumber("425-555-0105");

    // Create an operation to replace the entity.
    TableOperation replaceEntity = TableOperation.replace(specificEntity);

    // Submit the operation to the table service.
    cloudTable.execute(replaceEntity);
}
catch (Exception e)
{
    // Output the stack trace.
    e.printStackTrace();
}
```

Query a subset of entity properties

A query to a table can retrieve just a few properties from an entity. This technique, called projection, reduces bandwidth and can improve query performance, especially for large entities. The query in the following code uses the `select` method to return only the email addresses of entities in the table. The results are projected into a collection of `String` with the help of an `Entity Resolver`, which does the type conversion on the entities returned from the server. You can learn more about projection in [Azure Tables: Introducing Upsert and Query Projection][Azure Tables: Introducing Upsert and Query Projection]. The projection is not supported on the local storage emulator, so this code runs only when using an account on the table service.

```
try
{
    // Retrieve storage account from connection-string.
    CloudStorageAccount storageAccount =
        CloudStorageAccount.parse(storageConnectionString);

    // Create the table client.
    CloudTableClient tableClient = storageAccount.createCloudTableClient();

    // Create a cloud table object for the table.
    CloudTable cloudTable = tableClient.getTableReference("people");

    // Define a projection query that retrieves only the Email property
    TableQuery<CustomerEntity> projectionQuery =
        TableQuery.from(CustomerEntity.class)
        .select(new String[] {"Email"});

    // Define an Entity resolver to project the entity to the Email value.
    EntityResolver<String> emailResolver = new EntityResolver<String>() {
        @Override
        public String resolve(String PartitionKey, String RowKey, Date timeStamp, HashMap<String,
EntityProperty> properties, String etag) {
            return properties.get("Email").getValueAsString();
        }
    };

    // Loop through the results, displaying the Email values.
    for (String projectedString :
        cloudTable.execute(projectionQuery, emailResolver)) {
            System.out.println(projectedString);
    }
}
catch (Exception e)
{
    // Output the stack trace.
    e.printStackTrace();
}
```

Insert or Replace an entity

Often you want to add an entity to a table without knowing if it already exists in the table. An insert-or-replace operation allows you to make a single request, which will insert the entity if it does not exist or replace the existing one if it does. Building on prior examples, the following code inserts or replaces the entity for "Walter Harp". After creating a new entity, this code calls the **TableOperation.insertOrReplace** method. This code then calls **execute** on the **Cloud Table** object with the table and the insert or replace table operation as the parameters. To update only part of an entity, the **TableOperation.insertOrMerge** method can be used instead. Insert-or-replace is not supported on the local storage emulator, so this code runs only when using an account on the table service. You can learn more about insert-or-replace and insert-or-merge in this [Azure Tables: Introducing Upsert and Query Projection][Azure Tables: Introducing Upsert and Query Projection].

```
try
{
    // Retrieve storage account from connection-string.
    CloudStorageAccount storageAccount =
        CloudStorageAccount.parse(storageConnectionString);

    // Create the table client.
    CloudTableClient tableClient = storageAccount.createCloudTableClient();

    // Create a cloud table object for the table.
    CloudTable cloudTable = tableClient.getTableReference("people");

    // Create a new customer entity.
    CustomerEntity customer5 = new CustomerEntity("Harp", "Walter");
    customer5.setEmail("Walter@contoso.com");
    customer5.setPhoneNumber("425-555-0106");

    // Create an operation to add the new customer to the people table.
    TableOperation insertCustomer5 = TableOperation.insertOrReplace(customer5);

    // Submit the operation to the table service.
    cloudTable.execute(insertCustomer5);
}
catch (Exception e)
{
    // Output the stack trace.
    e.printStackTrace();
}
```

Delete an entity

You can easily delete an entity after you have retrieved it. After the entity is retrieved, call `TableOperation.delete` with the entity to delete. Then call `execute` on the `CloudTable` object. The following code retrieves and deletes a customer entity.

```
try
{
    // Retrieve storage account from connection-string.
    CloudStorageAccount storageAccount =
        CloudStorageAccount.parse(storageConnectionString);

    // Create the table client.
    CloudTableClient tableClient = storageAccount.createCloudTableClient();

    // Create a cloud table object for the table.
    CloudTable cloudTable = tableClient.getTableReference("people");

    // Create an operation to retrieve the entity with partition key of "Smith" and row key of "Jeff".
    TableOperation retrieveSmithJeff = TableOperation.retrieve("Smith", "Jeff", CustomerEntity.class);

    // Retrieve the entity with partition key of "Smith" and row key of "Jeff".
    CustomerEntity entitySmithJeff =
        cloudTable.execute(retrieveSmithJeff).getResultAsType();

    // Create an operation to delete the entity.
    TableOperation deleteSmithJeff = TableOperation.delete(entitySmithJeff);

    // Submit the delete operation to the table service.
    cloudTable.execute(deleteSmithJeff);
}
catch (Exception e)
{
    // Output the stack trace.
    e.printStackTrace();
}
```

Delete a table

Finally, the following code deletes a table from a storage account. Around 40 seconds after you delete a table, you cannot recreate it.

```
try
{
    // Retrieve storage account from connection-string.
    CloudStorageAccount storageAccount =
        CloudStorageAccount.parse(storageConnectionString);

    // Create the table client.
    CloudTableClient tableClient = storageAccount.createCloudTableClient();

    // Delete the table and all its data if it exists.
    CloudTable cloudTable = tableClient.getTableReference("people");
    cloudTable.deleteIfExists();
}
catch (Exception e)
{
    // Output the stack trace.
    e.printStackTrace();
}
```

TIP

Check out the Azure Storage code samples repository

For easy-to-use end-to-end Azure Storage code samples that you can download and run, please check out our list of Azure Storage Samples.

Next steps

- Getting Started with Azure Table Service in Java
- Microsoft Azure Storage Explorer is a free, standalone app from Microsoft that enables you to work visually with Azure Storage data on Windows, macOS, and Linux.
- Azure Storage SDK for Java
- Azure Storage Client SDK Reference
- Azure Storage REST API
- Azure Storage Team Blog

For more information, visit Azure for Java developers.

How to use Azure Table storage or the Azure Cosmos DB Table API from Node.js

11/2/2020 • 13 minutes to read • Edit Online

APPLIES TO: Table API

> **TIP**
>
> The content in this article applies to Azure Table storage and the Azure Cosmos DB Table API. The Azure Cosmos DB Table API is a premium offering for table storage that offers throughput-optimized tables, global distribution, and automatic secondary indexes.

This article shows you how to create tables, store your data, and perform CRUD operations on the data. Choose either the Azure Table service or the Azure Cosmos DB Table API. The samples are written in Node.js.

Create an Azure service account

You can work with tables using the Azure Table storage or the Azure Cosmos DB. To learn more about the differences between table offerings in these two services, see the Table offerings article. You'll need to create an account for the service you're going to use. The following sections show how to create both Azure Table storage and the Azure Cosmos DB account, however you can just use one of them.

Create an Azure storage account

The easiest way to create an Azure storage account is by using the Azure portal. To learn more, see Create a storage account.

You can also create an Azure storage account by using Azure PowerShell or Azure CLI.

If you prefer not to create a storage account at this time, you can also use the Azure Storage Emulator to run and test your code in a local environment. For more information, see Use the Azure Storage Emulator for development and testing.

Create an Azure Cosmos DB Table API account

For instructions on creating an Azure Cosmos DB Table API account, see Create a database account.

Configure your application to access Azure Storage or the Azure Cosmos DB Table API

To use Azure Storage or Azure Cosmos DB, you need the Azure Storage SDK for Node.js, which includes a set of convenience libraries that communicate with the Storage REST services.

Use Node Package Manager (NPM) to install the package

1. Use a command-line interface such as **PowerShell** (Windows), **Terminal** (Mac), or **Bash** (Unix), and navigate to the folder where you created your application.

2. Type **npm install azure-storage** in the command window. Output from the command is similar to the following example.

```
 azure-storage@0.5.0 node_modules\azure-storage
 +-- extend@1.2.1
 +-- xmlbuilder@0.4.3
 +-- mime@1.2.11
 +-- node-uuid@1.4.3
 +-- validator@3.22.2
 +-- underscore@1.4.4
 +-- readable-stream@1.0.33 (string_decoder@0.10.31, isarray@0.0.1, inherits@2.0.1, core-util-
is@1.0.1)
 +-- xml2js@0.2.7 (sax@0.5.2)
 +-- request@2.57.0 (caseless@0.10.0, aws-sign2@0.5.0, forever-agent@0.6.1, stringstream@0.0.4, oauth-
sign@0.8.0, tunnel-agent@0.4.1, isstream@0.1.2, json-stringify-safe@5.0.1, bl@0.9.4, combined-
stream@1.0.5, qs@3.1.0, mime-types@2.0.14, form-data@0.2.0, http-signature@0.11.0, tough-cookie@2.0.0,
hawk@2.3.1, har-validator@1.8.0)
```

3. You can manually run the **ls** command to verify that a **node_modules** folder was created. Inside that folder you will find the **azure-storage** package, which contains the libraries you need to access storage.

Import the package

Add the following code to the top of the **server.js** file in your application:

```
var azure = require('azure-storage');
```

Add your connection string

You can either connect to the Azure storage account or the Azure Cosmos DB Table API account. Get the connection string based on the type of account you are using.

Add an Azure Storage connection

The Azure module reads the environment variables AZURE_STORAGE_ACCOUNT and AZURE_STORAGE_ACCESS_KEY, or AZURE_STORAGE_CONNECTION_STRING for information required to connect to your Azure Storage account. If these environment variables are not set, you must specify the account information when calling `TableService`. For example, the following code creates a `TableService` object:

```
var tableSvc = azure.createTableService('myaccount', 'myaccesskey');
```

Add an Azure Cosmos DB connection

To add an Azure Cosmos DB connection, create a `TableService` object and specify your account name, primary key, and endpoint. You can copy these values from **Settings** > **Connection String** in the Azure portal for your Cosmos DB account. For example:

```
var tableSvc = azure.createTableService('myaccount', 'myprimarykey', 'myendpoint');
```

Create a table

The following code creates a `TableService` object and uses it to create a new table.

```
var tableSvc = azure.createTableService();
```

The call to `createTableIfNotExists` creates a new table with the specified name if it does not already exist. The following example creates a new table named 'mytable' if it does not already exist:

```
tableSvc.createTableIfNotExists('mytable', function(error, result, response){
  if(!error){
    // Table exists or created
  }
});
```

The `result.created` is `true` if a new table is created, and `false` if the table already exists. The `response` contains information about the request.

Apply filters

You can apply optional filtering to operations performed using `TableService`. Filtering operations can include logging, automatic retries, etc. Filters are objects that implement a method with the signature:

```
function handle (requestOptions, next)
```

After doing its preprocessing on the request options, the method must call **next**, passing a callback with the following signature:

```
function (returnObject, finalCallback, next)
```

In this callback, and after processing the `returnObject` (the response from the request to the server), the callback must either invoke `next` if it exists to continue processing other filters or simply invoke `finalCallback` otherwise to end the service invocation.

Two filters that implement retry logic are included with the Azure SDK for Node.js, `ExponentialRetryPolicyFilter` and `LinearRetryPolicyFilter`. The following creates a `TableService` object that uses the `ExponentialRetryPolicyFilter`:

```
var retryOperations = new azure.ExponentialRetryPolicyFilter();
var tableSvc = azure.createTableService().withFilter(retryOperations);
```

Add an entity to a table

To add an entity, first create an object that defines your entity properties. All entities must contain a **PartitionKey** and **RowKey**, which are unique identifiers for the entity.

- **PartitionKey** - Determines the partition in which the entity is stored.
- **RowKey** - Uniquely identifies the entity within the partition.

Both **PartitionKey** and **RowKey** must be string values. For more information, see Understanding the Table Service Data Model.

The following is an example of defining an entity. The **dueDate** is defined as a type of `Edm.DateTime`. Specifying the type is optional, and types are inferred if not specified.

```
var task = {
  PartitionKey: {'_':'hometasks'},
  RowKey: {'_': '1'},
  description: {'_':'take out the trash'},
  dueDate: {'_':new Date(2015, 6, 20), '$':'Edm.DateTime'}
};
```

> **NOTE**
> There is also a `Timestamp` field for each record, which is set by Azure when an entity is inserted or updated.

You can also use the `entityGenerator` to create entities. The following example creates the same task entity using the `entityGenerator`.

```
var entGen = azure.TableUtilities.entityGenerator;
var task = {
  PartitionKey: entGen.String('hometasks'),
  RowKey: entGen.String('1'),
  description: entGen.String('take out the trash'),
  dueDate: entGen.DateTime(new Date(Date.UTC(2015, 6, 20))),
};
```

To add an entity to your table, pass the entity object to the `insertEntity` method.

```
tableSvc.insertEntity('mytable',task, function (error, result, response) {
  if(!error){
    // Entity inserted
  }
});
```

If the operation is successful, `result` contains the ETag of the inserted record and `response` contains information about the operation.

Example response:

```
{ '.metadata': { etag: 'W/"datetime\'2015-02-25T01%3A22%3A22.5Z\'"' } }
```

> **NOTE**
> By default, `insertEntity` does not return the inserted entity as part of the `response` information. If you plan on performing other operations on this entity, or want to cache the information, it can be useful to have it returned as part of the `result`. You can do this by enabling `echoContent` as follows:
>
> `tableSvc.insertEntity('mytable', task, {echoContent: true}, function (error, result, response) {...}`

Update an entity

There are multiple methods available to update an existing entity:

- `replaceEntity` - Updates an existing entity by replacing it.
- `mergeEntity` - Updates an existing entity by merging new property values into the existing entity.
- `insertOrReplaceEntity` - Updates an existing entity by replacing it. If no entity exists, a new one will be inserted.
- `insertOrMergeEntity` - Updates an existing entity by merging new property values into the existing. If no entity exists, a new one will be inserted.

The following example demonstrates updating an entity using `replaceEntity`:

```
tableSvc.replaceEntity('mytable', updatedTask, function(error, result, response){
  if(!error) {
    // Entity updated
  }
});
```

NOTE

By default, updating an entity does not check to see if the data being updated has previously been modified by another process. To support concurrent updates:

1. Get the ETag of the object being updated. This is returned as part of the `response` for any entity-related operation and can be retrieved through `response['.metadata'].etag`.
2. When performing an update operation on an entity, add the ETag information previously retrieved to the new entity. For example:

   entity2['.metadata'].etag = currentEtag;

3. Perform the update operation. If the entity has been modified since you retrieved the ETag value, such as another instance of your application, an `error` is returned stating that the update condition specified in the request was not satisfied.

With `replaceEntity` and `mergeEntity`, if the entity that is being updated doesn't exist, then the update operation fails; therefore, if you want to store an entity regardless of whether it already exists, use `insertOrReplaceEntity` or `insertOrMergeEntity`.

The `result` for successful update operations contains the **Etag** of the updated entity.

Work with groups of entities

Sometimes it makes sense to submit multiple operations together in a batch to ensure atomic processing by the server. To accomplish that, use the `TableBatch` class to create a batch, and then use the `executeBatch` method of `TableService` to perform the batched operations.

The following example demonstrates submitting two entities in a batch:

```
var task1 = {
  PartitionKey: {'_':'hometasks'},
  RowKey: {'_': '1'},
  description: {'_':'Take out the trash'},
  dueDate: {'_':new Date(2015, 6, 20)}
};
var task2 = {
  PartitionKey: {'_':'hometasks'},
  RowKey: {'_': '2'},
  description: {'_':'Wash the dishes'},
  dueDate: {'_':new Date(2015, 6, 20)}
};

var batch = new azure.TableBatch();

batch.insertEntity(task1, {echoContent: true});
batch.insertEntity(task2, {echoContent: true});

tableSvc.executeBatch('mytable', batch, function (error, result, response) {
  if(!error) {
    // Batch completed
  }
});
```

For successful batch operations, `result` contains information for each operation in the batch.

Work with batched operations

You can inspect operations added to a batch by viewing the `operations` property. You can also use the following methods to work with operations:

- **clear** - Clears all operations from a batch.
- **getOperations** - Gets an operation from the batch.
- **hasOperations** - Returns true if the batch contains operations.
- **removeOperations** - Removes an operation.
- **size** - Returns the number of operations in the batch.

Retrieve an entity by key

To return a specific entity based on the **PartitionKey** and **RowKey**, use the **retrieveEntity** method.

```
tableSvc.retrieveEntity('mytable', 'hometasks', '1', function(error, result, response){
  if(!error){
    // result contains the entity
  }
});
```

After this operation is complete, `result` contains the entity.

Query a set of entities

To query a table, use the **TableQuery** object to build up a query expression using the following clauses:

- **select** - The fields to be returned from the query.
- **where** - The where clause.
  - **and** - An `and` where condition.
  - **or** - An `or` where condition.

- **top** - The number of items to fetch.

The following example builds a query that returns the top five items with a PartitionKey of 'hometasks'.

```
var query = new azure.TableQuery()
  .top(5)
  .where('PartitionKey eq ?', 'hometasks');
```

Because **select** is not used, all fields are returned. To perform the query against a table, use **queryEntities**. The following example uses this query to return entities from 'mytable'.

```
tableSvc.queryEntities('mytable',query, null, function(error, result, response) {
  if(!error) {
    // query was successful
  }
});
```

If successful, `result.entries` contains an array of entities that match the query. If the query was unable to return all entities, `result.continuationToken` is non-*null* and can be used as the third parameter of **queryEntities** to retrieve more results. For the initial query, use *null* for the third parameter.

Query a subset of entity properties

A query to a table can retrieve just a few fields from an entity. This reduces bandwidth and can improve query performance, especially for large entities. Use the **select** clause and pass the names of the fields to return. For example, the following query returns only the **description** and **dueDate** fields.

```
var query = new azure.TableQuery()
  .select(['description', 'dueDate'])
  .top(5)
  .where('PartitionKey eq ?', 'hometasks');
```

Delete an entity

You can delete an entity using its partition and row keys. In this example, the **task1** object contains the **RowKey** and **PartitionKey** values of the entity to delete. Then the object is passed to the **deleteEntity** method.

```
var task = {
  PartitionKey: {'_':'hometasks'},
  RowKey: {'_': '1'}
};

tableSvc.deleteEntity('mytable', task, function(error, response){
  if(!error) {
    // Entity deleted
  }
});
```

> **NOTE**
>
> Consider using ETags when deleting items, to ensure that the item hasn't been modified by another process. See Update an entity for information on using ETags.

Delete a table

The following code deletes a table from a storage account.

```
tableSvc.deleteTable('mytable', function(error, response){
    if(!error){
        // Table deleted
    }
});
```

If you are uncertain whether the table exists, use **deleteTableIfExists**.

Use continuation tokens

When you are querying tables for large amounts of results, look for continuation tokens. There may be large amounts of data available for your query that you might not realize if you do not build to recognize when a continuation token is present.

The **results** object returned during querying entities sets a `continuationToken` property when such a token is present. You can then use this when performing a query to continue to move across the partition and table entities.

When querying, you can provide a `continuationToken` parameter between the query object instance and the callback function:

```
var nextContinuationToken = null;
dc.table.queryEntities(tableName,
    query,
    nextContinuationToken,
    function (error, results) {
        if (error) throw error;

        // iterate through results.entries with results

        if (results.continuationToken) {
            nextContinuationToken = results.continuationToken;
        }

    });
```

If you inspect the `continuationToken` object, you will find properties such as `nextPartitionKey`, `nextRowKey` and `targetLocation`, which can be used to iterate through all the results.

You can also use `top` along with `continuationToken` to set the page size.

Work with shared access signatures

Shared access signatures (SAS) are a secure way to provide granular access to tables without providing your Storage account name or keys. SAS are often used to provide limited access to your data, such as allowing a mobile app to query records.

A trusted application such as a cloud-based service generates a SAS using the **generateSharedAccessSignature** of the **TableService**, and provides it to an untrusted or semi-trusted application such as a mobile app. The SAS is generated using a policy, which describes the start and end dates during which the SAS is valid, as well as the access level granted to the SAS holder.

The following example generates a new shared access policy that will allow the SAS holder to query ('r') the table, and expires 100 minutes after the time it is created.

```
var startDate = new Date();
var expiryDate = new Date(startDate);
expiryDate.setMinutes(startDate.getMinutes() + 100);
startDate.setMinutes(startDate.getMinutes() - 100);

var sharedAccessPolicy = {
  AccessPolicy: {
    Permissions: azure.TableUtilities.SharedAccessPermissions.QUERY,
    Start: startDate,
    Expiry: expiryDate
  },
};

var tableSAS = tableSvc.generateSharedAccessSignature('mytable', sharedAccessPolicy);
var host = tableSvc.host;
```

Note that you must also provide the host information, as it is required when the SAS holder attempts to access the table.

The client application then uses the SAS with **TableServiceWithSAS** to perform operations against the table. The following example connects to the table and performs a query. See Grant limited access to Azure Storage resources using shared access signatures (SAS) article for the format of tableSAS.

```
// Note in the following command, host is in the format:
`https://<your_storage_account_name>.table.core.windows.net` and the tableSAS is in the format: `sv=2018-03-
28&si=saspolicy&tn=mytable&sig=9aCzs76n0E7y5BpEi2GvsSv433BZa22leDOZXX%2BXXIU%3D`;

var sharedTableService = azure.createTableServiceWithSas(host, tableSAS);
var query = azure.TableQuery()
  .where('PartitionKey eq ?', 'hometasks');

sharedTableService.queryEntities(query, null, function(error, result, response) {
  if(!error) {
    // result contains the entities
  }
});
```

Because the SAS was generated with only query access, an error is returned if you attempt to insert, update, or delete entities.

Access Control Lists

You can also use an Access Control List (ACL) to set the access policy for a SAS. This is useful if you want to allow multiple clients to access the table, but provide different access policies for each client.

An ACL is implemented using an array of access policies, with an ID associated with each policy. The following example defines two policies, one for 'user1' and one for 'user2':

```
var sharedAccessPolicy = {
  user1: {
    Permissions: azure.TableUtilities.SharedAccessPermissions.QUERY,
    Start: startDate,
    Expiry: expiryDate
  },
  user2: {
    Permissions: azure.TableUtilities.SharedAccessPermissions.ADD,
    Start: startDate,
    Expiry: expiryDate
  }
};
```

The following example gets the current ACL for the **hometasks** table, and then adds the new policies using **setTableAcl**. This approach allows:

```
var extend = require('extend');
tableSvc.getTableAcl('hometasks', function(error, result, response) {
if(!error){
    var newSignedIdentifiers = extend(true, result.signedIdentifiers, sharedAccessPolicy);
    tableSvc.setTableAcl('hometasks', newSignedIdentifiers, function(error, result, response){
      if(!error){
        // ACL set
      }
    });
  }
});
```

After the ACL has been set, you can then create a SAS based on the ID for a policy. The following example creates a new SAS for 'user2':

```
tableSAS = tableSvc.generateSharedAccessSignature('hometasks', { Id: 'user2' });
```

Next steps

For more information, see the following resources.

- Microsoft Azure Storage Explorer is a free, standalone app from Microsoft that enables you to work visually with Azure Storage data on Windows, macOS, and Linux.
- Azure Storage SDK for Node.js repository on GitHub.
- Azure for Node.js Developers
- Create a Node.js web app in Azure
- Build and deploy a Node.js application to an Azure Cloud Service (using Windows PowerShell)

Get started with Azure Table storage and the Azure Cosmos DB Table API using Python

11/2/2020 • 7 minutes to read • Edit Online

APPLIES TO: Table API

> **TIP**
> The content in this article applies to Azure Table storage and the Azure Cosmos DB Table API. The Azure Cosmos DB Table API is a premium offering for table storage that offers throughput-optimized tables, global distribution, and automatic secondary indexes.

The Azure Table storage and the Azure Cosmos DB are services that store structured NoSQL data in the cloud, providing a key/attribute store with a schemaless design. Because Table storage and Azure Cosmos DB are schemaless, it's easy to adapt your data as the needs of your application evolve. Access to the table storage and table API data is fast and cost-effective for many types of applications, and is typically lower in cost than traditional SQL for similar volumes of data.

You can use the Table storage or the Azure Cosmos DB to store flexible datasets like user data for web applications, address books, device information, or other types of metadata your service requires. You can store any number of entities in a table, and a storage account may contain any number of tables, up to the capacity limit of the storage account.

About this sample

This sample shows you how to use the Azure Cosmos DB Table SDK for Python in common Azure Table storage scenarios. The name of the SDK indicates it is for use with Azure Cosmos DB, but it works with both Azure Cosmos DB and Azure Tables storage, each service just has a unique endpoint. These scenarios are explored using Python examples that illustrate how to:

- Create and delete tables
- Insert and query entities
- Modify entities

While working through the scenarios in this sample, you may want to refer to the Azure Cosmos DB SDK for Python API reference.

Prerequisites

You need the following to complete this sample successfully:

- Python 2.7, 3.3, 3.4, 3.5, or 3.6
- Azure Cosmos DB Table SDK for Python. This SDK connects with both Azure Table storage and the Azure Cosmos DB Table API.
- Azure Storage account or Azure Cosmos DB account

Create an Azure service account

You can work with tables using the Azure Table storage or the Azure Cosmos DB. To learn more about the differences between table offerings in these two services, see the Table offerings article. You'll need to create an

account for the service you're going to use. The following sections show how to create both Azure Table storage and the Azure Cosmos DB account, however you can just use one of them.

Create an Azure storage account

The easiest way to create an Azure storage account is by using the Azure portal. To learn more, see Create a storage account.

You can also create an Azure storage account by using Azure PowerShell or Azure CLI.

If you prefer not to create a storage account at this time, you can also use the Azure Storage Emulator to run and test your code in a local environment. For more information, see Use the Azure Storage Emulator for development and testing.

Create an Azure Cosmos DB Table API account

For instructions on creating an Azure Cosmos DB Table API account, see Create a database account.

Install the Azure Cosmos DB Table SDK for Python

After you've created a Storage account, your next step is to install the Microsoft Azure Cosmos DB Table SDK for Python. For details on installing the SDK, refer to the README.rst file in the Cosmos DB Table SDK for Python repository on GitHub.

Import the TableService and Entity classes

To work with entities in the Azure Table service in Python, you use the TableService and Entity classes. Add this code near the top your Python file to import both:

```
from azure.cosmosdb.table.tableservice import TableService
from azure.cosmosdb.table.models import Entity
```

Connect to Azure Table service

To connect to Azure Storage Table service, create a TableService object, and pass in your Storage account name and account key. Replace `myaccount` and `mykey` with your account name and key.

```
table_service = TableService(account_name='myaccount', account_key='mykey')
```

Connect to Azure Cosmos DB

To connect to Azure Cosmos DB, copy your primary connection string from the Azure portal, and create a TableService object using your copied connection string:

```
table_service =
TableService(connection_string='DefaultEndpointsProtocol=https;AccountName=myaccount;AccountKey=mykey;TableE
ndpoint=myendpoint;')
```

Create a table

Call create_table to create the table.

```
table_service.create_table('tasktable')
```

Add an entity to a table

To add an entity, you first create an object that represents your entity, then pass the object to the TableService.insert_entity method. The entity object can be a dictionary or an object of type Entity, and defines your entity's property names and values. Every entity must include the required PartitionKey and RowKey properties, in addition to any other properties you define for the entity.

This example creates a dictionary object representing an entity, then passes it to the insert_entity method to add it to the table:

```
task = {'PartitionKey': 'tasksSeattle', 'RowKey': '001',
        'description': 'Take out the trash', 'priority': 200}
table_service.insert_entity('tasktable', task)
```

This example creates an Entity object, then passes it to the insert_entity method to add it to the table:

```
task = Entity()
task.PartitionKey = 'tasksSeattle'
task.RowKey = '002'
task.description = 'Wash the car'
task.priority = 100
table_service.insert_entity('tasktable', task)
```

PartitionKey and RowKey

You must specify both a **PartitionKey** and a **RowKey** property for every entity. These are the unique identifiers of your entities, as together they form the primary key of an entity. You can query using these values much faster than you can query any other entity properties because only these properties are indexed.

The Table service uses **PartitionKey** to intelligently distribute table entities across storage nodes. Entities that have the same **PartitionKey** are stored on the same node. **RowKey** is the unique ID of the entity within the partition it belongs to.

Update an entity

To update all of an entity's property values, call the update_entity method. This example shows how to replace an existing entity with an updated version:

```
task = {'PartitionKey': 'tasksSeattle', 'RowKey': '001',
        'description': 'Take out the garbage', 'priority': 250}
table_service.update_entity('tasktable', task)
```

If the entity that is being updated doesn't already exist, then the update operation will fail. If you want to store an entity whether it exists or not, use insert_or_replace_entity. In the following example, the first call will replace the existing entity. The second call will insert a new entity, since no entity with the specified PartitionKey and RowKey exists in the table.

```
# Replace the entity created earlier
task = {'PartitionKey': 'tasksSeattle', 'RowKey': '001',
        'description': 'Take out the garbage again', 'priority': 250}
table_service.insert_or_replace_entity('tasktable', task)

# Insert a new entity
task = {'PartitionKey': 'tasksSeattle', 'RowKey': '003',
        'description': 'Buy detergent', 'priority': 300}
table_service.insert_or_replace_entity('tasktable', task)
```

> **TIP**
> The update_entity method replaces all properties and values of an existing entity, which you can also use to remove properties from an existing entity. You can use the merge_entity method to update an existing entity with new or modified property values without completely replacing the entity.

Modify multiple entities

To ensure the atomic processing of a request by the Table service, you can submit multiple operations together in a batch. First, use the TableBatch class to add multiple operations to a single batch. Next, call TableService.commit_batch to submit the operations in an atomic operation. All entities to be modified in batch must be in the same partition.

This example adds two entities together in a batch:

```
from azure.cosmosdb.table.tablebatch import TableBatch
batch = TableBatch()
task004 = {'PartitionKey': 'tasksSeattle', 'RowKey': '004',
           'description': 'Go grocery shopping', 'priority': 400}
task005 = {'PartitionKey': 'tasksSeattle', 'RowKey': '005',
           'description': 'Clean the bathroom', 'priority': 100}
batch.insert_entity(task004)
batch.insert_entity(task005)
table_service.commit_batch('tasktable', batch)
```

Batches can also be used with the context manager syntax:

```
task006 = {'PartitionKey': 'tasksSeattle', 'RowKey': '006',
           'description': 'Go grocery shopping', 'priority': 400}
task007 = {'PartitionKey': 'tasksSeattle', 'RowKey': '007',
           'description': 'Clean the bathroom', 'priority': 100}

with table_service.batch('tasktable') as batch:
    batch.insert_entity(task006)
    batch.insert_entity(task007)
```

Query for an entity

To query for an entity in a table, pass its PartitionKey and RowKey to the TableService.get_entity method.

```
task = table_service.get_entity('tasktable', 'tasksSeattle', '001')
print(task.description)
print(task.priority)
```

Query a set of entities

You can query for a set of entities by supplying a filter string with the **filter** parameter. This example finds all tasks in Seattle by applying a filter on PartitionKey:

```
tasks = table_service.query_entities(
    'tasktable', filter="PartitionKey eq 'tasksSeattle'")
for task in tasks:
    print(task.description)
    print(task.priority)
```

Query a subset of entity properties

You can also restrict which properties are returned for each entity in a query. This technique, called *projection*, reduces bandwidth and can improve query performance, especially for large entities or result sets. Use the **select** parameter and pass the names of the properties you want returned to the client.

The query in the following code returns only the descriptions of entities in the table.

> **NOTE**
>
> The following snippet works only against the Azure Storage. It is not supported by the Storage Emulator.

```
tasks = table_service.query_entities(
    'tasktable', filter="PartitionKey eq 'tasksSeattle'", select='description')
for task in tasks:
    print(task.description)
```

Delete an entity

Delete an entity by passing its **PartitionKey** and **RowKey** to the delete_entity method.

```
table_service.delete_entity('tasktable', 'tasksSeattle', '001')
```

Delete a table

If you no longer need a table or any of the entities within it, call the delete_table method to permanently delete the table from Azure Storage.

```
table_service.delete_table('tasktable')
```

Next steps

- FAQ - Develop with the Table API
- Azure Cosmos DB SDK for Python API reference
- Python Developer Center
- Microsoft Azure Storage Explorer: A free, cross-platform application for working visually with Azure Storage data on Windows, macOS, and Linux.
- Working with Python in Visual Studio (Windows)

How to use Azure Storage Table service or the Azure Cosmos DB Table API from PHP

11/2/2020 • 12 minutes to read • Edit Online

APPLIES TO: Table API

> **TIP**
>
> The content in this article applies to Azure Table storage and the Azure Cosmos DB Table API. The Azure Cosmos DB Table API is a premium offering for table storage that offers throughput-optimized tables, global distribution, and automatic secondary indexes.

This article shows you how to create tables, store your data, and perform CRUD operations on the data. Choose either the Azure Table service or the Azure Cosmos DB Table API. The samples are written in PHP and use the Azure Storage Table PHP Client Library. The scenarios covered include **creating and deleting a table**, and **inserting, deleting, and querying entities in a table**. For more information on the Azure Table service, see the Next steps section.

Create an Azure service account

You can work with tables using the Azure Table storage or the Azure Cosmos DB. To learn more about the differences between table offerings in these two services, see the Table offerings article. You'll need to create an account for the service you're going to use. The following sections show how to create both Azure Table storage and the Azure Cosmos DB account, however you can just use one of them.

Create an Azure storage account

The easiest way to create an Azure storage account is by using the Azure portal. To learn more, see Create a storage account.

You can also create an Azure storage account by using Azure PowerShell or Azure CLI.

If you prefer not to create a storage account at this time, you can also use the Azure Storage Emulator to run and test your code in a local environment. For more information, see Use the Azure Storage Emulator for development and testing.

Create an Azure Cosmos DB Table API account

For instructions on creating an Azure Cosmos DB Table API account, see Create a database account.

Create a PHP application

The only requirement to create a PHP application to access the Storage Table service or Azure Cosmos DB Table API is to reference classes in the azure-storage-table SDK for PHP from within your code. You can use any development tools to create your application, including Notepad.

In this guide, you use Storage Table service or Azure Cosmos DB features that can be called from within a PHP application locally, or in code running within an Azure web role, worker role, or website.

Get the client library

1. Create a file named composer.json in the root of your project and add the following code to it:

```
{
"require": {
 "microsoft/azure-storage-table": "*"
}
}
```

2. Download composer.phar in your root.
3. Open a command prompt and execute the following command in your project root:

```
php composer.phar install
```

   Alternatively, go to the Azure Storage Table PHP Client Library on GitHub to clone the source code.

Add required references

To use the Storage Table service or Azure Cosmos DB APIs, you must:

- Reference the autoloader file using the require_once statement, and
- Reference any classes you use.

The following example shows how to include the autoloader file and reference the **TableRestProxy** class.

```
require_once 'vendor/autoload.php';
use MicrosoftAzure\Storage\Table\TableRestProxy;
```

In the examples below, the `require_once` statement is always shown, but only the classes necessary for the example to execute are referenced.

Add your connection string

You can either connect to the Azure storage account or the Azure Cosmos DB Table API account. Get the connection string based on the type of account you are using.

Add a Storage Table service connection

To instantiate a Storage Table service client, you must first have a valid connection string. The format for the Storage Table service connection string is:

```
$connectionString = "DefaultEndpointsProtocol=[http|https];AccountName=[yourAccount];AccountKey=[yourKey]"
```

Add a Storage Emulator connection

To access the emulator storage:

```
UseDevelopmentStorage = true
```

Add an Azure Cosmos DB connection

To instantiate an Azure Cosmos DB Table client, you must first have a valid connection string. The format for the Azure Cosmos DB connection string is:

```
$connectionString = "DefaultEndpointsProtocol=[https];AccountName=[myaccount];AccountKey=
[myaccountkey];TableEndpoint=[https://myendpoint/]";
```

To create an Azure Table service client or Azure Cosmos DB client, you need to use the **TableRestProxy** class. You can:

- Pass the connection string directly to it or
- Use the **CloudConfigurationManager (CCM)** to check multiple external sources for the connection string:
  - By default, it comes with support for one external source - environmental variables.
  - You can add new sources by extending the `ConnectionStringSource` class.

For the examples outlined here, the connection string is passed directly.

```
require_once 'vendor/autoload.php';

use MicrosoftAzure\Storage\Table\TableRestProxy;

$tableClient = TableRestProxy::createTableService($connectionString);
```

Create a table

A **TableRestProxy** object lets you create a table with the **createTable** method. When creating a table, you can set the Table service timeout. (For more information about the Table service timeout, see Setting Timeouts for Table Service Operations.)

```
require_once 'vendor\autoload.php';

use MicrosoftAzure\Storage\Table\TableRestProxy;
use MicrosoftAzure\Storage\Common\Exceptions\ServiceException;

// Create Table REST proxy.
$tableClient = TableRestProxy::createTableService($connectionString);

try    {
    // Create table.
    $tableClient->createTable("mytable");
}
catch(ServiceException $e){
    $code = $e->getCode();
    $error_message = $e->getMessage();
    // Handle exception based on error codes and messages.
    // Error codes and messages can be found here:
    // https://docs.microsoft.com/rest/api/storageservices/Table-Service-Error-Codes
}
```

For information about restrictions on table names, see Understanding the Table Service Data Model.

Add an entity to a table

To add an entity to a table, create a new **Entity** object and pass it to **TableRestProxy->insertEntity**. Note that when you create an entity, you must specify a `PartitionKey` and `RowKey`. These are the unique identifiers for an entity and are values that can be queried much faster than other entity properties. The system uses `PartitionKey` to automatically distribute the table's entities over many Storage nodes. Entities with the same `PartitionKey` are stored on the same node. (Operations on multiple entities stored on the same node perform better than on entities stored across different nodes.) The `RowKey` is the unique ID of an entity within a partition.

```
require_once 'vendor/autoload.php';

use MicrosoftAzure\Storage\Table\TableRestProxy;
use MicrosoftAzure\Storage\Common\Exceptions\ServiceException;
use MicrosoftAzure\Storage\Table\Models\Entity;
use MicrosoftAzure\Storage\Table\Models\EdmType;

// Create table REST proxy.
$tableClient = TableRestProxy::createTableService($connectionString);

$entity = new Entity();
$entity->setPartitionKey("tasksSeattle");
$entity->setRowKey("1");
$entity->addProperty("Description", null, "Take out the trash.");
$entity->addProperty("DueDate",
                        EdmType::DATETIME,
                        new DateTime("2012-11-05T08:15:00-08:00"));
$entity->addProperty("Location", EdmType::STRING, "Home");

try{
    $tableClient->insertEntity("mytable", $entity);
}
catch(ServiceException $e){
    // Handle exception based on error codes and messages.
    // Error codes and messages are here:
    // https://docs.microsoft.com/rest/api/storageservices/Table-Service-Error-Codes
    $code = $e->getCode();
    $error_message = $e->getMessage();
}
```

For information about Table properties and types, see Understanding the Table Service Data Model.

The **TableRestProxy** class offers two alternative methods for inserting entities: **insertOrMergeEntity** and **insertOrReplaceEntity**. To use these methods, create a new **Entity** and pass it as a parameter to either method. Each method will insert the entity if it does not exist. If the entity already exists, **insertOrMergeEntity** updates property values if the properties already exist and adds new properties if they do not exist, while **insertOrReplaceEntity** completely replaces an existing entity. The following example shows how to use **insertOrMergeEntity**. If the entity with `PartitionKey` "tasksSeattle" and `RowKey` "1" does not already exist, it will be inserted. However, if it has previously been inserted (as shown in the example above), the `DueDate` property is updated, and the `Status` property is added. The `Description` and `Location` properties are also updated, but with values that effectively leave them unchanged. If these latter two properties were not added as shown in the example, but existed on the target entity, their existing values would remain unchanged.

```
require_once 'vendor/autoload.php';

use MicrosoftAzure\Storage\Table\TableRestProxy;
use MicrosoftAzure\Storage\Common\Exceptions\ServiceException;
use MicrosoftAzure\Storage\Table\Models\Entity;
use MicrosoftAzure\Storage\Table\Models\EdmType;

// Create table REST proxy.
$tableClient = TableRestProxy::createTableService($connectionString);

//Create new entity.
$entity = new Entity();

// PartitionKey and RowKey are required.
$entity->setPartitionKey("tasksSeattle");
$entity->setRowKey("1");

// If entity exists, existing properties are updated with new values and
// new properties are added. Missing properties are unchanged.
$entity->addProperty("Description", null, "Take out the trash.");
$entity->addProperty("DueDate", EdmType::DATETIME, new DateTime()); // Modified the DueDate field.
$entity->addProperty("Location", EdmType::STRING, "Home");
$entity->addProperty("Status", EdmType::STRING, "Complete"); // Added Status field.

try    {
    // Calling insertOrReplaceEntity, instead of insertOrMergeEntity as shown,
    // would simply replace the entity with PartitionKey "tasksSeattle" and RowKey "1".
    $tableClient->insertOrMergeEntity("mytable", $entity);
}
catch(ServiceException $e){
    // Handle exception based on error codes and messages.
    // Error codes and messages are here:
    // https://docs.microsoft.com/rest/api/storageservices/Table-Service-Error-Codes
    $code = $e->getCode();
    $error_message = $e->getMessage();
    echo $code.": ".$error_message."<br />";
}
```

Retrieve a single entity

The **TableRestProxy->getEntity** method allows you to retrieve a single entity by querying for its `PartitionKey` and `RowKey`. In the example below, the partition key `tasksSeattle` and row key `1` are passed to the **getEntity** method.

```
require_once 'vendor/autoload.php';

use MicrosoftAzure\Storage\Table\TableRestProxy;
use MicrosoftAzure\Storage\Common\Exceptions\ServiceException;

// Create table REST proxy.
$tableClient = TableRestProxy::createTableService($connectionString);

try    {
    $result = $tableClient->getEntity("mytable", "tasksSeattle", 1);
}
catch(ServiceException $e){
    // Handle exception based on error codes and messages.
    // Error codes and messages are here:
    // https://docs.microsoft.com/rest/api/storageservices/Table-Service-Error-Codes
    $code = $e->getCode();
    $error_message = $e->getMessage();
    echo $code.": ".$error_message."<br />";
}

$entity = $result->getEntity();

echo $entity->getPartitionKey().":".$entity->getRowKey();
```

Retrieve all entities in a partition

Entity queries are constructed using filters (for more information, see Querying Tables and Entities). To retrieve all entities in partition, use the filter "PartitionKey eq *partition_name*". The following example shows how to retrieve all entities in the `tasksSeattle` partition by passing a filter to the **queryEntities** method.

```
require_once 'vendor/autoload.php';

use MicrosoftAzure\Storage\Table\TableRestProxy;
use MicrosoftAzure\Storage\Common\Exceptions\ServiceException;

// Create table REST proxy.
$tableClient = TableRestProxy::createTableService($connectionString);

$filter = "PartitionKey eq 'tasksSeattle'";

try    {
    $result = $tableClient->queryEntities("mytable", $filter);
}
catch(ServiceException $e){
    // Handle exception based on error codes and messages.
    // Error codes and messages are here:
    // https://docs.microsoft.com/rest/api/storageservices/Table-Service-Error-Codes
    $code = $e->getCode();
    $error_message = $e->getMessage();
    echo $code.": ".$error_message."<br />";
}

$entities = $result->getEntities();

foreach($entities as $entity){
    echo $entity->getPartitionKey().":".$entity->getRowKey()."<br />";
}
```

Retrieve a subset of entities in a partition

The same pattern used in the previous example can be used to retrieve any subset of entities in a partition. The

subset of entities you retrieve are determined by the filter you use (for more information, see Querying Tables and Entities).The following example shows how to use a filter to retrieve all entities with a specific `Location` and a `DueDate` less than a specified date.

```
require_once 'vendor/autoload.php';

use MicrosoftAzure\Storage\Table\TableRestProxy;
use MicrosoftAzure\Storage\Common\Exceptions\ServiceException;

// Create table REST proxy.
$tableClient = TableRestProxy::createTableService($connectionString);

$filter = "Location eq 'Office' and DueDate lt '2012-11-5'";

try    {
    $result = $tableClient->queryEntities("mytable", $filter);
}
catch(ServiceException $e){
    // Handle exception based on error codes and messages.
    // Error codes and messages are here:
    // https://docs.microsoft.com/rest/api/storageservices/Table-Service-Error-Codes
    $code = $e->getCode();
    $error_message = $e->getMessage();
    echo $code.": ".$error_message."<br />";
}

$entities = $result->getEntities();

foreach($entities as $entity){
    echo $entity->getPartitionKey().":".$entity->getRowKey()."<br />";
}
```

Retrieve a subset of entity properties

A query can retrieve a subset of entity properties. This technique, called *projection*, reduces bandwidth and can improve query performance, especially for large entities. To specify a property to retrieve, pass the name of the property to the **`Query->addSelectField`** method. You can call this method multiple times to add more properties. After executing **`TableRestProxy->queryEntities`**, the returned entities will only have the selected properties. (If you want to return a subset of Table entities, use a filter as shown in the queries above.)

```
require_once 'vendor/autoload.php';

use MicrosoftAzure\Storage\Table\TableRestProxy;
use MicrosoftAzure\Storage\Common\Exceptions\ServiceException;
use MicrosoftAzure\Storage\Table\Models\QueryEntitiesOptions;

// Create table REST proxy.
$tableClient = TableRestProxy::createTableService($connectionString);

$options = new QueryEntitiesOptions();
$options->addSelectField("Description");

try    {
    $result = $tableClient->queryEntities("mytable", $options);
}
catch(ServiceException $e){
    // Handle exception based on error codes and messages.
    // Error codes and messages are here:
    // https://docs.microsoft.com/rest/api/storageservices/Table-Service-Error-Codes
    $code = $e->getCode();
    $error_message = $e->getMessage();
    echo $code.": ".$error_message."<br />";
}

// All entities in the table are returned, regardless of whether
// they have the Description field.
// To limit the results returned, use a filter.
$entities = $result->getEntities();

foreach($entities as $entity){
    $description = $entity->getProperty("Description")->getValue();
    echo $description."<br />";
}
```

Update an entity

You can update an existing entity by using the **Entity->setProperty** and **Entity->addProperty** methods on the entity, and then calling **TableRestProxy->updateEntity**. The following example retrieves an entity, modifies one property, removes another property, and adds a new property. Note that you can remove a property by setting its value to **null**.

```
require_once 'vendor/autoload.php';

use MicrosoftAzure\Storage\Table\TableRestProxy;
use MicrosoftAzure\Storage\Common\Exceptions\ServiceException;
use MicrosoftAzure\Storage\Table\Models\Entity;
use MicrosoftAzure\Storage\Table\Models\EdmType;

// Create table REST proxy.
$tableClient = TableRestProxy::createTableService($connectionString);

$result = $tableClient->getEntity("mytable", "tasksSeattle", 1);

$entity = $result->getEntity();
$entity->setPropertyValue("DueDate", new DateTime()); //Modified DueDate.
$entity->setPropertyValue("Location", null); //Removed Location.
$entity->addProperty("Status", EdmType::STRING, "In progress"); //Added Status.

try    {
    $tableClient->updateEntity("mytable", $entity);
}
catch(ServiceException $e){
    // Handle exception based on error codes and messages.
    // Error codes and messages are here:
    // https://docs.microsoft.com/rest/api/storageservices/Table-Service-Error-Codes
    $code = $e->getCode();
    $error_message = $e->getMessage();
    echo $code.": ".$error_message."<br />";
}
```

Delete an entity

To delete an entity, pass the table name, and the entity's `PartitionKey` and `RowKey` to the **TableRestProxy->deleteEntity** method.

```
require_once 'vendor/autoload.php';

use MicrosoftAzure\Storage\Table\TableRestProxy;
use MicrosoftAzure\Storage\Common\Exceptions\ServiceException;

// Create table REST proxy.
$tableClient = TableRestProxy::createTableService($connectionString);

try    {
    // Delete entity.
    $tableClient->deleteEntity("mytable", "tasksSeattle", "2");
}
catch(ServiceException $e){
    // Handle exception based on error codes and messages.
    // Error codes and messages are here:
    // https://docs.microsoft.com/rest/api/storageservices/Table-Service-Error-Codes
    $code = $e->getCode();
    $error_message = $e->getMessage();
    echo $code.": ".$error_message."<br />";
}
```

For concurrency checks, you can set the Etag for an entity to be deleted by using the **DeleteEntityOptions->setEtag** method and passing the **DeleteEntityOptions** object to **deleteEntity** as a fourth parameter.

Batch table operations

The **TableRestProxy->batch** method allows you to execute multiple operations in a single request. The pattern here involves adding operations to **BatchRequest** object and then passing the **BatchRequest** object to the

TableRestProxy->batch method. To add an operation to a **BatchRequest** object, you can call any of the following methods multiple times:

- **addInsertEntity** (adds an insertEntity operation)
- **addUpdateEntity** (adds an updateEntity operation)
- **addMergeEntity** (adds a mergeEntity operation)
- **addInsertOrReplaceEntity** (adds an insertOrReplaceEntity operation)
- **addInsertOrMergeEntity** (adds an insertOrMergeEntity operation)
- **addDeleteEntity** (adds a deleteEntity operation)

The following example shows how to execute **insertEntity** and **deleteEntity** operations in a single request.

```
require_once 'vendor/autoload.php';

use MicrosoftAzure\Storage\Table\TableRestProxy;
use MicrosoftAzure\Storage\Common\Exceptions\ServiceException;
use MicrosoftAzure\Storage\Table\Models\Entity;
use MicrosoftAzure\Storage\Table\Models\EdmType;
use MicrosoftAzure\Storage\Table\Models\BatchOperations;

// Configure a connection string for Storage Table service.
$connectionString = "DefaultEndpointsProtocol=[http|https];AccountName=[yourAccount];AccountKey=[yourKey]"

// Create table REST proxy.
$tableClient = TableRestProxy::createTableService($connectionString);

// Create list of batch operation.
$operations = new BatchOperations();

$entity1 = new Entity();
$entity1->setPartitionKey("tasksSeattle");
$entity1->setRowKey("2");
$entity1->addProperty("Description", null, "Clean roof gutters.");
$entity1->addProperty("DueDate",
                        EdmType::DATETIME,
                        new DateTime("2012-11-05T08:15:00-08:00"));
$entity1->addProperty("Location", EdmType::STRING, "Home");

// Add operation to list of batch operations.
$operations->addInsertEntity("mytable", $entity1);

// Add operation to list of batch operations.
$operations->addDeleteEntity("mytable", "tasksSeattle", "1");

try    {
    $tableClient->batch($operations);
}
catch(ServiceException $e){
    // Handle exception based on error codes and messages.
    // Error codes and messages are here:
    // https://docs.microsoft.com/rest/api/storageservices/Table-Service-Error-Codes
    $code = $e->getCode();
    $error_message = $e->getMessage();
    echo $code.": ".$error_message."<br />";
}
```

For more information about batching Table operations, see Performing Entity Group Transactions.

Delete a table

Finally, to delete a table, pass the table name to the **TableRestProxy->deleteTable** method.

```
require_once 'vendor/autoload.php';

use MicrosoftAzure\Storage\Table\TableRestProxy;
use MicrosoftAzure\Storage\Common\Exceptions\ServiceException;

// Create table REST proxy.
$tableClient = TableRestProxy::createTableService($connectionString);

try    {
    // Delete table.
    $tableClient->deleteTable("mytable");
}
catch(ServiceException $e){
    // Handle exception based on error codes and messages.
    // Error codes and messages are here:
    // https://docs.microsoft.com/rest/api/storageservices/Table-Service-Error-Codes
    $code = $e->getCode();
    $error_message = $e->getMessage();
    echo $code.": ".$error_message."<br />";
}
```

Next steps

Now that you've learned the basics of the Azure Table service and Azure Cosmos DB, follow these links to learn more.

- Microsoft Azure Storage Explorer is a free, standalone app from Microsoft that enables you to work visually with Azure Storage data on Windows, macOS, and Linux.
- PHP Developer Center.

How to use Azure Table storage and Azure Cosmos DB Table API with C++

12/4/2020 • 14 minutes to read • Edit Online

APPLIES TO: Table API

> **TIP**
> The content in this article applies to Azure Table storage and the Azure Cosmos DB Table API. The Azure Cosmos DB Table API is a premium offering for table storage that offers throughput-optimized tables, global distribution, and automatic secondary indexes.

This guide shows you common scenarios by using the Azure Table storage service or Azure Cosmos DB Table API. The samples are written in C++ and use the Azure Storage Client Library for C++. This article covers the following scenarios:

- Create and delete a table
- Work with table entities

> **NOTE**
> This guide targets the Azure Storage Client Library for C++ version 1.0.0 and above. The recommended version is Storage Client Library 2.2.0, which is available by using NuGet or GitHub.

Create accounts

Create an Azure service account

You can work with tables using the Azure Table storage or the Azure Cosmos DB. To learn more about the differences between table offerings in these two services, see the Table offerings article. You'll need to create an account for the service you're going to use. The following sections show how to create both Azure Table storage and the Azure Cosmos DB account, however you can just use one of them.

Create an Azure storage account

The easiest way to create an Azure storage account is by using the Azure portal. To learn more, see Create a storage account.

You can also create an Azure storage account by using Azure PowerShell or Azure CLI.

If you prefer not to create a storage account at this time, you can also use the Azure Storage Emulator to run and test your code in a local environment. For more information, see Use the Azure Storage Emulator for development and testing.

Create an Azure Cosmos DB Table API account

For instructions on creating an Azure Cosmos DB Table API account, see Create a database account.

Create a C++ application

In this guide, you use storage features from a C++ application. To do so, install the Azure Storage Client Library for C++.

To install the Azure Storage Client Library for C++, use the following methods:

- **Linux:** Follow the instructions given in the Azure Storage Client Library for C++ README: Getting Started on Linux page.
- **Windows:** On Windows, use vcpkg as the dependency manager. Follow the quick-start to initialize vcpkg. Then, use the following command to install the library:

```
.\vcpkg.exe install azure-storage-cpp
```

You can find a guide for how to build the source code and export to Nuget in the README file.

Configure access to the Table client library

To use the Azure storage APIs to access tables, add the following `include` statements to the top of the C++ file:

```
#include <was/storage_account.h>
#include <was/table.h>
```

An Azure Storage client or Cosmos DB client uses a connection string to store endpoints and credentials to access data management services. When you run a client application, you must provide the storage connection string or Azure Cosmos DB connection string in the appropriate format.

Set up an Azure Storage connection string

This example shows how to declare a static field to hold the Azure Storage connection string:

```
// Define the Storage connection string with your values.
const utility::string_t storage_connection_string(U("DefaultEndpointsProtocol=https;AccountName=
<your_storage_account>;AccountKey=<your_storage_account_key>"));
```

Use the name of your Storage account for `<your_storage_account>`. For <your_storage_account_key>, use the access key for the Storage account listed in the Azure portal. For information on Storage accounts and access keys, see Create a storage account.

Set up an Azure Cosmos DB connection string

This example shows how to declare a static field to hold the Azure Cosmos DB connection string:

```
// Define the Azure Cosmos DB connection string with your values.
const utility::string_t storage_connection_string(U("DefaultEndpointsProtocol=https;AccountName=
<your_cosmos_db_account>;AccountKey=<your_cosmos_db_account_key>;TableEndpoint=<your_cosmos_db_endpoint>"));
```

Use the name of your Azure Cosmos DB account for `<your_cosmos_db_account>`. Enter your primary key for `<your_cosmos_db_account_key>`. Enter the endpoint listed in the Azure portal for `<your_cosmos_db_endpoint>`.

To test your application in your local Windows-based computer, you can use the Azure Storage Emulator that is installed with the Azure SDK. The Storage Emulator is a utility that simulates the Azure Blob, Queue, and Table services available on your local development machine. The following example shows how to declare a static field to hold the connection string to your local storage emulator:

```
// Define the connection string with Azure Storage Emulator.
const utility::string_t storage_connection_string(U("UseDevelopmentStorage=true;"));
```

To start the Azure Storage Emulator, from your Windows desktop, select the **Start** button or the Windows key. Enter and run *Microsoft Azure Storage Emulator*. For more information, see Use the Azure Storage Emulator for development and testing.

Retrieve your connection string

You can use the `cloud_storage_account` class to represent your storage account information. To retrieve your storage account information from the storage connection string, use the `parse` method.

```
// Retrieve the storage account from the connection string.
azure::storage::cloud_storage_account storage_account =
azure::storage::cloud_storage_account::parse(storage_connection_string);
```

Next, get a reference to a `cloud_table_client` class. This class lets you get reference objects for tables and entities stored within the Table storage service. The following code creates a `cloud_table_client` object by using the storage account object you retrieved previously:

```
// Create the table client.
azure::storage::cloud_table_client table_client = storage_account.create_cloud_table_client();
```

Create and add entities to a table

Create a table

A `cloud_table_client` object lets you get reference objects for tables and entities. The following code creates a `cloud_table_client` object and uses it to create a new table.

```
// Retrieve the storage account from the connection string.
azure::storage::cloud_storage_account storage_account =
azure::storage::cloud_storage_account::parse(storage_connection_string);

// Create the table client.
azure::storage::cloud_table_client table_client = storage_account.create_cloud_table_client();

// Retrieve a reference to a table.
azure::storage::cloud_table table = table_client.get_table_reference(U("people"));

// Create the table if it doesn't exist.
table.create_if_not_exists();
```

Add an entity to a table

To add an entity to a table, create a new `table_entity` object and pass it to `table_operation::insert_entity`. The following code uses the customer's first name as the row key and last name as the partition key. Together, an entity's partition and row key uniquely identify the entity in the table. Entities with the same partition key can be queried faster than entities with different partition keys. Using diverse partition keys allows for greater parallel operation scalability. For more information, see Microsoft Azure storage performance and scalability checklist.

The following code creates a new instance of `table_entity` with some customer data to store. The code next calls `table_operation::insert_entity` to create a `table_operation` object to insert an entity into a table, and associates the new table entity with it. Finally, the code calls the `execute` method on the `cloud_table` object. The new `table_operation` sends a request to the Table service to insert the new customer entity into the `people` table.

```
// Retrieve the storage account from the connection string.
azure::storage::cloud_storage_account storage_account =
azure::storage::cloud_storage_account::parse(storage_connection_string);

// Create the table client.
azure::storage::cloud_table_client table_client = storage_account.create_cloud_table_client();

// Retrieve a reference to a table.
azure::storage::cloud_table table = table_client.get_table_reference(U("people"));

// Create the table if it doesn't exist.
table.create_if_not_exists();

// Create a new customer entity.
azure::storage::table_entity customer1(U("Harp"), U("Walter"));

azure::storage::table_entity::properties_type& properties = customer1.properties();
properties.reserve(2);
properties[U("Email")] = azure::storage::entity_property(U("Walter@contoso.com"));

properties[U("Phone")] = azure::storage::entity_property(U("425-555-0101"));

// Create the table operation that inserts the customer entity.
azure::storage::table_operation insert_operation = azure::storage::table_operation::insert_entity(customer1);

// Execute the insert operation.
azure::storage::table_result insert_result = table.execute(insert_operation);
```

Insert a batch of entities

You can insert a batch of entities to the Table service in one write operation. The following code creates a `table_batch_operation` object, and then adds three insert operations to it. Each insert operation is added by creating a new entity object, setting its values, and then calling the `insert` method on the `table_batch_operation` object to associate the entity with a new insert operation. Then, the code calls `cloud_table.execute` to run the operation.

```
// Retrieve the storage account from the connection string.
azure::storage::cloud_storage_account storage_account =
azure::storage::cloud_storage_account::parse(storage_connection_string);

// Create the table client.
azure::storage::cloud_table_client table_client = storage_account.create_cloud_table_client();

// Create a cloud table object for the table.
azure::storage::cloud_table table = table_client.get_table_reference(U("people"));

// Define a batch operation.
azure::storage::table_batch_operation batch_operation;

// Create a customer entity and add it to the table.
azure::storage::table_entity customer1(U("Smith"), U("Jeff"));

azure::storage::table_entity::properties_type& properties1 = customer1.properties();
properties1.reserve(2);
properties1[U("Email")] = azure::storage::entity_property(U("Jeff@contoso.com"));
properties1[U("Phone")] = azure::storage::entity_property(U("425-555-0104"));

// Create another customer entity and add it to the table.
azure::storage::table_entity customer2(U("Smith"), U("Ben"));

azure::storage::table_entity::properties_type& properties2 = customer2.properties();
properties2.reserve(2);
properties2[U("Email")] = azure::storage::entity_property(U("Ben@contoso.com"));
properties2[U("Phone")] = azure::storage::entity_property(U("425-555-0102"));

// Create a third customer entity to add to the table.
azure::storage::table_entity customer3(U("Smith"), U("Denise"));

azure::storage::table_entity::properties_type& properties3 = customer3.properties();
properties3.reserve(2);
properties3[U("Email")] = azure::storage::entity_property(U("Denise@contoso.com"));
properties3[U("Phone")] = azure::storage::entity_property(U("425-555-0103"));

// Add customer entities to the batch insert operation.
batch_operation.insert_or_replace_entity(customer1);
batch_operation.insert_or_replace_entity(customer2);
batch_operation.insert_or_replace_entity(customer3);

// Execute the batch operation.
std::vector<azure::storage::table_result> results = table.execute_batch(batch_operation);
```

Some things to note on batch operations:

- You can do up to 100 `insert`, `delete`, `merge`, `replace`, `insert-or-merge`, and `insert-or-replace` operations in any combination in a single batch.
- A batch operation can have a retrieve operation, if it's the only operation in the batch.
- All entities in a single batch operation must have the same partition key.
- A batch operation is limited to a 4-MB data payload.

Query and modify entities

Retrieve all entities in a partition

To query a table for all entities in a partition, use a `table_query` object. The following code example specifies a filter for entities where `Smith` is the partition key. This example prints the fields of each entity in the query results to the console.

> **NOTE**
>
> These methods are not currently supported for C++ in Azure Cosmos DB.

```
// Retrieve the storage account from the connection string.
azure::storage::cloud_storage_account storage_account =
azure::storage::cloud_storage_account::parse(storage_connection_string);

// Create the table client.
azure::storage::cloud_table_client table_client = storage_account.create_cloud_table_client();

// Create a cloud table object for the table.
azure::storage::cloud_table table = table_client.get_table_reference(U("people"));

// Construct the query operation for all customer entities where PartitionKey="Smith".
azure::storage::table_query query;

query.set_filter_string(azure::storage::table_query::generate_filter_condition(U("PartitionKey"),
azure::storage::query_comparison_operator::equal, U("Smith")));

// Execute the query.
azure::storage::table_query_iterator it = table.execute_query(query);

// Print the fields for each customer.
azure::storage::table_query_iterator end_of_results;
for (; it != end_of_results; ++it)
{
    const azure::storage::table_entity::properties_type& properties = it->properties();

    std::wcout << U("PartitionKey: ") << it->partition_key() << U(", RowKey: ") << it->row_key()
        << U(", Property1: ") << properties.at(U("Email")).string_value()
        << U(", Property2: ") << properties.at(U("Phone")).string_value() << std::endl;
}
```

The query in this example returns all the entities that match the filter criteria. If you have large tables and need to download the table entities often, we recommend that you store your data in Azure storage blobs instead.

Retrieve a range of entities in a partition

If you don't want to query all the entities in a partition, you can specify a range. Combine the partition key filter with a row key filter. The following code example uses two filters to get all entities in partition `Smith` where the row key (first name) starts with a letter earlier than `E` in the alphabet, and then prints the query results.

> **NOTE**
>
> These methods are not currently supported for C++ in Azure Cosmos DB.

```
// Retrieve the storage account from the connection string.
azure::storage::cloud_storage_account storage_account =
azure::storage::cloud_storage_account::parse(storage_connection_string);

// Create the table client.
azure::storage::cloud_table_client table_client = storage_account.create_cloud_table_client();

// Create a cloud table object for the table.
azure::storage::cloud_table table = table_client.get_table_reference(U("people"));

// Create the table query.
azure::storage::table_query query;

query.set_filter_string(azure::storage::table_query::combine_filter_conditions(
    azure::storage::table_query::generate_filter_condition(U("PartitionKey"),
    azure::storage::query_comparison_operator::equal, U("Smith")),
    azure::storage::query_logical_operator::op_and,
    azure::storage::table_query::generate_filter_condition(U("RowKey"),
azure::storage::query_comparison_operator::less_than, U("E"))));

// Execute the query.
azure::storage::table_query_iterator it = table.execute_query(query);

// Loop through the results, displaying information about the entity.
azure::storage::table_query_iterator end_of_results;
for (; it != end_of_results; ++it)
{
    const azure::storage::table_entity::properties_type& properties = it->properties();

    std::wcout << U("PartitionKey: ") << it->partition_key() << U(", RowKey: ") << it->row_key()
        << U(", Property1: ") << properties.at(U("Email")).string_value()
        << U(", Property2: ") << properties.at(U("Phone")).string_value() << std::endl;
}
```

Retrieve a single entity

You can write a query to retrieve a single, specific entity. The following code uses `table_operation::retrieve_entity` to specify the customer `Jeff Smith`. This method returns just one entity, rather than a collection, and the returned value is in `table_result`. Specifying both partition and row keys in a query is the fastest way to retrieve a single entity from the Table service.

```
azure::storage::cloud_storage_account storage_account =
azure::storage::cloud_storage_account::parse(storage_connection_string);

// Create the table client.
azure::storage::cloud_table_client table_client = storage_account.create_cloud_table_client();

// Create a cloud table object for the table.
azure::storage::cloud_table table = table_client.get_table_reference(U("people"));

// Retrieve the entity with partition key of "Smith" and row key of "Jeff".
azure::storage::table_operation retrieve_operation =
azure::storage::table_operation::retrieve_entity(U("Smith"), U("Jeff"));
azure::storage::table_result retrieve_result = table.execute(retrieve_operation);

// Output the entity.
azure::storage::table_entity entity = retrieve_result.entity();
const azure::storage::table_entity::properties_type& properties = entity.properties();

std::wcout << U("PartitionKey: ") << entity.partition_key() << U(", RowKey: ") << entity.row_key()
    << U(", Property1: ") << properties.at(U("Email")).string_value()
    << U(", Property2: ") << properties.at(U("Phone")).string_value() << std::endl;
```

Replace an entity

To replace an entity, retrieve it from the Table service, modify the entity object, and then save the changes back to the Table service. The following code changes an existing customer's phone number and email address. Instead of calling `table_operation::insert_entity`, this code uses `table_operation::replace_entity`. This approach causes the entity to be fully replaced on the server, unless the entity on the server has changed since it was retrieved. If it has been changed, the operation fails. This failure prevents your application from overwriting a change made between the retrieval and update by another component. The proper handling of this failure is to retrieve the entity again, make your changes, if still valid, and then do another `table_operation::replace_entity` operation.

```
// Retrieve the storage account from the connection string.
azure::storage::cloud_storage_account storage_account =
azure::storage::cloud_storage_account::parse(storage_connection_string);

// Create the table client.
azure::storage::cloud_table_client table_client = storage_account.create_cloud_table_client();

// Create a cloud table object for the table.
azure::storage::cloud_table table = table_client.get_table_reference(U("people"));

// Replace an entity.
azure::storage::table_entity entity_to_replace(U("Smith"), U("Jeff"));
azure::storage::table_entity::properties_type& properties_to_replace = entity_to_replace.properties();
properties_to_replace.reserve(2);

// Specify a new phone number.
properties_to_replace[U("Phone")] = azure::storage::entity_property(U("425-555-0106"));

// Specify a new email address.
properties_to_replace[U("Email")] = azure::storage::entity_property(U("JeffS@contoso.com"));

// Create an operation to replace the entity.
azure::storage::table_operation replace_operation =
azure::storage::table_operation::replace_entity(entity_to_replace);

// Submit the operation to the Table service.
azure::storage::table_result replace_result = table.execute(replace_operation);
```

Insert or replace an entity

`table_operation::replace_entity` operations fail if the entity has been changed since it was retrieved from the server. Furthermore, you must retrieve the entity from the server first in order for `table_operation::replace_entity` to be successful. Sometimes, you don't know if the entity exists on the server. The current values stored in it are irrelevant, because your update should overwrite them all. To accomplish this result, use a `table_operation::insert_or_replace_entity` operation. This operation inserts the entity if it doesn't exist. The operation replaces the entity if it exists. In the following code example, the customer entity for `Jeff Smith` is still retrieved, but it's then saved back to the server by using `table_operation::insert_or_replace_entity`. Any updates made to the entity between the retrieval and update operation will be overwritten.

```
// Retrieve the storage account from the connection string.
azure::storage::cloud_storage_account storage_account =
azure::storage::cloud_storage_account::parse(storage_connection_string);

// Create the table client.
azure::storage::cloud_table_client table_client = storage_account.create_cloud_table_client();

// Create a cloud table object for the table.
azure::storage::cloud_table table = table_client.get_table_reference(U("people"));

// Insert or replace an entity.
azure::storage::table_entity entity_to_insert_or_replace(U("Smith"), U("Jeff"));
azure::storage::table_entity::properties_type& properties_to_insert_or_replace =
entity_to_insert_or_replace.properties();

properties_to_insert_or_replace.reserve(2);

// Specify a phone number.
properties_to_insert_or_replace[U("Phone")] = azure::storage::entity_property(U("425-555-0107"));

// Specify an email address.
properties_to_insert_or_replace[U("Email")] = azure::storage::entity_property(U("Jeffsm@contoso.com"));

// Create an operation to insert or replace the entity.
azure::storage::table_operation insert_or_replace_operation =
azure::storage::table_operation::insert_or_replace_entity(entity_to_insert_or_replace);

// Submit the operation to the Table service.
azure::storage::table_result insert_or_replace_result = table.execute(insert_or_replace_operation);
```

Query a subset of entity properties

A query to a table can retrieve just a few properties from an entity. The query in the following code uses the `table_query::set_select_columns` method to return only the email addresses of entities in the table.

```
// Retrieve the storage account from the connection string.
azure::storage::cloud_storage_account storage_account =
azure::storage::cloud_storage_account::parse(storage_connection_string);

// Create the table client.
azure::storage::cloud_table_client table_client = storage_account.create_cloud_table_client();

// Create a cloud table object for the table.
azure::storage::cloud_table table = table_client.get_table_reference(U("people"));

// Define the query, and select only the Email property.
azure::storage::table_query query;
std::vector<utility::string_t> columns;

columns.push_back(U("Email"));
query.set_select_columns(columns);

// Execute the query.
azure::storage::table_query_iterator it = table.execute_query(query);

// Display the results.
azure::storage::table_query_iterator end_of_results;
for (; it != end_of_results; ++it)
{
    std::wcout << U("PartitionKey: ") << it->partition_key() << U(", RowKey: ") << it->row_key();

    const azure::storage::table_entity::properties_type& properties = it->properties();
    for (auto prop_it = properties.begin(); prop_it != properties.end(); ++prop_it)
    {
        std::wcout << ", " << prop_it->first << ": " << prop_it->second.str();
    }

    std::wcout << std::endl;
}
```

NOTE

Querying a few properties from an entity is a more efficient operation than retrieving all properties.

Delete content

Delete an entity

You can delete an entity after you retrieve it. After you retrieve an entity, call `table_operation::delete_entity` with the entity to delete. Then call the `cloud_table.execute` method. The following code retrieves and deletes an entity with a partition key of `Smith` and a row key of `Jeff`.

```
// Retrieve the storage account from the connection string.
azure::storage::cloud_storage_account storage_account =
azure::storage::cloud_storage_account::parse(storage_connection_string);

// Create the table client.
azure::storage::cloud_table_client table_client = storage_account.create_cloud_table_client();

// Create a cloud table object for the table.
azure::storage::cloud_table table = table_client.get_table_reference(U("people"));

// Create an operation to retrieve the entity with partition key of "Smith" and row key of "Jeff".
azure::storage::table_operation retrieve_operation =
azure::storage::table_operation::retrieve_entity(U("Smith"), U("Jeff"));
azure::storage::table_result retrieve_result = table.execute(retrieve_operation);

// Create an operation to delete the entity.
azure::storage::table_operation delete_operation =
azure::storage::table_operation::delete_entity(retrieve_result.entity());

// Submit the delete operation to the Table service.
azure::storage::table_result delete_result = table.execute(delete_operation);
```

Delete a table

Finally, the following code example deletes a table from a storage account. A table that has been deleted is unavailable to be re-created for some time following the deletion.

```
// Retrieve the storage account from the connection string.
azure::storage::cloud_storage_account storage_account =
azure::storage::cloud_storage_account::parse(storage_connection_string);

// Create the table client.
azure::storage::cloud_table_client table_client = storage_account.create_cloud_table_client();

// Create a cloud table object for the table.
azure::storage::cloud_table table = table_client.get_table_reference(U("people"));

// Delete the table if it exists
if (table.delete_table_if_exists())
{
    std::cout << "Table deleted!";
}
else
{
    std::cout << "Table didn't exist";
}
```

Troubleshooting

For Visual Studio Community Edition, if your project gets build errors because of the include files *storage_account.h* and *table.h*, remove the **/permissive-** compiler switch:

1. In **Solution Explorer**, right-click your project and select **Properties**.
2. In the **Property Pages** dialog box, expand **Configuration Properties**, expand **C/C++**, and select **Language**.
3. Set **Conformance mode** to **No**.

Next steps

Microsoft Azure Storage Explorer is a free, standalone app from Microsoft that enables you to work visually with Azure Storage data on Windows, macOS, and Linux.

Follow these links to learn more about Azure Storage and the Table API in Azure Cosmos DB:

- Introduction to the Table API
- List Azure Storage resources in C++
- Storage Client Library for C++ reference
- Azure Storage documentation

How to use Azure Table Storage and the Azure Cosmos DB Table API with Ruby

11/2/2020 • 6 minutes to read • Edit Online

APPLIES TO: Table API

> **TIP**
> The content in this article applies to Azure Table storage and the Azure Cosmos DB Table API. The Azure Cosmos DB Table API is a premium offering for table storage that offers throughput-optimized tables, global distribution, and automatic secondary indexes.

This article shows you how to create tables, store your data, and perform CRUD operations on the data. Choose either the Azure Table service or the Azure Cosmos DB Table API. The samples described in this article are written in Ruby and uses the Azure Storage Table Client Library for Ruby. The scenarios covered include create a table, delete a table, insert entities, and query entities from the table.

Create an Azure service account

You can work with tables using the Azure Table storage or the Azure Cosmos DB. To learn more about the differences between table offerings in these two services, see the Table offerings article. You'll need to create an account for the service you're going to use. The following sections show how to create both Azure Table storage and the Azure Cosmos DB account, however you can just use one of them.

Create an Azure storage account

The easiest way to create an Azure storage account is by using the Azure portal. To learn more, see Create a storage account.

You can also create an Azure storage account by using Azure PowerShell or Azure CLI.

If you prefer not to create a storage account at this time, you can also use the Azure Storage Emulator to run and test your code in a local environment. For more information, see Use the Azure Storage Emulator for development and testing.

Create an Azure Cosmos DB account

For instructions on creating an Azure Cosmos DB Table API account, see Create a database account.

Add access to Azure storage or Azure Cosmos DB

To use Azure Storage or Azure Cosmos DB, you must download and use the Ruby Azure package that includes a set of convenience libraries that communicate with the Table REST services.

Use RubyGems to obtain the package

1. Use a command-line interface such as **PowerShell** (Windows), **Terminal** (Mac), or **Bash** (Unix).
2. Type **gem install azure-storage-table** in the command window to install the gem and dependencies.

Import the package

Use your favorite text editor, add the following to the top of the Ruby file where you intend to use Storage:

```
require "azure/storage/table"
```

Add your connection string

You can either connect to the Azure storage account or the Azure Cosmos DB Table API account. Get the connection string based on the type of account you are using.

Add an Azure Storage connection

The Azure Storage module reads the environment variables **AZURE_STORAGE_ACCOUNT** and **AZURE_STORAGE_ACCESS_KEY** for information required to connect to your Azure Storage account. If these environment variables are not set, you must specify the account information before using **Azure::Storage::Table::TableService** with the following code:

```
Azure.config.storage_account_name = "<your Azure Storage account>"
Azure.config.storage_access_key = "<your Azure Storage access key>"
```

To obtain these values from a classic or Resource Manager storage account in the Azure portal:

1. Log in to the Azure portal.
2. Navigate to the Storage account you want to use.
3. In the Settings blade on the right, click **Access Keys**.
4. In the Access keys blade that appears, you'll see the access key 1 and access key 2. You can use either of these.
5. Click the copy icon to copy the key to the clipboard.

Add an Azure Cosmos DB connection

To connect to Azure Cosmos DB, copy your primary connection string from the Azure portal, and create a **Client** object using your copied connection string. You can pass the **Client** object when you create a **TableService** object:

```
common_client = Azure::Storage::Common::Client.create(storage_account_name:'myaccount',
storage_access_key:'mykey', storage_table_host:'mycosmosdb_endpoint')
table_client = Azure::Storage::Table::TableService.new(client: common_client)
```

Create a table

The **Azure::Storage::Table::TableService** object lets you work with tables and entities. To create a table, use the **create_table()** method. The following example creates a table or prints the error if there is any.

```
azure_table_service = Azure::Storage::Table::TableService.new
begin
    azure_table_service.create_table("testtable")
rescue
    puts $!
end
```

Add an entity to a table

To add an entity, first create a hash object that defines your entity properties. Note that for every entity you must specify a **PartitionKey** and **RowKey**. These are the unique identifiers of your entities, and are values that can be queried much faster than your other properties. Azure Storage uses **PartitionKey** to automatically distribute the table's entities over many storage nodes. Entities with the same **PartitionKey** are stored on the same node. The

RowKey is the unique ID of the entity within the partition it belongs to.

```
entity = { "content" => "test entity",
    :PartitionKey => "test-partition-key", :RowKey => "1" }
azure_table_service.insert_entity("testtable", entity)
```

Update an entity

There are multiple methods available to update an existing entity:

- **update_entity():** Update an existing entity by replacing it.
- **merge_entity():** Updates an existing entity by merging new property values into the existing entity.
- **insert_or_merge_entity():** Updates an existing entity by replacing it. If no entity exists, a new one will be inserted:
- **insert_or_replace_entity():** Updates an existing entity by merging new property values into the existing entity. If no entity exists, a new one will be inserted.

The following example demonstrates updating an entity using **update_entity()**:

```
entity = { "content" => "test entity with updated content",
    :PartitionKey => "test-partition-key", :RowKey => "1" }
azure_table_service.update_entity("testtable", entity)
```

With **update_entity()** and **merge_entity()**, if the entity that you are updating doesn't exist then the update operation will fail. Therefore, if you want to store an entity regardless of whether it already exists, you should instead use **insert_or_replace_entity()** or **insert_or_merge_entity()**.

Work with groups of entities

Sometimes it makes sense to submit multiple operations together in a batch to ensure atomic processing by the server. To accomplish that, you first create a **Batch** object and then use the **execute_batch()** method on **TableService**. The following example demonstrates submitting two entities with RowKey 2 and 3 in a batch. Notice that it only works for entities with the same PartitionKey.

```
azure_table_service = Azure::TableService.new
batch = Azure::Storage::Table::Batch.new("testtable",
    "test-partition-key") do
    insert "2", { "content" => "new content 2" }
    insert "3", { "content" => "new content 3" }
end
results = azure_table_service.execute_batch(batch)
```

Query for an entity

To query an entity in a table, use the **get_entity()** method, by passing the table name, **PartitionKey** and **RowKey**.

```
result = azure_table_service.get_entity("testtable", "test-partition-key",
    "1")
```

Query a set of entities

To query a set of entities in a table, create a query hash object and use the **query_entities()** method. The

following example demonstrates getting all the entities with the same **PartitionKey**:

```
query = { :filter => "PartitionKey eq 'test-partition-key'" }
result, token = azure_table_service.query_entities("testtable", query)
```

> **NOTE**
> If the result set is too large for a single query to return, a continuation token is returned that you can use to retrieve subsequent pages.

Query a subset of entity properties

A query to a table can retrieve just a few properties from an entity. This technique, called "projection," reduces bandwidth and can improve query performance, especially for large entities. Use the select clause and pass the names of the properties you would like to bring over to the client.

```
query = { :filter => "PartitionKey eq 'test-partition-key'",
    :select => ["content"] }
result, token = azure_table_service.query_entities("testtable", query)
```

Delete an entity

To delete an entity, use the **delete_entity()** method. Pass in the name of the table that contains the entity, the PartitionKey, and the RowKey of the entity.

```
azure_table_service.delete_entity("testtable", "test-partition-key", "1")
```

Delete a table

To delete a table, use the **delete_table()** method and pass in the name of the table you want to delete.

```
azure_table_service.delete_table("testtable")
```

Next steps

- Microsoft Azure Storage Explorer is a free, standalone app from Microsoft that enables you to work visually with Azure Storage data on Windows, macOS, and Linux.
- Ruby Developer Center
- Microsoft Azure Storage Table Client Library for Ruby

Frequently asked questions about the Table API in Azure Cosmos DB

12/4/2020 • 15 minutes to read • Edit Online

APPLIES TO: Table API

The Azure Cosmos DB Table API is available in the Azure portal First you must sign up for an Azure subscription. After you've signed up, you can add an Azure Cosmos DB Table API account to your Azure subscription, and then add tables to your account. You can find the supported languages and associated quick-starts in the Introduction to Azure Cosmos DB Table API.

Table API in Azure Cosmos DB Vs Azure Table storage

Where is Table API not identical with Azure Table storage behavior?

There are some behavior differences that users coming from Azure Table storage who want to create tables with the Azure Cosmos DB Table API should be aware of:

- Azure Cosmos DB Table API uses a reserved capacity model in order to ensure guaranteed performance but this means that one pays for the capacity as soon as the table is created, even if the capacity isn't being used. With Azure Table storage one only pays for capacity that's used. This helps to explain why Table API can offer a 10 ms read and 15 ms write SLA at the 99th percentile while Azure Table storage offers a 10-second SLA. But as a consequence, with Table API tables, even empty tables without any requests, cost money in order to ensure the capacity is available to handle any requests to them at the SLA offered by Azure Cosmos DB.
- Query results returned by the Table API aren't sorted in partition key/row key order as they are in Azure Table storage.
- Row keys can only be up to 255 bytes.
- Batches can only have up to 2 MBs.
- CORS isn't currently supported.
- Table names in Azure Table storage aren't case-sensitive, but they are in Azure Cosmos DB Table API.
- Some of Azure Cosmos DB's internal formats for encoding information, such as binary fields, are currently not as efficient as one might like. Therefore this can cause unexpected limitations on data size. For example, currently one couldn't use the full one Meg of a table entity to store binary data because the encoding increases the data's size.
- Entity property name 'ID' currently not supported.
- TableQuery TakeCount isn't limited to 1000.
- In terms of the REST API there are a number of endpoints/query options that aren't supported by Azure Cosmos DB Table API:

REST METHOD(S)	REST ENDPOINT/QUERY OPTION	DOC URLS	EXPLANATION

REST METHOD(S)	REST ENDPOINT/QUERY OPTION	DOC URLS	EXPLANATION
GET, PUT	`/?restype=service@comp=properties`	Set Table Service Properties and Get Table Service Properties	This endpoint is used to set CORS rules, storage analytics configuration, and logging settings. CORS is currently not supported and analytics and logging are handled differently in Azure Cosmos DB than Azure Storage Tables
OPTIONS	`/<table-resource-name>`	Pre-flight CORS table request	This is part of CORS which Azure Cosmos DB doesn't currently support.
GET	`/?restype=service@comp=stats`	Get Table Service Stats	Provides information how quickly data is replicating between primary and secondaries. This isn't needed in Cosmos DB as the replication is part of writes.
GET, PUT	`/mytable?comp=acl`	Get Table ACL and Set Table ACL	This gets and sets the stored access policies used to manage Shared Access Signatures (SAS). Although SAS is supported, they are set and managed differently.

- Azure Cosmos DB Table API only supports the JSON format, not ATOM.
- While Azure Cosmos DB supports Shared Access Signatures (SAS) there are certain policies it doesn't support, specifically those related to management operations such as the right to create new tables.
- For the .NET SDK in particular, there are some classes and methods that Azure Cosmos DB doesn't currently support.

CLASS	UNSUPPORTED METHOD
CloudTableClient	*ServiceProperties*
	ServiceStats
CloudTable	SetPermissions*
	GetPermissions*
TableServiceContext	* (this class is deprecated)
TableServiceEntity	" "
TableServiceExtensions	" "

CLASS	UNSUPPORTED METHOD
TableServiceQuery	" "

Other frequently asked questions

Do I need a new SDK to use the Table API?

No, existing storage SDKs should still work. However, it's recommended that one always gets the latest SDKs for the best support and in many cases superior performance. See the list of available languages in the Introduction to Azure Cosmos DB Table API.

What is the connection string that I need to use to connect to the Table API?

The connection string is:

```
DefaultEndpointsProtocol=https;AccountName=<AccountNamefromCosmosDB;AccountKey=
<FromKeysPaneofCosmosDB>;TableEndpoint=https://<AccountName>.table.cosmosdb.azure.com
```

You can get the connection string from the Connection String page in the Azure portal.

How do I override the config settings for the request options in the .NET SDK for the Table API?

Some settings are handled on the CreateCloudTableClient method and other via the app.config in the appSettings section in the client application. For information about config settings, see Azure Cosmos DB capabilities.

Are there any changes for customers who are using the existing Azure Table storage SDKs?

None. There are no changes for existing or new customers who are using the existing Azure Table storage SDKs.

How do I view table data that's stored in Azure Cosmos DB for use with the Table API?

You can use the Azure portal to browse the data. You can also use the Table API code or the tools mentioned in the next answer.

Which tools work with the Table API?

You can use the Azure Storage Explorer.

Tools with the flexibility to take a connection string in the format specified previously can support the new Table API. A list of table tools is provided on the Azure Storage Client Tools page.

Is the concurrency on operations controlled?

Yes, optimistic concurrency is provided via the use of the ETag mechanism.

Is the OData query model supported for entities?

Yes, the Table API supports OData query and LINQ query.

Can I connect to Azure Table Storage and Azure Cosmos DB Table API side by side in the same application?

Yes, you can connect by creating two separate instances of the CloudTableClient, each pointing to its own URI via the connection string.

How do I migrate an existing Azure Table storage application to this offering?

AzCopy and the Azure Cosmos DB Data Migration Tool are both supported.

How is expansion of the storage size done for this service if, for example, I start with *n* GB of data and my data will grow to 1 TB over time?

Azure Cosmos DB is designed to provide unlimited storage via the use of horizontal scaling. The service can monitor and effectively increase your storage.

How do I monitor the Table API offering?

You can use the Table API **Metrics** pane to monitor requests and storage usage.

How do I calculate the throughput I require?

You can use the capacity estimator to calculate the TableThroughput that's required for the operations. For more information, see Estimate Request Units and Data Storage. In general, you can show your entity as JSON and provide the numbers for your operations.

Can I use the Table API SDK locally with the emulator?

Not at this time.

Can my existing application work with the Table API?

Yes, the same API is supported.

Do I need to migrate my existing Azure Table storage applications to the SDK if I don't want to use the Table API features?

No, you can create and use existing Azure Table storage assets without interruption of any kind. However, if you don't use the Table API, you can't benefit from the automatic index, the additional consistency option, or global distribution.

How do I add replication of the data in the Table API across more than one region of Azure?

You can use the Azure Cosmos DB portal's global replication settings to add regions that are suitable for your application. To develop a globally distributed application, you should also add your application with the PreferredLocation information set to the local region for providing low read latency.

How do I change the primary write region for the account in the Table API?

You can use the Azure Cosmos DB global replication portal pane to add a region and then fail over to the required region. For instructions, see Developing with multi-region Azure Cosmos DB accounts.

How do I configure my preferred read regions for low latency when I distribute my data?

To help read from the local location, use the PreferredLocation key in the app.config file. For existing applications, the Table API throws an error if LocationMode is set. Remove that code, because the Table API picks up this information from the app.config file.

How should I think about consistency levels in the Table API?

Azure Cosmos DB provides well-reasoned trade-offs between consistency, availability, and latency. Azure Cosmos DB offers five consistency levels to Table API developers, so you can choose the right consistency model at the table level and make individual requests while querying the data. When a client connects, it can specify a consistency level. You can change the level via the consistencyLevel argument of CreateCloudTableClient.

The Table API provides low-latency reads with "Read your own writes," with Bounded-staleness consistency as the default. For more information, see Consistency levels.

By default, Azure Table storage provides Strong consistency within a region and Eventual consistency in the secondary locations.

Does Azure Cosmos DB Table API offer more consistency levels than Azure Table storage?

Yes, for information about how to benefit from the distributed nature of Azure Cosmos DB, see Consistency levels. Because guarantees are provided for the consistency levels, you can use them with confidence.

When global distribution is enabled, how long does it take to replicate the data?

Azure Cosmos DB commits the data durably in the local region and pushes the data to other regions immediately in a matter of milliseconds. This replication is dependent only on the round-trip time (RTT) of the datacenter. To learn more about the global-distribution capability of Azure Cosmos DB, see Azure Cosmos DB: A globally distributed database service on Azure.

Can the read request consistency level be changed?

With Azure Cosmos DB, you can set the consistency level at the container level (on the table). By using the .NET SDK, you can change the level by providing the value for TableConsistencyLevel key in the app.config file. The possible values are: Strong, Bounded Staleness, Session, Consistent Prefix, and Eventual. For more information, see Tunable data consistency levels in Azure Cosmos DB. The key idea is that you can't set the request consistency level at more than the setting for the table. For example, you can't set the consistency level for the table at Eventual and the request consistency level at Strong.

How does the Table API handle failover if a region goes down?

The Table API leverages the globally distributed platform of Azure Cosmos DB. To ensure that your application can tolerate datacenter downtime, enable at least one more region for the account in the Azure Cosmos DB portal Developing with multi-region Azure Cosmos DB accounts. You can set the priority of the region by using the portal Developing with multi-region Azure Cosmos DB accounts.

You can add as many regions as you want for the account and control where it can fail over to by providing a failover priority. To use the database, you need to provide an application there too. When you do so, your customers won't experience downtime. The latest .NET client SDK is auto homing but the other SDKs aren't. That is, it can detect the region that's down and automatically fail over to the new region.

Is the Table API enabled for backups?

Yes, the Table API leverages the platform of Azure Cosmos DB for backups. Backups are made automatically. For more information, see Online backup and restore with Azure Cosmos DB.

Does the Table API index all attributes of an entity by default?

Yes, all attributes of an entity are indexed by default. For more information, see Azure Cosmos DB: Indexing policies.

Does this mean I don't have to create more than one index to satisfy the queries?

Yes, Azure Cosmos DB Table API provides automatic indexing of all attributes without any schema definition. This automation frees developers to focus on the application rather than on index creation and management. For more information, see Azure Cosmos DB: Indexing policies.

Can I change the indexing policy?

Yes, you can change the indexing policy by providing the index definition. You need to properly encode and escape the settings.

For the non-.NET SDKs, the indexing policy can only be set in the portal at **Data Explorer**, navigate to the specific table you want to change and then go to the **Scale & Settings**->Indexing Policy, make the desired change and then **Save**.

From the .NET SDK it can be submitted in the app.config file:

```
{
  "indexingMode": "consistent",
  "automatic": true,
  "includedPaths": [
    {
      "path": "/somepath",
      "indexes": [
        {
          "kind": "Range",
          "dataType": "Number",
          "precision": -1
        },
        {
          "kind": "Range",
          "dataType": "String",
          "precision": -1
        }
      ]
    }
  ],
  "excludedPaths":
[
 {
      "path": "/anotherpath"
 }
]
}
```

Azure Cosmos DB as a platform seems to have lot of capabilities, such as sorting, aggregates, hierarchy, and other functionality. Will you be adding these capabilities to the Table API?

The Table API provides the same query functionality as Azure Table storage. Azure Cosmos DB also supports sorting, aggregates, geospatial query, hierarchy, and a wide range of built-in functions. For more information, see SQL queries.

When should I change TableThroughput for the Table API?

You should change TableThroughput when either of the following conditions applies:

- You're performing an extract, transform, and load (ETL) of data, or you want to upload a lot of data in short amount of time.
- You need more throughput from the container or from a set of containers at the back end. For example, you see that the used throughput is more than the provisioned throughput, and you're getting throttled. For more information, see Set throughput for Azure Cosmos containers.

Can I scale up or scale down the throughput of my Table API table?

Yes, you can use the Azure Cosmos DB portal's scale pane to scale the throughput. For more information, see Set throughput.

Is a default TableThroughput set for newly provisioned tables?

Yes, if you don't override the TableThroughput via app.config and don't use a pre-created container in Azure Cosmos DB, the service creates a table with throughput of 400.

Is there any change of pricing for existing customers of the Azure Table storage service?

None. There's no change in price for existing Azure Table storage customers.

How is the price calculated for the Table API?

The price depends on the allocated TableThroughput.

How do I handle any rate limiting on the tables in Table API offering?

If the request rate is more than the capacity of the provisioned throughput for the underlying container or a set of

containers, you get an error, and the SDK retries the call by applying the retry policy.

Why do I need to choose a throughput apart from PartitionKey and RowKey to take advantage of the Table API offering of Azure Cosmos DB?

Azure Cosmos DB sets a default throughput for your container if you don't provide one in the app.config file or via the portal.

Azure Cosmos DB provides guarantees for performance and latency, with upper bounds on operation. This guarantee is possible when the engine can enforce governance on the tenant's operations. Setting TableThroughput ensures that you get the guaranteed throughput and latency, because the platform reserves this capacity and guarantees operational success.

By using the throughput specification, you can elastically change it to benefit from the seasonality of your application, meet the throughput needs, and save costs.

Azure Table storage has been inexpensive for me, because I pay only to store the data, and I rarely query. The Azure Cosmos DB Table API offering seems to be charging me even though I haven't performed a single transaction or stored anything. Can you explain?

Azure Cosmos DB is designed to be a globally distributed, SLA-based system with guarantees for availability, latency, and throughput. When you reserve throughput in Azure Cosmos DB, it's guaranteed, unlike the throughput of other systems. Azure Cosmos DB provides additional capabilities that customers have requested, such as secondary indexes and global distribution.

I never get a quota full" notification (indicating that a partition is full) when I ingest data into Azure Table storage. With the Table API, I do get this message. Is this offering limiting me and forcing me to change my existing application?

Azure Cosmos DB is an SLA-based system that provides unlimited scale, with guarantees for latency, throughput, availability, and consistency. To ensure guaranteed premium performance, make sure that your data size and index are manageable and scalable. The 20-GB limit on the number of entities or items per partition key is to ensure that we provide great lookup and query performance. To ensure that your application scales well, even for Azure Storage, we recommend that you *not* create a hot partition by storing all information in one partition and querying it.

So PartitionKey and RowKey are still required with the Table API?

Yes. Because the surface area of the Table API is similar to that of the Azure Table storage SDK, the partition key provides an efficient way to distribute the data. The row key is unique within that partition. The row key needs to be present and can't be null as in the standard SDK. The length of RowKey is 255 bytes and the length of PartitionKey is 1 KB.

What are the error messages for the Table API?

Azure Table storage and Azure Cosmos DB Table API use the same SDKs so most of the errors will be the same.

Why do I get throttled when I try to create lot of tables one after another in the Table API?

Azure Cosmos DB is an SLA-based system that provides latency, throughput, availability, and consistency guarantees. Because it's a provisioned system, it reserves resources to guarantee these requirements. The rapid rate of creation of tables is detected and throttled. We recommend that you look at the rate of creation of tables and lower it to less than 5 per minute. Remember that the Table API is a provisioned system. The moment you provision it, you'll begin to pay for it.

How do I provide feedback about the SDK or bugs?

You can share your feedback in any of the following ways:

- User voice
- Microsoft Q&A question page
- Stack Overflow. Stack Overflow is best for programming questions. Make sure your question is on-topic and provide as many details as possible, making the question clear and answerable.

Next steps

- Build a Table API app with .NET SDK and Azure Cosmos DB
- Build a Java app to manage Azure Cosmos DB Table API data

Developing with Azure Cosmos DB Table API and Azure Table storage

11/2/2020 • 3 minutes to read • Edit Online

APPLIES TO: Table API

Azure Cosmos DB Table API and Azure Table storage share the same table data model and expose the same create, delete, update, and query operations through their SDKs.

If you currently use Azure Table Storage, you gain the following benefits by moving to the Azure Cosmos DB Table API:

FEATURE	AZURE TABLE STORAGE	AZURE COSMOS DB TABLE API
Latency	Fast, but no upper bounds on latency.	Single-digit millisecond latency for reads and writes, backed with <10-ms latency reads and <15-ms latency writes at the 99th percentile, at any scale, anywhere in the world.
Throughput	Variable throughput model. Tables have a scalability limit of 20,000 operations/s.	Highly scalable with dedicated reserved throughput per table that's backed by SLAs. Accounts have no upper limit on throughput and support >10 million operations/s per table.
Global distribution	Single region with one optional readable secondary read region for high availability. You can't initiate failover.	Turnkey global distribution from one to 30+ regions. Support for automatic and manual failovers at any time, anywhere in the world.
Indexing	Only primary index on PartitionKey and RowKey. No secondary indexes.	Automatic and complete indexing on all properties, no index management.
Query	Query execution uses index for primary key, and scans otherwise.	Queries can take advantage of automatic indexing on properties for fast query times.
Consistency	Strong within primary region. Eventual within secondary region.	Five well-defined consistency levels to trade off availability, latency, throughput, and consistency based on your application needs.
Pricing	Storage-optimized.	Throughput-optimized.
SLAs	99.99% availability.	99.99% availability SLA for all single region accounts and all multi-region accounts with relaxed consistency, and 99.999% read availability on all multi-region database accounts Industry-leading comprehensive SLAs on general availability.

Developing with the Azure Cosmos DB Table API

At this time, the Azure Cosmos DB Table API has four SDKs available for development:

- Microsoft.Azure.Cosmos.Table: .NET SDK. This library targets .NET Standard and has the same classes and method signatures as the public Windows Azure Storage SDK, but also has the ability to connect to Azure Cosmos DB accounts using the Table API. Users of .NET Framework library Microsoft.Azure.CosmosDB.Table are recommended to upgrade to Microsoft.Azure.Cosmos.Table as it is in maintenance mode and will be deprecated soon.
- Python SDK: The new Azure Cosmos DB Python SDK is the only SDK that supports Azure Table storage in Python. This SDK connects with both Azure Table storage and Azure Cosmos DB Table API.
- Java SDK: This Azure Storage SDK has the ability to connect to Azure Cosmos DB accounts using the Table API.
- Node.js SDK: This Azure Storage SDK has the ability to connect to Azure Cosmos DB accounts using the Table API.

Additional information about working with the Table API is available in the FAQ: Develop with the Table API article.

Developing with Azure Table storage

Azure Table storage has these SDKs available for development:

- The Microsoft.Azure.Storage.Blob, Microsoft.Azure.Storage.File, Microsoft.Azure.Storage.Queue, and Microsoft.Azure.Storage.Common libraries allow you to work with the Azure Table storage service. If you are using the Table API in Azure Cosmos DB, you can instead use the Microsoft.Azure.CosmosDB.Table library.
- Python SDK. The Azure Cosmos DB Table SDK for Python supports the Table Storage service (because Azure Table Storage and Cosmos DB's Table API share the same features and functionalities, and in an effort to factorize our SDK development efforts, we recommend to use this SDK).
- Azure Storage SDK for Java. This Azure Storage SDK provides a client library in Java to consume Azure Table storage.
- Node.js SDK. This SDK provides a Node.js package and a browser-compatible JavaScript client library to consume the storage Table service.
- AzureRmStorageTable PowerShell module. This PowerShell module has cmdlets to work with storage Tables.
- Azure Storage Client Library for C++. This library enables you to build applications against Azure Storage.
- Azure Storage Table Client Library for Ruby. This project provides a Ruby package that makes it easy to access Azure storage Table services.
- Azure Storage Table PHP Client Library. This project provides a PHP client library that makes it easy to access Azure storage Table services.

Create a container in Azure Cosmos DB Table API

11/2/2020 • 2 minutes to read • Edit Online

APPLIES TO: Table API

This article explains the different ways to create a container in Azure Cosmos DB Table API. It shows how to create a container using Azure portal, Azure CLI, PowerShell, or supported SDKs. This article demonstrates how to create a container, specify the partition key, and provision throughput.

This article explains the different ways to create a container in Azure Cosmos DB Table API. If you are using a different API, see API for MongoDB, Cassandra API, Gremlin API, and SQL API articles to create the container.

> **NOTE**
> When creating containers, make sure you don't create two containers with the same name but different casing. That's because some parts of the Azure platform are not case-sensitive, and this can result in confusion/collision of telemetry and actions on containers with such names.

Create using Azure portal

1. Sign in to the Azure portal.
2. Create a new Azure Cosmos account, or selectan existing account.
3. Open the **Data Explorer** pane, and select **New Table**. Next, provide the following details:
   - Enter a Table ID.
   - Enter a throughput to be provisioned (for example, 1000 RUs).
   - Select **OK**.

> **NOTE**
> For Table API, the partition key is specified each time you add a new row.

Create using Azure CLI

Create a Table API table with Azure CLI. For a listing of all Azure CLI samples across all Azure Cosmos DB APIs see,

Azure CLI samples for Azure Cosmos DB.

Create using PowerShell

Create a Table API table with PowerShell. For a listing of all PowerShell samples across all Azure Cosmos DB APIs see, PowerShell Samples

Next steps

- Partitioning in Azure Cosmos DB
- Request Units in Azure Cosmos DB
- Provision throughput on containers and databases
- Work with Azure Cosmos account

Find the request unit charge for operations executed in Azure Cosmos DB Table API

11/2/2020 • 2 minutes to read • Edit Online

APPLIES TO: Table API

Azure Cosmos DB supports many APIs, such as SQL, MongoDB, Cassandra, Gremlin, and Table. Each API has its own set of database operations. These operations range from simple point reads and writes to complex queries. Each database operation consumes system resources based on the complexity of the operation.

The cost of all database operations is normalized by Azure Cosmos DB and is expressed by Request Units (or RUs, for short). You can think of RUs as a performance currency abstracting the system resources such as CPU, IOPS, and memory that are required to perform the database operations supported by Azure Cosmos DB. No matter which API you use to interact with your Azure Cosmos container, costs are always measured by RUs. Whether the database operation is a write, point read, or query, costs are always measured in RUs. To learn more, see the request units and it's considerations article.

This article presents the different ways you can find the request unit (RU) consumption for any operation executed against a container in Azure Cosmos DB Table API. If you are using a different API, see API for MongoDB, Cassandra API, Gremlin API, and SQL API articles to find the RU/s charge.

Use the .NET SDK

Currently, the only SDK that returns the RU charge for table operations is the .NET Standard SDK. The `TableResult` object exposes a `RequestCharge` property that is populated by the SDK when you use it against the Azure Cosmos DB Table API:

```
CloudTable tableReference = client.GetTableReference("table");
TableResult tableResult = tableReference.Execute(TableOperation.Insert(new DynamicTableEntity("partitionKey",
"rowKey")));
if (tableResult.RequestCharge.HasValue) // would be false when using Azure Storage Tables
{
    double requestCharge = tableResult.RequestCharge.Value;
}
```

For more information, see Quickstart: Build a Table API app by using the .NET SDK and Azure Cosmos DB.

Next steps

To learn about optimizing your RU consumption, see these articles:

- Request units and throughput in Azure Cosmos DB
- Optimize provisioned throughput cost in Azure Cosmos DB
- Optimize query cost in Azure Cosmos DB

Azure Table storage table design guide: Scalable and performant tables

11/2/2020 • 74 minutes to read • Edit Online

APPLIES TO: Table API

> **TIP**
> The content in this article applies to the original Azure Table storage. However, there is now a premium offering for table storage: the Azure Cosmos DB Table API. This API offers throughput-optimized tables, global distribution, and automatic secondary indexes. There are some feature differences between Table API in Azure Cosmos DB and Azure table storage. For more information, and to try out the premium experience, see Azure Cosmos DB Table API.

To design scalable and performant tables, you must consider a variety of factors, including cost. If you've previously designed schemas for relational databases, these considerations will be familiar to you. But while there are some similarities between Azure Table storage and relational models, there are also many important differences. These differences typically lead to different designs that might look counter-intuitive or wrong to someone familiar with relational databases, but that do make sense if you're designing for a NoSQL key/value store, such as Table storage.

Table storage is designed to support cloud-scale applications that can contain billions of entities ("rows" in relational database terminology) of data, or for datasets that must support high transaction volumes. You therefore need to think differently about how you store your data, and understand how Table storage works. A well-designed NoSQL data store can enable your solution to scale much further (and at a lower cost) than a solution that uses a relational database. This guide helps you with these topics.

About Azure Table storage

This section highlights some of the key features of Table storage that are especially relevant to designing for performance and scalability. If you're new to Azure Storage and Table storage, see Introduction to Microsoft Azure Storage and Get started with Azure Table storage by using .NET before reading the remainder of this article. Although the focus of this guide is on Table storage, it does include some discussion of Azure Queue storage and Azure Blob storage, and how you might use them along with Table storage in a solution.

Table storage uses a tabular format to store data. In the standard terminology, each row of the table represents an entity, and the columns store the various properties of that entity. Every entity has a pair of keys to uniquely identify it, and a timestamp column that Table storage uses to track when the entity was last updated. The timestamp field is added automatically, and you can't manually overwrite the timestamp with an arbitrary value. Table storage uses this last-modified timestamp (LMT) to manage optimistic concurrency.

> **NOTE**
> Table storage REST API operations also return an `ETag` value that it derives from the LMT. In this document, the terms ETag and LMT are used interchangeably, because they refer to the same underlying data.

The following example shows a simple table design to store employee and department entities. Many of the examples shown later in this guide are based on this simple design.

PARTITIONKEY	ROWKEY	TIMESTAMP	

Marketing	00001	2014-08-22T00:50:32Z	<table><tr><th>FIRSTNAME</th><th>LASTNAME</th><th>AGE</th><th>EMAIL</th></tr><tr><td>Don</td><td>Hall</td><td>34</td><td>donh@contoso.com</td></tr></table>
Marketing	00002	2014-08-22T00:50:34Z	<table><tr><th>FIRSTNAME</th><th>LASTNAME</th><th>AGE</th><th>EMAIL</th></tr><tr><td>Jun</td><td>Cao</td><td>47</td><td>junc@contoso.com</td></tr></table>
Marketing	Department	2014-08-22T00:50:30Z	<table><tr><th>DEPARTMENTNAME</th><th>EMPLOYEECOUNT</th></tr><tr><td>Marketing</td><td>153</td></tr></table>
Sales	00010	2014-08-22T00:50:44Z	<table><tr><th>FIRSTNAME</th><th>LASTNAME</th><th>AGE</th><th>EMAIL</th></tr><tr><td>Ken</td><td>Kwok</td><td>23</td><td>kenk@contoso.com</td></tr></table>

So far, this design looks similar to a table in a relational database. The key differences are the mandatory columns and the ability to store multiple entity types in the same table. In addition, each of the user-defined properties, such as **FirstName** or **Age**, has a data type, such as integer or string, just like a column in a relational database. Unlike in a relational database, however, the schema-less nature of Table storage means that a property need not have the same data type on each entity. To store complex data types in a single property, you must use a serialized format such as JSON or XML. For more information, see Understanding Table storage data model.

Your choice of `PartitionKey` and `RowKey` is fundamental to good table design. Every entity stored in a table must have a unique combination of `PartitionKey` and `RowKey`. As with keys in a relational database table, the `PartitionKey` and `RowKey` values are indexed to create a clustered index that enables fast look-ups. Table storage, however, doesn't create any secondary indexes, so these are the only two indexed properties (some of the patterns described later show how you can work around this apparent limitation).

A table is made up of one or more partitions, and many of the design decisions you make will be around choosing a suitable `PartitionKey` and `RowKey` to optimize your solution. A solution can consist of just a single table that contains all your entities organized into partitions, but typically a solution has multiple tables. Tables help you to

logically organize your entities, and help you manage access to the data by using access control lists. You can drop an entire table by using a single storage operation.

Table partitions

The account name, table name, and `PartitionKey` together identify the partition within the storage service where Table storage stores the entity. As well as being part of the addressing scheme for entities, partitions define a scope for transactions (see the section later in this article, Entity group transactions), and form the basis of how Table storage scales. For more information on table partitions, see Performance and scalability checklist for Table storage.

In Table storage, an individual node services one or more complete partitions, and the service scales by dynamically load-balancing partitions across nodes. If a node is under load, Table storage can split the range of partitions serviced by that node onto different nodes. When traffic subsides, Table storage can merge the partition ranges from quiet nodes back onto a single node.

For more information about the internal details of Table storage, and in particular how it manages partitions, see Microsoft Azure Storage: A highly available cloud storage service with strong consistency.

Entity group transactions

In Table storage, entity group transactions (EGTs) are the only built-in mechanism for performing atomic updates across multiple entities. EGTs are also referred to as *batch transactions*. EGTs can only operate on entities stored in the same partition (sharing the same partition key in a particular table), so anytime you need atomic transactional behavior across multiple entities, ensure that those entities are in the same partition. This is often a reason for keeping multiple entity types in the same table (and partition), and not using multiple tables for different entity types. A single EGT can operate on at most 100 entities. If you submit multiple concurrent EGTs for processing, it's important to ensure that those EGTs don't operate on entities that are common across EGTs. Otherwise, you risk delaying processing.

EGTs also introduce a potential trade-off for you to evaluate in your design. Using more partitions increases the scalability of your application, because Azure has more opportunities for load-balancing requests across nodes. But this might limit the ability of your application to perform atomic transactions and maintain strong consistency for your data. Furthermore, there are specific scalability targets at the level of a partition that might limit the throughput of transactions you can expect for a single node.

For more information about scalability targets for Azure storage accounts, see Scalability targets for standard storage accounts. For more information about scalability targets for Table storage, see Scalability and performance targets for Table storage. Later sections of this guide discuss various design strategies that help you manage trade-offs such as this one, and discuss how best to choose your partition key based on the specific requirements of your client application.

Capacity considerations

The following table includes some of the key values to be aware of when you're designing a Table storage solution:

TOTAL CAPACITY OF AN AZURE STORAGE ACCOUNT	500 TB
Number of tables in an Azure storage account	Limited only by the capacity of the storage account.
Number of partitions in a table	Limited only by the capacity of the storage account.
Number of entities in a partition	Limited only by the capacity of the storage account.
Size of an individual entity	Up to 1 MB, with a maximum of 255 properties (including the `PartitionKey`, `RowKey`, and `Timestamp`).
Size of the `PartitionKey`	A string up to 1 KB in size.

TOTAL CAPACITY OF AN AZURE STORAGE ACCOUNT	500 TB
Size of the `RowKey`	A string up to 1 KB in size.
Size of an entity group transaction	A transaction can include at most 100 entities, and the payload must be less than 4 MB in size. An EGT can only update an entity once.

For more information, see Understanding the Table service data model.

Cost considerations

Table storage is relatively inexpensive, but you should include cost estimates for both capacity usage and the quantity of transactions as part of your evaluation of any solution that uses Table storage. In many scenarios, however, storing denormalized or duplicate data in order to improve the performance or scalability of your solution is a valid approach to take. For more information about pricing, see Azure Storage pricing.

Guidelines for table design

These lists summarize some of the key guidelines you should keep in mind when you're designing your tables. This guide addresses them all in more detail later on. These guidelines are different from the guidelines you'd typically follow for relational database design.

Designing your Table storage to be *read* efficient:

- **Design for querying in read-heavy applications.** When you're designing your tables, think about the queries (especially the latency-sensitive ones) you'll run before you think about how you'll update your entities. This typically results in an efficient and performant solution.
- **Specify both `PartitionKey` and `RowKey` in your queries.** *Point queries* such as these are the most efficient Table storage queries.
- **Consider storing duplicate copies of entities.** Table storage is cheap, so consider storing the same entity multiple times (with different keys), to enable more efficient queries.
- **Consider denormalizing your data.** Table storage is cheap, so consider denormalizing your data. For example, store summary entities so that queries for aggregate data only need to access a single entity.
- **Use compound key values.** The only keys you have are `PartitionKey` and `RowKey`. For example, use compound key values to enable alternate keyed access paths to entities.
- **Use query projection.** You can reduce the amount of data that you transfer over the network by using queries that select just the fields you need.

Designing your Table storage to be *write* efficient:

- **Don't create hot partitions.** Choose keys that enable you to spread your requests across multiple partitions at any point of time.
- **Avoid spikes in traffic.** Distribute the traffic over a reasonable period of time, and avoid spikes in traffic.
- **Don't necessarily create a separate table for each type of entity.** When you require atomic transactions across entity types, you can store these multiple entity types in the same partition in the same table.
- **Consider the maximum throughput you must achieve.** You must be aware of the scalability targets for Table storage, and ensure that your design won't cause you to exceed them.

Later in this guide, you'll see examples that put all of these principles into practice.

Design for querying

Table storage can be read intensive, write intensive, or a mix of the two. This section considers designing to support read operations efficiently. Typically, a design that supports read operations efficiently is also efficient for write

operations. However, there are additional considerations when designing to support write operations. These are discussed in the next section, Design for data modification.

A good starting point to enable you to read data efficiently is to ask "What queries will my application need to run to retrieve the data it needs?"

> **NOTE**
> With Table storage, it's important to get the design correct up front, because it's difficult and expensive to change it later. For example, in a relational database, it's often possible to address performance issues simply by adding indexes to an existing database. This isn't an option with Table storage.

How your choice of `PartitionKey` and `RowKey` affects query performance

The following examples assume Table storage is storing employee entities with the following structure (most of the examples omit the `Timestamp` property for clarity):

COLUMN NAME	DATA TYPE
`PartitionKey` (Department name)	String
`RowKey` (Employee ID)	String
`FirstName`	String
`LastName`	String
`Age`	Integer
`EmailAddress`	String

Here are some general guidelines for designing Table storage queries. The filter syntax used in the following examples is from the Table storage REST API. For more information, see Query entities.

- A *point query* is the most efficient lookup to use, and is recommended for high-volume lookups or lookups requiring the lowest latency. Such a query can use the indexes to locate an individual entity efficiently by specifying both the `PartitionKey` and `RowKey` values. For example: `$filter=(PartitionKey eq 'Sales') and (RowKey eq '2')`.
- Second best is a *range query*. It uses the `PartitionKey`, and filters on a range of `RowKey` values to return more than one entity. The `PartitionKey` value identifies a specific partition, and the `RowKey` values identify a subset of the entities in that partition. For example: `$filter=PartitionKey eq 'Sales' and RowKey ge 'S' and RowKey lt 'T'`.
- Third best is a *partition scan*. It uses the `PartitionKey`, and filters on another non-key property and might return more than one entity. The `PartitionKey` value identifies a specific partition, and the property values select for a subset of the entities in that partition. For example: `$filter=PartitionKey eq 'Sales' and LastName eq 'Smith'`.
- A *table scan* doesn't include the `PartitionKey`, and is inefficient because it searches all of the partitions that make up your table for any matching entities. It performs a table scan regardless of whether or not your filter uses the `RowKey`. For example: `$filter=LastName eq 'Jones'`.
- Azure Table storage queries that return multiple entities sort them in `PartitionKey` and `RowKey` order. To avoid resorting the entities in the client, choose a `RowKey` that defines the most common sort order. Query results returned by the Azure Table API in Azure Cosmos DB aren't sorted by partition key or row key. For a detailed list of feature differences, see differences between Table API in Azure Cosmos DB and Azure Table storage.

Using an "**or**" to specify a filter based on `RowKey` values results in a partition scan, and isn't treated as a range query. Therefore, avoid queries that use filters such as: `$filter=PartitionKey eq 'Sales' and (RowKey eq '121' or RowKey eq '322')`.

For examples of client-side code that use the Storage Client Library to run efficient queries, see:

- Run a point query by using the Storage Client Library
- Retrieve multiple entities by using LINQ
- Server-side projection

For examples of client-side code that can handle multiple entity types stored in the same table, see:

- Work with heterogeneous entity types

Choose an appropriate `PartitionKey`

Your choice of `PartitionKey` should balance the need to enable the use of EGTs (to ensure consistency) against the requirement to distribute your entities across multiple partitions (to ensure a scalable solution).

At one extreme, you can store all your entities in a single partition. But this might limit the scalability of your solution, and would prevent Table storage from being able to load-balance requests. At the other extreme, you can store one entity per partition. This is highly scalable and enables Table storage to load-balance requests, but prevents you from using entity group transactions.

An ideal `PartitionKey` enables you to use efficient queries, and has sufficient partitions to ensure your solution is scalable. Typically, you'll find that your entities will have a suitable property that distributes your entities across sufficient partitions.

> **NOTE**
> For example, in a system that stores information about users or employees, `UserID` can be a good `PartitionKey`. You might have several entities that use a particular `UserID` as the partition key. Each entity that stores data about a user is grouped into a single partition. These entities are accessible via EGTs, while still being highly scalable.

There are additional considerations in your choice of `PartitionKey` that relate to how you insert, update, and delete entities. For more information, see Design for data modification later in this article.

Optimize queries for Table storage

Table storage automatically indexes your entities by using the `PartitionKey` and `RowKey` values in a single clustered index. This is the reason that point queries are the most efficient to use. However, there are no indexes other than that on the clustered index on the `PartitionKey` and `RowKey`.

Many designs must meet requirements to enable lookup of entities based on multiple criteria. For example, locating employee entities based on email, employee ID, or last name. The following patterns in the section Table design patterns address these types of requirements. The patterns also describe ways of working around the fact that Table storage doesn't provide secondary indexes.

- Intra-partition secondary index pattern: Store multiple copies of each entity by using different `RowKey` values (in the same partition). This enables fast and efficient lookups, and alternate sort orders by using different `RowKey` values.
- Inter-partition secondary index pattern: Store multiple copies of each entity by using different `RowKey` values in separate partitions or in separate tables. This enables fast and efficient lookups, and alternate sort orders by using different `RowKey` values.
- Index entities pattern: Maintain index entities to enable efficient searches that return lists of entities.

Sort data in Table storage

Table storage returns query results sorted in ascending order, based on `PartitionKey` and then by `RowKey`.

> **NOTE**
> Query results returned by the Azure Table API in Azure Cosmos DB aren't sorted by partition key or row key. For a detailed list of feature differences, see differences between Table API in Azure Cosmos DB and Azure Table storage.

Keys in Table storage are string values. To ensure that numeric values sort correctly, you should convert them to a fixed length, and pad them with zeroes. For example, if the employee ID value you use as the `RowKey` is an integer value, you should convert employee ID `123` to `00000123`.

Many applications have requirements to use data sorted in different orders: for example, sorting employees by name, or by joining date. The following patterns in the section Table design patterns address how to alternate sort orders for your entities:

- Intra-partition secondary index pattern: Store multiple copies of each entity by using different `RowKey` values (in the same partition). This enables fast and efficient lookups, and alternate sort orders by using different `RowKey` values.
- Inter-partition secondary index pattern: Store multiple copies of each entity by using different `RowKey` values in separate partitions in separate tables. This enables fast and efficient lookups, and alternate sort orders by using different `RowKey` values.
- Log tail pattern: Retrieve the *n* entities most recently added to a partition, by using a `RowKey` value that sorts in reverse date and time order.

Design for data modification

This section focuses on the design considerations for optimizing inserts, updates, and deletes. In some cases, you'll need to evaluate the trade-off between designs that optimize for querying against designs that optimize for data modification. This evaluation is similar to what you do in designs for relational databases (although the techniques for managing the design trade-offs are different in a relational database). The section Table design patterns describes some detailed design patterns for Table storage, and highlights some of these trade-offs. In practice, you'll find that many designs optimized for querying entities also work well for modifying entities.

Optimize the performance of insert, update, and delete operations

To update or delete an entity, you must be able to identify it by using the `PartitionKey` and `RowKey` values. In this respect, your choice of `PartitionKey` and `RowKey` for modifying entities should follow similar criteria to your choice to support point queries. You want to identify entities as efficiently as possible. You don't want to use an inefficient partition or table scan to locate an entity in order to discover the `PartitionKey` and `RowKey` values you need to update or delete it.

The following patterns in the section Table design patterns address optimizing the performance of your insert, update, and delete operations:

- High volume delete pattern: Enable the deletion of a high volume of entities by storing all the entities for simultaneous deletion in their own separate table. You delete the entities by deleting the table.
- Data series pattern: Store complete data series in a single entity to minimize the number of requests you make.
- Wide entities pattern: Use multiple physical entities to store logical entities with more than 252 properties.
- Large entities pattern: Use blob storage to store large property values.

Ensure consistency in your stored entities

The other key factor that influences your choice of keys for optimizing data modifications is how to ensure consistency by using atomic transactions. You can only use an EGT to operate on entities stored in the same partition.

The following patterns in the section Table design patterns address managing consistency:

- Intra-partition secondary index pattern: Store multiple copies of each entity by using different `RowKey` values (in the same partition). This enables fast and efficient lookups, and alternate sort orders by using different `RowKey` values.
- Inter-partition secondary index pattern: Store multiple copies of each entity by using different `RowKey` values in separate partitions or in separate tables. This enables fast and efficient lookups, and alternate sort orders by using different `RowKey` values.
- Eventually consistent transactions pattern: Enable eventually consistent behavior across partition boundaries or storage system boundaries by using Azure queues.
- Index entities pattern: Maintain index entities to enable efficient searches that return lists of entities.
- Denormalization pattern: Combine related data together in a single entity, to enable you to retrieve all the data you need with a single point query.
- Data series pattern: Store complete data series in a single entity, to minimize the number of requests you make.

For more information, see Entity group transactions later in this article.

Ensure your design for efficient modifications facilitates efficient queries

In many cases, a design for efficient querying results in efficient modifications, but you should always evaluate whether this is the case for your specific scenario. Some of the patterns in the section Table design patterns explicitly evaluate trade-offs between querying and modifying entities, and you should always take into account the number of each type of operation.

The following patterns in the section Table design patterns address trade-offs between designing for efficient queries and designing for efficient data modification:

- Compound key pattern: Use compound `RowKey` values to enable a client to look up related data with a single point query.
- Log tail pattern: Retrieve the *n* entities most recently added to a partition, by using a `RowKey` value that sorts in reverse date and time order.

Encrypt table data

The .NET Azure Storage client library supports encryption of string entity properties for insert and replace operations. The encrypted strings are stored on the service as binary properties, and they're converted back to strings after decryption.

For tables, in addition to the encryption policy, users must specify the properties to be encrypted. Either specify an `EncryptProperty` attribute (for POCO entities that derive from `TableEntity`), or specify an encryption resolver in request options. An encryption resolver is a delegate that takes a partition key, row key, and property name, and returns a Boolean that indicates whether that property should be encrypted. During encryption, the client library uses this information to decide whether a property should be encrypted while writing to the wire. The delegate also provides for the possibility of logic around how properties are encrypted. (For example, if X, then encrypt property A; otherwise encrypt properties A and B.) It's not necessary to provide this information while reading or querying entities.

Merge isn't currently supported. Because a subset of properties might have been encrypted previously by using a different key, simply merging the new properties and updating the metadata will result in data loss. Merging either requires making extra service calls to read the pre-existing entity from the service, or using a new key per property. Neither of these are suitable for performance reasons.

For information about encrypting table data, see Client-side encryption and Azure Key Vault for Microsoft Azure Storage.

Model relationships

Building domain models is a key step in the design of complex systems. Typically, you use the modeling process to identify entities and the relationships between them, as a way to understand the business domain and inform the design of your system. This section focuses on how you can translate some of the common relationship types found in domain models to designs for Table storage. The process of mapping from a logical data model to a physical NoSQL-based data model is different from that used when designing a relational database. Relational databases design typically assumes a data normalization process optimized for minimizing redundancy. Such design also assumes a declarative querying capability that abstracts the implementation of how the database works.

One-to-many relationships

One-to-many relationships between business domain objects occur frequently: for example, one department has many employees. There are several ways to implement one-to-many relationships in Table storage, each with pros and cons that might be relevant to the particular scenario.

Consider the example of a large multinational corporation with tens of thousands of departments and employee entities. Every department has many employees and each employee is associated with one specific department. One approach is to store separate department and employee entities, such as the following:

This example shows an implicit one-to-many relationship between the types, based on the `PartitionKey` value. Each department can have many employees.

This example also shows a department entity and its related employee entities in the same partition. You can choose to use different partitions, tables, or even storage accounts for the different entity types.

An alternative approach is to denormalize your data, and store only employee entities with denormalized department data, as shown in the following example. In this particular scenario, this denormalized approach might not be the best if you have a requirement to be able to change the details of a department manager. To do this, you would need to update every employee in the department.

For more information, see the Denormalization pattern later in this guide.

The following table summarizes the pros and cons of each of the approaches for storing employee and department entities that have a one-to-many relationship. You should also consider how often you expect to perform various operations. It might be acceptable to have a design that includes an expensive operation if that operation only happens infrequently.

APPROACH	PROS	CONS

Separate entity types, same partition, same table	• You can update a department entity with a single operation. • You can use an EGT to maintain consistency if you have a requirement to modify a department entity whenever you update/insert/delete an employee entity. For example, if you maintain a departmental employee count for each department.	• You might need to retrieve both an employee and a department entity for some client activities. • Storage operations happen in the same partition. At high transaction volumes, this can result in a hotspot. • You can't move an employee to a new department by using an EGT.
Separate entity types, different partitions, or tables or storage accounts	• You can update a department entity or employee entity with a single operation. • At high transaction volumes, this can help spread the load across more partitions.	• You might need to retrieve both an employee and a department entity for some client activities. • You can't use EGTs to maintain consistency when you update/insert/delete an employee and update a department. For example, updating an employee count in a department entity. • You can't move an employee to a new department by using an EGT.
Denormalize into single entity type	• You can retrieve all the information you need with a single request.	• It can be expensive to maintain consistency if you need to update department information (this would require you to update all the employees in a department).

How you choose among these options, and which of the pros and cons are most significant, depends on your specific application scenarios. For example, how often do you modify department entities? Do all your employee queries need the additional departmental information? How close are you to the scalability limits on your partitions or your storage account?

One-to-one relationships

Domain models can include one-to-one relationships between entities. If you need to implement a one-to-one relationship in Table storage, you must also choose how to link the two related entities when you need to retrieve them both. This link can be either implicit, based on a convention in the key values, or explicit, by storing a link in the form of `PartitionKey` and `RowKey` values in each entity to its related entity. For a discussion of whether you should store the related entities in the same partition, see the section One-to-many relationships.

There are also implementation considerations that might lead you to implement one-to-one relationships in Table storage:

- Handling large entities (for more information, see Large entities pattern).
- Implementing access controls (for more information, see Control access with shared access signatures).

Join in the client

Although there are ways to model relationships in Table storage, don't forget that the two prime reasons for using Table storage are scalability and performance. If you find you are modeling many relationships that compromise the performance and scalability of your solution, you should ask yourself if it's necessary to build all the data relationships into your table design. You might be able to simplify the design, and improve the scalability and

performance of your solution, if you let your client application perform any necessary joins.

For example, if you have small tables that contain data that doesn't change often, you can retrieve this data once, and cache it on the client. This can avoid repeated roundtrips to retrieve the same data. In the examples we've looked at in this guide, the set of departments in a small organization is likely to be small and change infrequently. This makes it a good candidate for data that a client application can download once and cache as lookup data.

Inheritance relationships

If your client application uses a set of classes that form part of an inheritance relationship to represent business entities, you can easily persist those entities in Table storage. For example, you might have the following set of classes defined in your client application, where `Person` is an abstract class.

You can persist instances of the two concrete classes in Table storage by using a single `Person` table. Use entities that look like the following:

Customer entity	Employee entity
PartitionKey	PartitionKey
RowKey	RowKey
PersonType ("Customer")	PersonType ("Employee")
FirstName (string)	FirstName (string)
LastName (string)	LastName (string)
EmailAddress (string)	EmailAddress (string)
Address (string)	Department (string)
Phone (string)	EmployeeID (string)

For more information about working with multiple entity types in the same table in client code, see Work with heterogeneous entity types later in this guide. This provides examples of how to recognize the entity type in client code.

Table design patterns

In previous sections, you learned about how to optimize your table design for both retrieving entity data by using queries, and for inserting, updating, and deleting entity data. This section describes some patterns appropriate for use with Table storage. In addition, you'll see how you can practically address some of the issues and trade-offs raised previously in this guide. The following diagram summarizes the relationships among the different patterns:

The pattern map highlights some relationships between patterns (blue) and anti-patterns (orange) that are documented in this guide. There are of course many other patterns that are worth considering. For example, one of the key scenarios for Table storage is to use the materialized view pattern from the command query responsibility segregation pattern.

Intra-partition secondary index pattern

Store multiple copies of each entity by using different `RowKey` values (in the same partition). This enables fast and efficient lookups, and alternate sort orders by using different `RowKey` values. Updates between copies can be kept consistent by using EGTs.

Context and problem

Table storage automatically indexes entities by using the `PartitionKey` and `RowKey` values. This enables a client application to retrieve an entity efficiently by using these values. For example, using the following table structure, a client application can use a point query to retrieve an individual employee entity by using the department name and the employee ID (the `PartitionKey` and `RowKey` values). A client can also retrieve entities sorted by employee ID within each department.

Employee entity
PartitionKey (Department name) RowKey (Employee Id)
FirstName (string) LastName (string) Age (integer) EmailAddress (string)

If you also want to find an employee entity based on the value of another property, such as email address, you must use a less efficient partition scan to find a match. This is because Table storage doesn't provide secondary indexes. In addition, there's no option to request a list of employees sorted in a different order than `RowKey` order.

Solution

To work around the lack of secondary indexes, you can store multiple copies of each entity, with each copy using a

different `RowKey` value. If you store an entity with the following structures, you can efficiently retrieve employee entities based on email address or employee ID. The prefix values for `RowKey`, `empid_`, and `email_` enable you to query for a single employee, or a range of employees, by using a range of email addresses or employee IDs.

The following two filter criteria (one looking up by employee ID, and one looking up by email address) both specify point queries:

- $filter=(PartitionKey eq 'Sales') and (RowKey eq 'empid_000223')
- $filter=(PartitionKey eq 'Sales') and (RowKey eq 'email_jonesj@contoso.com')

If you query for a range of employee entities, you can specify a range sorted in employee ID order, or a range sorted in email address order. Query for entities with the appropriate prefix in the `RowKey`.

- To find all the employees in the Sales department with an employee ID in the range 000100 to 000199, use: $filter=(PartitionKey eq 'Sales') and (RowKey ge 'empid_000100') and (RowKey le 'empid_000199')
- To find all the employees in the Sales department with an email address starting with the letter "a", use: $filter=(PartitionKey eq 'Sales') and (RowKey ge 'email_a') and (RowKey lt 'email_b')

The filter syntax used in the preceding examples is from the Table storage REST API. For more information, see Query entities.

Issues and considerations

Consider the following points when deciding how to implement this pattern:

- Table storage is relatively cheap to use, so the cost overhead of storing duplicate data shouldn't be a major concern. However, you should always evaluate the cost of your design based on your anticipated storage requirements, and only add duplicate entities to support the queries your client application will run.
- Because the secondary index entities are stored in the same partition as the original entities, ensure that you don't exceed the scalability targets for an individual partition.
- You can keep your duplicate entities consistent with each other by using EGTs to update the two copies of the entity atomically. This implies that you should store all copies of an entity in the same partition. For more information, see Use entity group transactions.
- The value used for the `RowKey` must be unique for each entity. Consider using compound key values.
- Padding numeric values in the `RowKey` (for example, the employee ID 000223) enables correct sorting and filtering based on upper and lower bounds.
- You don't necessarily need to duplicate all the properties of your entity. For example, if the queries that look up the entities by using the email address in the `RowKey` never need the employee's age, these entities can have the following structure:

- Typically, it's better to store duplicate data and ensure that you can retrieve all the data you need with a single query, than to use one query to locate an entity and another to look up the required data.

When to use this pattern

Use this pattern when:

- Your client application needs to retrieve entities by using a variety of different keys.
- Your client needs to retrieve entities in different sort orders.
- You can identify each entity by using a variety of unique values.

However, be sure that you don't exceed the partition scalability limits when you're performing entity lookups by using the different `RowKey` values.

Related patterns and guidance

The following patterns and guidance might also be relevant when implementing this pattern:

- Inter-partition secondary index pattern
- Compound key pattern
- Entity group transactions
- Work with heterogeneous entity types

Inter-partition secondary index pattern

Store multiple copies of each entity by using different `RowKey` values in separate partitions or in separate tables. This enables fast and efficient lookups, and alternate sort orders by using different `RowKey` values.

Context and problem

Table storage automatically indexes entities by using the `PartitionKey` and `RowKey` values. This enables a client application to retrieve an entity efficiently by using these values. For example, using the following table structure, a client application can use a point query to retrieve an individual employee entity by using the department name and the employee ID (the `PartitionKey` and `RowKey` values). A client can also retrieve entities sorted by employee ID within each department.

[9]

If you also want to be able to find an employee entity based on the value of another property, such as email address, you must use a less efficient partition scan to find a match. This is because Table storage doesn't provide secondary indexes. In addition, there's no option to request a list of employees sorted in a different order than `RowKey` order.

You're anticipating a high volume of transactions against these entities, and want to minimize the risk of the Table storage rate limiting your client.

Solution

To work around the lack of secondary indexes, you can store multiple copies of each entity, with each copy using different `PartitionKey` and `RowKey` values. If you store an entity with the following structures, you can efficiently retrieve employee entities based on email address or employee ID. The prefix values for `PartitionKey`, `empid_`, and `email_` enable you to identify which index you want to use for a query.

The following two filter criteria (one looking up by employee ID, and one looking up by email address) both specify point queries:

- $filter=(PartitionKey eq 'empid_Sales') and (RowKey eq '000223')
- $filter=(PartitionKey eq 'email_Sales') and (RowKey eq 'jonesj@contoso.com')

If you query for a range of employee entities, you can specify a range sorted in employee ID order, or a range sorted in email address order. Query for entities with the appropriate prefix in the `RowKey`.

- To find all the employees in the Sales department with an employee ID in the range **000100** to **000199**, sorted in employee ID order, use: $filter=(PartitionKey eq 'empid_Sales') and (RowKey ge '000100') and (RowKey le '000199')
- To find all the employees in the Sales department with an email address that starts with "a", sorted in email address order, use: $filter=(PartitionKey eq 'email_Sales') and (RowKey ge 'a') and (RowKey lt 'b')

Note that the filter syntax used in the preceding examples is from the Table storage REST API. For more information, see Query entities.

Issues and considerations

Consider the following points when deciding how to implement this pattern:

- You can keep your duplicate entities eventually consistent with each other by using the Eventually consistent transactions pattern to maintain the primary and secondary index entities.
- Table storage is relatively cheap to use, so the cost overhead of storing duplicate data should not be a major concern. However, always evaluate the cost of your design based on your anticipated storage requirements, and only add duplicate entities to support the queries your client application will run.
- The value used for the `RowKey` must be unique for each entity. Consider using compound key values.
- Padding numeric values in the `RowKey` (for example, the employee ID 000223) enables correct sorting and filtering based on upper and lower bounds.
- You don't necessarily need to duplicate all the properties of your entity. For example, if the queries that look up the entities by using the email address in the `RowKey` never need the employee's age, these entities can have the following structure:

- Typically, it's better to store duplicate data and ensure that you can retrieve all the data you need with a single query, than to use one query to locate an entity by using the secondary index and another to look up the required data in the primary index.

When to use this pattern

Use this pattern when:

- Your client application needs to retrieve entities by using a variety of different keys.
- Your client needs to retrieve entities in different sort orders.
- You can identify each entity by using a variety of unique values.

Use this pattern when you want to avoid exceeding the partition scalability limits when you are performing entity lookups by using the different `RowKey` values.

Related patterns and guidance

The following patterns and guidance might also be relevant when implementing this pattern:

- Eventually consistent transactions pattern
- Intra-partition secondary index pattern
- Compound key pattern
- Entity group transactions
- Work with heterogeneous entity types

Eventually consistent transactions pattern

Enable eventually consistent behavior across partition boundaries or storage system boundaries by using Azure queues.

Context and problem

EGTs enable atomic transactions across multiple entities that share the same partition key. For performance and scalability reasons, you might decide to store entities that have consistency requirements in separate partitions or in a separate storage system. In such a scenario, you can't use EGTs to maintain consistency. For example, you might have a requirement to maintain eventual consistency between:

- Entities stored in two different partitions in the same table, in different tables, or in different storage accounts.
- An entity stored in Table storage and a blob stored in Blob storage.
- An entity stored in Table storage and a file in a file system.
- An entity stored in Table storage, yet indexed by using Azure Cognitive Search.

Solution

By using Azure queues, you can implement a solution that delivers eventual consistency across two or more partitions or storage systems.

To illustrate this approach, assume you have a requirement to be able to archive former employee entities. Former employee entities are rarely queried, and should be excluded from any activities that deal with current employees. To implement this requirement, you store active employees in the **Current** table and former employees in the **Archive** table. Archiving an employee requires you to delete the entity from the **Current** table, and add the entity to the **Archive** table.

But you can't use an EGT to perform these two operations. To avoid the risk that a failure causes an entity to appear in both or neither tables, the archive operation must be eventually consistent. The following sequence diagram outlines the steps in this operation.

A client initiates the archive operation by placing a message on an Azure queue (in this example, to archive employee #456). A worker role polls the queue for new messages; when it finds one, it reads the message and leaves a hidden copy on the queue. The worker role next fetches a copy of the entity from the **Current** table, inserts a copy in the **Archive** table, and then deletes the original from the **Current** table. Finally, if there were no errors from the previous steps, the worker role deletes the hidden message from the queue.

In this example, step 4 in the diagram inserts the employee into the **Archive** table. It can add the employee to a blob in Blob storage or a file in a file system.

Recover from failures

It's important that the operations in steps 4-5 in the diagram be *idempotent* in case the worker role needs to restart the archive operation. If you're using Table storage, for step 4 you should use an "insert or replace" operation; for step 5, you should use a "delete if exists" operation in the client library you're using. If you're using another storage system, you must use an appropriate idempotent operation.

If the worker role never completes step 6 in the diagram, then, after a timeout, the message reappears on the queue ready for the worker role to try to reprocess it. The worker role can check how many times a message on the queue has been read and, if necessary, flag it as a "poison" message for investigation by sending it to a separate queue. For more information about reading queue messages and checking the dequeue count, see Get messages.

Some errors from Table storage and Queue storage are transient errors, and your client application should include suitable retry logic to handle them.

Issues and considerations

Consider the following points when deciding how to implement this pattern:

- This solution doesn't provide for transaction isolation. For example, a client might read the **Current** and **Archive** tables when the worker role was between steps 4-5 in the diagram, and see an inconsistent view of the data. The data will be consistent eventually.
- You must be sure that steps 4-5 are idempotent in order to ensure eventual consistency.
- You can scale the solution by using multiple queues and worker role instances.

When to use this pattern

Use this pattern when you want to guarantee eventual consistency between entities that exist in different partitions or tables. You can extend this pattern to ensure eventual consistency for operations across Table storage and Blob storage, and other non-Azure Storage data sources, such as a database or the file system.

Related patterns and guidance

The following patterns and guidance might also be relevant when implementing this pattern:

- Entity group transactions
- Merge or replace

> **NOTE**
> If transaction isolation is important to your solution, consider redesigning your tables to enable you to use EGTs.

Index entities pattern

Maintain index entities to enable efficient searches that return lists of entities.

Context and problem

Table storage automatically indexes entities by using the `PartitionKey` and `RowKey` values. This enables a client application to retrieve an entity efficiently by using a point query. For example, using the following table structure, a client application can efficiently retrieve an individual employee entity by using the department name and the employee ID (the `PartitionKey` and `RowKey`).

Employee entity
PartitionKey (Department name) RowKey (Employee Id)
FirstName (string) LastName (string) Age (integer) EmailAddress (string)

If you also want to be able to retrieve a list of employee entities based on the value of another non-unique property, such as last name, you must use a less efficient partition scan. This scan finds matches, rather than using an index to look them up directly. This is because Table storage doesn't provide secondary indexes.

Solution

To enable lookup by last name with the preceding entity structure, you must maintain lists of employee IDs. If you want to retrieve the employee entities with a particular last name, such as Jones, you must first locate the list of employee IDs for employees with Jones as their last name, and then retrieve those employee entities. There are three main options for storing the lists of employee IDs:

- Use Blob storage.
- Create index entities in the same partition as the employee entities.
- Create index entities in a separate partition or table.

Option 1: Use Blob storage

Create a blob for every unique last name, and in each blob store a list of the `PartitionKey` (department) and `RowKey` (employee ID) values for employees who have that last name. When you add or delete an employee, ensure that the content of the relevant blob is eventually consistent with the employee entities.

Option 2: Create index entities in the same partition

Use index entities that store the following data:

Employee index entity
PartitionKey (Department name) RowKey (LastName)
EmployeeIDs (String containing a list of employee ids with same last name)

The `EmployeeIDs` property contains a list of employee IDs for employees with the last name stored in the `RowKey`.

The following steps outline the process you should follow when you're adding a new employee. In this example, we're adding an employee with ID 000152 and last name Jones in the Sales department:

1. Retrieve the index entity with a `PartitionKey` value "Sales", and the `RowKey` value "Jones". Save the ETag of this entity to use in step 2.
2. Create an entity group transaction (that is, a batch operation) that inserts the new employee entity (`PartitionKey` value "Sales" and `RowKey` value "000152"), and updates the index entity (`PartitionKey` value "Sales" and `RowKey` value "Jones"). The EGT does this by adding the new employee ID to the list in the EmployeeIDs field. For more information about EGTs, see Entity group transactions.
3. If the EGT fails because of an optimistic concurrency error (that is, someone else has modified the index entity), then you need to start over at step 1.

You can use a similar approach to deleting an employee if you're using the second option. Changing an employee's last name is slightly more complex, because you need to run an EGT that updates three entities: the employee entity, the index entity for the old last name, and the index entity for the new last name. You must retrieve each entity before making any changes, in order to retrieve the ETag values that you can then use to perform the updates by using optimistic concurrency.

The following steps outline the process you should follow when you need to look up all the employees with a particular last name in a department. In this example, we're looking up all the employees with last name Jones in the Sales department:

1. Retrieve the index entity with a `PartitionKey` value "Sales", and the `RowKey` value "Jones".
2. Parse the list of employee IDs in the `EmployeeIDs` field.
3. If you need additional information about each of these employees (such as their email addresses), retrieve each of the employee entities by using `PartitionKey` value "Sales", and `RowKey` values from the list of employees you obtained in step 2.

Option 3: Create index entities in a separate partition or table

For this option, use index entities that store the following data:

Employee index entity
PartitionKey ("indexentities") RowKey (LastName)
EmployeeIDs (String containing a list of employee ids with same last name)

The `EmployeeIDs` property contains a list of employee IDs for employees with the last name stored in the `RowKey` and `PartitionKey`.

You can't use EGTs to maintain consistency, because the index entities are in a separate partition from the employee entities. Ensure that the index entities are eventually consistent with the employee entities.

Issues and considerations

Consider the following points when deciding how to implement this pattern:

- This solution requires at least two queries to retrieve matching entities: one to query the index entities to obtain the list of `RowKey` values, and then queries to retrieve each entity in the list.
- Because an individual entity has a maximum size of 1 MB, option 2 and option 3 in the solution assume that the list of employee IDs for any particular last name is never more than 1 MB. If the list of employee IDs is likely to be more than 1 MB in size, use option 1 and store the index data in Blob storage.

- If you use option 2 (using EGTs to handle adding and deleting employees, and changing an employee's last name), you must evaluate if the volume of transactions will approach the scalability limits in a particular partition. If this is the case, you should consider an eventually consistent solution (option 1 or option 3). These use queues to handle the update requests, and enable you to store your index entities in a separate partition from the employee entities.
- Option 2 in this solution assumes that you want to look up by last name within a department. For example, you want to retrieve a list of employees with a last name Jones in the Sales department. If you want to be able to look up all the employees with a last name Jones across the whole organization, use either option 1 or option 3.
- You can implement a queue-based solution that delivers eventual consistency. For more details, see the Eventually consistent transactions pattern.

When to use this pattern

Use this pattern when you want to look up a set of entities that all share a common property value, such as all employees with the last name Jones.

Related patterns and guidance

The following patterns and guidance might also be relevant when implementing this pattern:

- Compound key pattern
- Eventually consistent transactions pattern
- Entity group transactions
- Work with heterogeneous entity types

Denormalization pattern

Combine related data together in a single entity to enable you to retrieve all the data you need with a single point query.

Context and problem

In a relational database, you typically normalize data to remove duplication that occurs when queries retrieve data from multiple tables. If you normalize your data in Azure tables, you must make multiple round trips from the client to the server to retrieve your related data. For example, with the following table structure, you need two round trips to retrieve the details for a department. One trip fetches the department entity that includes the manager's ID, and the second trip fetches the manager's details in an employee entity.

Solution

Instead of storing the data in two separate entities, denormalize the data and keep a copy of the manager's details in the department entity. For example:

With department entities stored with these properties, you can now retrieve all the details you need about a department by using a point query.

Issues and considerations

Consider the following points when deciding how to implement this pattern:

- There is some cost overhead associated with storing some data twice. The performance benefit resulting from fewer requests to Table storage typically outweighs the marginal increase in storage costs. Further, this cost is partially offset by a reduction in the number of transactions you require to fetch the details of a department.
- You must maintain the consistency of the two entities that store information about managers. You can handle the consistency issue by using EGTs to update multiple entities in a single atomic transaction. In this case, the department entity and the employee entity for the department manager are stored in the same partition.

When to use this pattern

Use this pattern when you frequently need to look up related information. This pattern reduces the number of queries your client must make to retrieve the data it requires.

Related patterns and guidance

The following patterns and guidance might also be relevant when implementing this pattern:

- Compound key pattern
- Entity group transactions
- Work with heterogeneous entity types

Compound key pattern

Use compound `RowKey` values to enable a client to look up related data with a single point query.

Context and problem

In a relational database, it's natural to use joins in queries to return related pieces of data to the client in a single query. For example, you might use the employee ID to look up a list of related entities that contain performance and review data for that employee.

Assume you are storing employee entities in Table storage by using the following structure:

Employee entity
PartitionKey (Department name) RowKey (Employee Id)
FirstName (string) LastName (string) Age (integer) EmailAddress (string)

You also need to store historical data relating to reviews and performance for each year the employee has worked for your organization, and you need to be able to access this information by year. One option is to create another table that stores entities with the following structure:

Employee review entity
PartitionKey (Employee Id) RowKey (Year)
FirstName (string) LastName (string) ManagerRating (integer) PeerRating (integer) Comments (string)

With this approach, you might decide to duplicate some information (such as first name and last name) in the new entity, to enable you to retrieve your data with a single request. However, you can't maintain strong consistency because you can't use an EGT to update the two entities atomically.

Solution

Store a new entity type in your original table by using entities with the following structure:

Notice how the `RowKey` is now a compound key, made up of the employee ID and the year of the review data. This enables you to retrieve the employee's performance and review data with a single request for a single entity.

The following example outlines how you can retrieve all the review data for a particular employee (such as employee 000123 in the Sales department):

$filter=(PartitionKey eq 'Sales') and (RowKey ge 'empid_000123') and (RowKey lt 'empid_000124')&$select=RowKey,Manager Rating,Peer Rating,Comments

Issues and considerations

Consider the following points when deciding how to implement this pattern:

- You should use a suitable separator character that makes it easy to parse the `RowKey` value: for example, `000123_2012`.
- You're also storing this entity in the same partition as other entities that contain related data for the same employee. This means you can use EGTs to maintain strong consistency.
- You should consider how frequently you'll query the data to determine whether this pattern is appropriate. For example, if you access the review data infrequently, and the main employee data often, you should keep them as separate entities.

When to use this pattern

Use this pattern when you need to store one or more related entities that you query frequently.

Related patterns and guidance

The following patterns and guidance might also be relevant when implementing this pattern:

- Entity group transactions
- Work with heterogeneous entity types
- Eventually consistent transactions pattern

Log tail pattern

Retrieve the *n* entities most recently added to a partition by using a `RowKey` value that sorts in reverse date and time order.

> **NOTE**
>
> Query results returned by the Azure Table API in Azure Cosmos DB aren't sorted by partition key or row key. Thus, while this pattern is suitable for Table storage, it isn't suitable for Azure Cosmos DB. For a detailed list of feature differences, see differences between Table API in Azure Cosmos DB and Azure Table Storage.

Context and problem

A common requirement is to be able to retrieve the most recently created entities, for example the ten most recent expense claims submitted by an employee. Table queries support a `$top` query operation to return the first *n* entities from a set. There's no equivalent query operation to return the last *n* entities in a set.

Solution

Store the entities by using a `RowKey` that naturally sorts in reverse date/time order, so the most recent entry is

always the first one in the table.

For example, to be able to retrieve the ten most recent expense claims submitted by an employee, you can use a reverse tick value derived from the current date/time. The following C# code sample shows one way to create a suitable "inverted ticks" value for a `RowKey` that sorts from the most recent to the oldest:

```
string invertedTicks = string.Format("{0:D19}", DateTime.MaxValue.Ticks - DateTime.UtcNow.Ticks);
```

You can get back to the date/time value by using the following code:

```
DateTime dt = new DateTime(DateTime.MaxValue.Ticks - Int64.Parse(invertedTicks));
```

The table query looks like this:

```
https://myaccount.table.core.windows.net/EmployeeExpense(PartitionKey='empid')?$top=10
```

Issues and considerations

Consider the following points when deciding how to implement this pattern:

- You must pad the reverse tick value with leading zeroes, to ensure the string value sorts as expected.
- You must be aware of the scalability targets at the level of a partition. Be careful to not create hot spot partitions.

When to use this pattern

Use this pattern when you need to access entities in reverse date/time order, or when you need to access the most recently added entities.

Related patterns and guidance

The following patterns and guidance might also be relevant when implementing this pattern:

- Prepend / append anti-pattern
- Retrieve entities

High volume delete pattern

Enable the deletion of a high volume of entities by storing all the entities for simultaneous deletion in their own separate table. You delete the entities by deleting the table.

Context and problem

Many applications delete old data that no longer needs to be available to a client application, or that the application has archived to another storage medium. You typically identify such data by a date. For example, you have a requirement to delete records of all sign-in requests that are more than 60 days old.

One possible design is to use the date and time of the sign-in request in the `RowKey`:

This approach avoids partition hotspots, because the application can insert and delete sign-in entities for each user in a separate partition. However, this approach can be costly and time consuming if you have a large number of entities. First, you need to perform a table scan in order to identify all the entities to delete, and then you must delete each old entity. You can reduce the number of round trips to the server required to delete the old entities by batching multiple delete requests into EGTs.

Solution

Use a separate table for each day of sign-in attempts. You can use the preceding entity design to avoid hotspots when you are inserting entities. Deleting old entities is now simply a question of deleting one table every day (a single storage operation), instead of finding and deleting hundreds and thousands of individual sign-in entities

every day.

Issues and considerations

Consider the following points when deciding how to implement this pattern:

- Does your design support other ways your application will use the data, such as looking up specific entities, linking with other data, or generating aggregate information?
- Does your design avoid hot spots when you are inserting new entities?
- Expect a delay if you want to reuse the same table name after deleting it. It's better to always use unique table names.
- Expect some rate limiting when you first use a new table, while Table storage learns the access patterns and distributes the partitions across nodes. You should consider how frequently you need to create new tables.

When to use this pattern

Use this pattern when you have a high volume of entities that you must delete at the same time.

Related patterns and guidance

The following patterns and guidance might also be relevant when implementing this pattern:

- Entity group transactions
- Modify entities

Data series pattern

Store complete data series in a single entity to minimize the number of requests you make.

Context and problem

A common scenario is for an application to store a series of data that it typically needs to retrieve all at once. For example, your application might record how many IM messages each employee sends every hour, and then use this information to plot how many messages each user sent over the preceding 24 hours. One design might be to store 24 entities for each employee:

With this design, you can easily locate and update the entity to update for each employee whenever the application needs to update the message count value. However, to retrieve the information to plot a chart of the activity for the preceding 24 hours, you must retrieve 24 entities.

Solution

Use the following design, with a separate property to store the message count for each hour:

With this design, you can use a merge operation to update the message count for an employee for a specific hour. Now, you can retrieve all the information you need to plot the chart by using a request for a single entity.

Issues and considerations

Consider the following points when deciding how to implement this pattern:

- If your complete data series doesn't fit into a single entity (an entity can have up to 252 properties), use an

alternative data store such as a blob.

- If you have multiple clients updating an entity simultaneously, use the **ETag** to implement optimistic concurrency. If you have many clients, you might experience high contention.

When to use this pattern

Use this pattern when you need to update and retrieve a data series associated with an individual entity.

Related patterns and guidance

The following patterns and guidance might also be relevant when implementing this pattern:

- Large entities pattern
- Merge or replace
- Eventually consistent transactions pattern (if you're storing the data series in a blob)

Wide entities pattern

Use multiple physical entities to store logical entities with more than 252 properties.

Context and problem

An individual entity can have no more than 252 properties (excluding the mandatory system properties), and can't store more than 1 MB of data in total. In a relational database, you would typically work around any limits on the size of a row by adding a new table, and enforcing a 1-to-1 relationship between them.

Solution

By using Table storage, you can store multiple entities to represent a single large business object with more than 252 properties. For example, if you want to store a count of the number of IM messages sent by each employee for the last 365 days, you can use the following design that uses two entities with different schemas:

If you need to make a change that requires updating both entities to keep them synchronized with each other, you can use an EGT. Otherwise, you can use a single merge operation to update the message count for a specific day. To retrieve all the data for an individual employee, you must retrieve both entities. You can do this with two efficient requests that use both a `PartitionKey` and a `RowKey` value.

Issues and considerations

Consider the following point when deciding how to implement this pattern:

- Retrieving a complete logical entity involves at least two storage transactions: one to retrieve each physical entity.

When to use this pattern

Use this pattern when you need to store entities whose size or number of properties exceeds the limits for an individual entity in Table storage.

Related patterns and guidance

The following patterns and guidance might also be relevant when implementing this pattern:

- Entity group transactions
- Merge or replace

Large entities pattern

Use Blob storage to store large property values.

Context and problem

An individual entity can't store more than 1 MB of data in total. If one or several of your properties store values that cause the total size of your entity to exceed this value, you can't store the entire entity in Table storage.

Solution

If your entity exceeds 1 MB in size because one or more properties contain a large amount of data, you can store data in Blob storage, and then store the address of the blob in a property in the entity. For example, you can store the photo of an employee in Blob storage, and store a link to the photo in the `Photo` property of your employee entity:

Issues and considerations

Consider the following points when deciding how to implement this pattern:

- To maintain eventual consistency between the entity in Table storage and the data in Blob storage, use the Eventually consistent transactions pattern to maintain your entities.
- Retrieving a complete entity involves at least two storage transactions: one to retrieve the entity and one to retrieve the blob data.

When to use this pattern

Use this pattern when you need to store entities whose size exceeds the limits for an individual entity in Table storage.

Related patterns and guidance

The following patterns and guidance might also be relevant when implementing this pattern:

- Eventually consistent transactions pattern
- Wide entities pattern

Prepend/append anti-pattern

When you have a high volume of inserts, increase scalability by spreading the inserts across multiple partitions.

Context and problem

Prepending or appending entities to your stored entities typically results in the application adding new entities to the first or last partition of a sequence of partitions. In this case, all of the inserts at any particular time are taking place in the same partition, creating a hotspot. This prevents Table storage from load-balancing inserts across multiple nodes, and possibly causes your application to hit the scalability targets for partition. For example, consider the case of an application that logs network and resource access by employees. An entity structure such as the following can result in the current hour's partition becoming a hotspot, if the volume of transactions reaches the scalability target for an individual partition:

Solution

The following alternative entity structure avoids a hotspot on any particular partition, as the application logs events:

Notice with this example how both the `PartitionKey` and `RowKey` are compound keys. The `PartitionKey` uses both the department and employee ID to distribute the logging across multiple partitions.

Issues and considerations

Consider the following points when deciding how to implement this pattern:

- Does the alternative key structure that avoids creating hot partitions on inserts efficiently support the queries your client application makes?
- Does your anticipated volume of transactions mean that you're likely to reach the scalability targets for an individual partition, and be throttled by Table storage?

When to use this pattern

Avoid the prepend/append anti-pattern when your volume of transactions is likely to result in rate limiting by Table storage when you access a hot partition.

Related patterns and guidance

The following patterns and guidance might also be relevant when implementing this pattern:

- Compound key pattern
- Log tail pattern
- Modify entities

Log data anti-pattern

Typically, you should use Blob storage instead of Table storage to store log data.

Context and problem

A common use case for log data is to retrieve a selection of log entries for a specific date/time range. For example, you want to find all the error and critical messages that your application logged between 15:04 and 15:06 on a specific date. You don't want to use the date and time of the log message to determine the partition you save log entities to. That results in a hot partition because at any particular time, all the log entities will share the same

`PartitionKey` value (see the Prepend/append anti-pattern). For example, the following entity schema for a log message results in a hot partition, because the application writes all log messages to the partition for the current date and hour:

Log message entity
PartitionKey (Date and hour of log message) RowKey (Time of log message and unique log message id (GUID)) Severity (integer) Source (string) Message (string) Department (string) ClientIP (string)

In this example, the `RowKey` includes the date and time of the log message to ensure that log messages are sorted in date/time order. The `RowKey` also includes a message ID, in case multiple log messages share the same date and time.

Another approach is to use a `PartitionKey` that ensures that the application writes messages across a range of partitions. For example, if the source of the log message provides a way to distribute messages across many partitions, you can use the following entity schema:

Log message entity
PartitionKey (Source) RowKey (Time of log message and unique log message id (GUID)) Severity (integer) Message (string) Department (string) ClientIP (string)

However, the problem with this schema is that to retrieve all the log messages for a specific time span, you must search every partition in the table.

Solution

The previous section highlighted the problem of trying to use Table storage to store log entries, and suggested two unsatisfactory designs. One solution led to a hot partition with the risk of poor performance writing log messages. The other solution resulted in poor query performance, because of the requirement to scan every partition in the table to retrieve log messages for a specific time span. Blob storage offers a better solution for this type of scenario, and this is how Azure Storage analytics stores the log data it collects.

This section outlines how Storage analytics stores log data in Blob storage, as an illustration of this approach to storing data that you typically query by range.

Storage analytics stores log messages in a delimited format in multiple blobs. The delimited format makes it easy for a client application to parse the data in the log message.

Storage analytics uses a naming convention for blobs that enables you to locate the blob (or blobs) that contain the log messages for which you are searching. For example, a blob named "queue/2014/07/31/1800/000001.log" contains log messages that relate to the queue service for the hour starting at 18:00 on July 31, 2014. The "000001" indicates that this is the first log file for this period. Storage analytics also records the timestamps of the first and last log messages stored in the file, as part of the blob's metadata. The API for Blob storage enables you locate blobs in a container based on a name prefix. To locate all the blobs that contain queue log data for the hour starting at 18:00, you can use the prefix "queue/2014/07/31/1800".

Storage analytics buffers log messages internally, and then periodically updates the appropriate blob or creates a new one with the latest batch of log entries. This reduces the number of writes it must perform to Blob storage.

If you're implementing a similar solution in your own application, consider how to manage the trade-off between reliability and cost and scalability. In other words, evaluate the effect of writing every log entry to Blob storage as it

happens, compared to buffering updates in your application and writing them to Blob storage in batches.

Issues and considerations

Consider the following points when deciding how to store log data:

- If you create a table design that avoids potential hot partitions, you might find that you can't access your log data efficiently.
- To process log data, a client often needs to load many records.
- Although log data is often structured, Blob storage might be a better solution.

Implementation considerations

This section discusses some of the considerations to bear in mind when you implement the patterns described in the previous sections. Most of this section uses examples written in C# that use the Storage Client Library (version 4.3.0 at the time of writing).

Retrieve entities

As discussed in the section Design for querying, the most efficient query is a point query. However, in some scenarios you might need to retrieve multiple entities. This section describes some common approaches to retrieving entities by using the Storage Client Library.

Run a point query by using the Storage Client Library

The easiest way to run a point query is to use the **Retrieve** table operation. As shown in the following C# code snippet, this operation retrieves an entity with a `PartitionKey` of value "Sales", and a `RowKey` of value "212":

```
TableOperation retrieveOperation = TableOperation.Retrieve<EmployeeEntity>("Sales", "212");
var retrieveResult = employeeTable.Execute(retrieveOperation);
if (retrieveResult.Result != null)
{
    EmployeeEntity employee = (EmployeeEntity)retrieveResult.Result;
    ...
}
```

Notice how this example expects the entity it retrieves to be of type `EmployeeEntity`.

Retrieve multiple entities by using LINQ

You can retrieve multiple entities by using LINQ with Storage Client Library, and specifying a query with a **where** clause. To avoid a table scan, you should always include the `PartitionKey` value in the where clause, and if possible the `RowKey` value to avoid table and partition scans. Table storage supports a limited set of comparison operators (greater than, greater than or equal, less than, less than or equal, equal, and not equal) to use in the where clause. The following C# code snippet finds all the employees whose last name starts with "B" (assuming that the `RowKey` stores the last name) in the Sales department (assuming the `PartitionKey` stores the department name):

```
TableQuery<EmployeeEntity> employeeQuery = employeeTable.CreateQuery<EmployeeEntity>();
var query = (from employee in employeeQuery
            where employee.PartitionKey == "Sales" &&
            employee.RowKey.CompareTo("B") >= 0 &&
            employee.RowKey.CompareTo("C") < 0
            select employee).AsTableQuery();
var employees = query.Execute();
```

Notice how the query specifies both a `RowKey` and a `PartitionKey` to ensure better performance.

The following code sample shows equivalent functionality by using the fluent API (for more information about fluent APIs in general, see Best practices for designing a fluent API):

```
TableQuery<EmployeeEntity> employeeQuery = new TableQuery<EmployeeEntity>().Where(
    TableQuery.CombineFilters(
    TableQuery.CombineFilters(
        TableQuery.GenerateFilterCondition(
    "PartitionKey", QueryComparisons.Equal, "Sales"),
    TableOperators.And,
    TableQuery.GenerateFilterCondition(
    "RowKey", QueryComparisons.GreaterThanOrEqual, "B")
),
TableOperators.And,
TableQuery.GenerateFilterCondition("RowKey", QueryComparisons.LessThan, "C")
    )
);
var employees = employeeTable.ExecuteQuery(employeeQuery);
```

NOTE

The sample nests multiple `CombineFilters` methods to include the three filter conditions.

Retrieve large numbers of entities from a query

An optimal query returns an individual entity based on a `PartitionKey` value and a `RowKey` value. However, in some scenarios you might have a requirement to return many entities from the same partition, or even from many partitions. You should always fully test the performance of your application in such scenarios.

A query against Table storage can return a maximum of 1,000 entities at one time, and run for a maximum of five seconds. Table storage returns a continuation token to enable the client application to request the next set of entities, if any of the following are true:

- The result set contains more than 1,000 entities.
- The query didn't complete within five seconds.
- The query crosses the partition boundary.

For more information about how continuation tokens work, see Query timeout and pagination.

If you're using the Storage Client Library, it can automatically handle continuation tokens for you as it returns entities from Table storage. For example, the following C# code sample automatically handles continuation tokens if Table storage returns them in a response:

```
string filter = TableQuery.GenerateFilterCondition(
        "PartitionKey", QueryComparisons.Equal, "Sales");
TableQuery<EmployeeEntity> employeeQuery =
        new TableQuery<EmployeeEntity>().Where(filter);

var employees = employeeTable.ExecuteQuery(employeeQuery);
foreach (var emp in employees)
{
        ...
}
```

The following C# code handles continuation tokens explicitly:

```
string filter = TableQuery.GenerateFilterCondition(
        "PartitionKey", QueryComparisons.Equal, "Sales");
TableQuery<EmployeeEntity> employeeQuery =
        new TableQuery<EmployeeEntity>().Where(filter);

TableContinuationToken continuationToken = null;

do
{
        var employees = employeeTable.ExecuteQuerySegmented(
        employeeQuery, continuationToken);
    foreach (var emp in employees)
    {
    ...
    }
    continuationToken = employees.ContinuationToken;
} while (continuationToken != null);
```

By using continuation tokens explicitly, you can control when your application retrieves the next segment of data. For example, if your client application enables users to page through the entities stored in a table, a user might decide not to page through all the entities retrieved by the query. Your application would only use a continuation token to retrieve the next segment when the user had finished paging through all the entities in the current segment. This approach has several benefits:

- You can limit the amount of data to retrieve from Table storage and that you move over the network.
- You can perform asynchronous I/O in .NET.
- You can serialize the continuation token to persistent storage, so you can continue in the event of an application crash.

> **NOTE**
> A continuation token typically returns a segment containing 1,000 entities, although it can contain fewer. This is also the case if you limit the number of entries a query returns by using **Take** to return the first n entities that match your lookup criteria. Table storage might return a segment containing fewer than n entities, along with a continuation token to enable you to retrieve the remaining entities.

The following C# code shows how to modify the number of entities returned inside a segment:

```
employeeQuery.TakeCount = 50;
```

Server-side projection

A single entity can have up to 255 properties and be up to 1 MB in size. When you query the table and retrieve entities, you might not need all the properties, and can avoid transferring data unnecessarily (to help reduce latency and cost). You can use server-side projection to transfer just the properties you need. The following example retrieves just the `Email` property (along with `PartitionKey`, `RowKey`, `Timestamp`, and `ETag`) from the entities selected by the query.

```
string filter = TableQuery.GenerateFilterCondition(
        "PartitionKey", QueryComparisons.Equal, "Sales");
List<string> columns = new List<string>() { "Email" };
TableQuery<EmployeeEntity> employeeQuery =
        new TableQuery<EmployeeEntity>().Where(filter).Select(columns);

var entities = employeeTable.ExecuteQuery(employeeQuery);
foreach (var e in entities)
{
        Console.WriteLine("RowKey: {0}, EmployeeEmail: {1}", e.RowKey, e.Email);
}
```

Notice how the `RowKey` value is available even though it isn't included in the list of properties to retrieve.

Modify entities

The Storage Client Library enables you to modify your entities stored in Table storage by inserting, deleting, and updating entities. You can use EGTs to batch multiple inserts, update, and delete operations together, to reduce the number of round trips required and improve the performance of your solution.

Exceptions thrown when the Storage Client Library runs an EGT typically include the index of the entity that caused the batch to fail. This is helpful when you are debugging code that uses EGTs.

You should also consider how your design affects how your client application handles concurrency and update operations.

Managing concurrency

By default, Table storage implements optimistic concurrency checks at the level of individual entities for insert, merge, and delete operations, although it's possible for a client to force Table storage to bypass these checks. For more information, see Managing concurrency in Microsoft Azure Storage.

Merge or replace

The `Replace` method of the `TableOperation` class always replaces the complete entity in Table storage. If you don't include a property in the request when that property exists in the stored entity, the request removes that property from the stored entity. Unless you want to remove a property explicitly from a stored entity, you must include every property in the request.

You can use the `Merge` method of the `TableOperation` class to reduce the amount of data that you send to Table storage when you want to update an entity. The `Merge` method replaces any properties in the stored entity with property values from the entity included in the request. This method leaves intact any properties in the stored entity that aren't included in the request. This is useful if you have large entities, and only need to update a small number of properties in a request.

> **NOTE**
>
> The `*Replace` and `Merge` methods fail if the entity doesn't exist. As an alternative, you can use the `InsertOrReplace` and `InsertOrMerge` methods that create a new entity if it doesn't exist.

Work with heterogeneous entity types

Table storage is a *schema-less* table store. That means that a single table can store entities of multiple types, providing great flexibility in your design. The following example illustrates a table storing both employee and department entities:

PARTITIONKEY	ROWKEY	TIMESTAMP	

			FIRSTNAME, LASTNAME, AGE, EMAIL
			FIRSTNAME, LASTNAME, AGE, EMAIL
			DEPARTMENTNAME, EMPLOYEECOUNT
			FIRSTNAME, LASTNAME, AGE, EMAIL

Each entity must still have `PartitionKey`, `RowKey`, and `Timestamp` values, but can have any set of properties. Furthermore, there's nothing to indicate the type of an entity unless you choose to store that information somewhere. There are two options for identifying the entity type:

- Prepend the entity type to the `RowKey` (or possibly the `PartitionKey`). For example, `EMPLOYEE_000123` or `DEPARTMENT_SALES` as `RowKey` values.
- Use a separate property to record the entity type, as shown in the following table.

PARTITIONKEY	ROWKEY	TIMESTAMP	
			ENTITYTYPE, FIRSTNAME, LASTNAME, AGE, EMAIL Employee

			<table><tr><th>ENTITYTYPE</th><th>FIRSTNAME</th><th>LASTNAME</th><th>AGE</th><th>EMAIL</th></tr><tr><td>Employee</td><td></td><td></td><td></td><td></td></tr></table>
			<table><tr><th>ENTITY TYPE</th><th>DEPARTMENT NAME</th><th>EMPLOYEECOUNT</th></tr><tr><td>Department</td><td></td><td></td></tr></table>
			<table><tr><th>ENTITYTYPE</th><th>FIRSTNAME</th><th>LASTNAME</th><th>AGE</th><th>EMAIL</th></tr><tr><td>Employee</td><td></td><td></td><td></td><td></td></tr></table>

The first option, prepending the entity type to the `RowKey`, is useful if there is a possibility that two entities of different types might have the same key value. It also groups entities of the same type together in the partition.

The techniques discussed in this section are especially relevant to the discussion aboutInheritance relationships.

> **NOTE**
> Consider including a version number in the entity type value, to enable client applications to evolve POCO objects and work with different versions.

The remainder of this section describes some of the features in the Storage Client Library that facilitate working with multiple entity types in the same table.

Retrieve heterogeneous entity types

If you're using the Storage Client Library, you have three options for working with multiple entity types.

If you know the type of the entity stored with specific `RowKey` and `PartitionKey` values, then you can specify the entity type when you retrieve the entity. You saw this in the previous two examples that retrieve entities of type `EmployeeEntity`: Run a point query by using the Storage Client Library and Retrieve multiple entities by using LINQ.

The second option is to use the `DynamicTableEntity` type (a property bag), instead of a concrete POCO entity type. This option might also improve performance, because there's no need to serialize and deserialize the entity to .NET types. The following C# code potentially retrieves multiple entities of different types from the table, but returns all entities as `DynamicTableEntity` instances. It then uses the `EntityType` property to determine the type of each entity:

```
string filter = TableQuery.CombineFilters(
    TableQuery.GenerateFilterCondition("PartitionKey",
    QueryComparisons.Equal, "Sales"),
    TableOperators.And,
    TableQuery.CombineFilters(
    TableQuery.GenerateFilterCondition("RowKey",
                    QueryComparisons.GreaterThanOrEqual, "B"),
        TableOperators.And,
        TableQuery.GenerateFilterCondition("RowKey",
        QueryComparisons.LessThan, "F")
    )
);
TableQuery<DynamicTableEntity> entityQuery =
    new TableQuery<DynamicTableEntity>().Where(filter);
var employees = employeeTable.ExecuteQuery(entityQuery);

IEnumerable<DynamicTableEntity> entities = employeeTable.ExecuteQuery(entityQuery);
foreach (var e in entities)
{
EntityProperty entityTypeProperty;
if (e.Properties.TryGetValue("EntityType", out entityTypeProperty))
{
    if (entityTypeProperty.StringValue == "Employee")
    {
        // Use entityTypeProperty, RowKey, PartitionKey, Etag, and Timestamp
        }
    }
}
```

To retrieve other properties, you must use the `TryGetValue` method on the `Properties` property of the `DynamicTableEntity` class.

A third option is to combine using the `DynamicTableEntity` type and an `EntityResolver` instance. This enables you to resolve to multiple POCO types in the same query. In this example, the `EntityResolver` delegate is using the `EntityType` property to distinguish between the two types of entity that the query returns. The `Resolve` method uses the `resolver` delegate to resolve `DynamicTableEntity` instances to `TableEntity` instances.

```
EntityResolver<TableEntity> resolver = (pk, rk, ts, props, etag) =>
{

        TableEntity resolvedEntity = null;
        if (props["EntityType"].StringValue == "Department")
        {
        resolvedEntity = new DepartmentEntity();
        }
        else if (props["EntityType"].StringValue == "Employee")
        {
        resolvedEntity = new EmployeeEntity();
        }
        else throw new ArgumentException("Unrecognized entity", "props");

        resolvedEntity.PartitionKey = pk;
        resolvedEntity.RowKey = rk;
        resolvedEntity.Timestamp = ts;
        resolvedEntity.ETag = etag;
        resolvedEntity.ReadEntity(props, null);
        return resolvedEntity;
};

string filter = TableQuery.GenerateFilterCondition(
        "PartitionKey", QueryComparisons.Equal, "Sales");
TableQuery<DynamicTableEntity> entityQuery =
        new TableQuery<DynamicTableEntity>().Where(filter);

var entities = employeeTable.ExecuteQuery(entityQuery, resolver);
foreach (var e in entities)
{
        if (e is DepartmentEntity)
        {
    ...
        }
        if (e is EmployeeEntity)
        {
    ...
        }
}
```

Modify heterogeneous entity types

You don't need to know the type of an entity to delete it, and you always know the type of an entity when you insert it. However, you can use the `DynamicTableEntity` type to update an entity without knowing its type, and without using a POCO entity class. The following code sample retrieves a single entity, and checks that the `EmployeeCount` property exists before updating it.

```
TableResult result =
        employeeTable.Execute(TableOperation.Retrieve(partitionKey, rowKey));
DynamicTableEntity department = (DynamicTableEntity)result.Result;

EntityProperty countProperty;

if (!department.Properties.TryGetValue("EmployeeCount", out countProperty))
{
        throw new
        InvalidOperationException("Invalid entity, EmployeeCount property not found.");
}
countProperty.Int32Value += 1;
employeeTable.Execute(TableOperation.Merge(department));
```

Control access with shared access signatures

You can use shared access signature (SAS) tokens to enable client applications to modify (and query) table entities

directly, without the need to authenticate directly with Table storage. Typically, there are three main benefits to using SAS in your application:

- You don't need to distribute your storage account key to an insecure platform (such as a mobile device) in order to allow that device to access and modify entities in Table storage.
- You can offload some of the work that web and worker roles perform in managing your entities. You can offload to client devices such as end-user computers and mobile devices.
- You can assign a constrained and time-limited set of permissions to a client (such as allowing read-only access to specific resources).

For more information about using SAS tokens with Table storage, see Using shared access signatures (SAS).

However, you must still generate the SAS tokens that grant a client application to the entities in Table storage. Do this in an environment that has secure access to your storage account keys. Typically, you use a web or worker role to generate the SAS tokens and deliver them to the client applications that need access to your entities. Because there is still an overhead involved in generating and delivering SAS tokens to clients, you should consider how best to reduce this overhead, especially in high-volume scenarios.

It's possible to generate a SAS token that grants access to a subset of the entities in a table. By default, you create a SAS token for an entire table. But it's also possible to specify that the SAS token grant access to either a range of `PartitionKey` values, or a range of `PartitionKey` and `RowKey` values. You might choose to generate SAS tokens for individual users of your system, such that each user's SAS token only allows them access to their own entities in Table storage.

Asynchronous and parallel operations

Provided you are spreading your requests across multiple partitions, you can improve throughput and client responsiveness by using asynchronous or parallel queries. For example, you might have two or more worker role instances accessing your tables in parallel. You can have individual worker roles responsible for particular sets of partitions, or simply have multiple worker role instances, each able to access all the partitions in a table.

Within a client instance, you can improve throughput by running storage operations asynchronously. The Storage Client Library makes it easy to write asynchronous queries and modifications. For example, you might start with the synchronous method that retrieves all the entities in a partition, as shown in the following C# code:

```
private static void ManyEntitiesQuery(CloudTable employeeTable, string department)
{
        string filter = TableQuery.GenerateFilterCondition(
        "PartitionKey", QueryComparisons.Equal, department);
        TableQuery<EmployeeEntity> employeeQuery =
        new TableQuery<EmployeeEntity>().Where(filter);

        TableContinuationToken continuationToken = null;

        do
        {
        var employees = employeeTable.ExecuteQuerySegmented(
                employeeQuery, continuationToken);
        foreach (var emp in employees)
    {
        ...
    }
        continuationToken = employees.ContinuationToken;
        } while (continuationToken != null);
}
```

You can easily modify this code so that the query runs asynchronously, as follows:

```
private static async Task ManyEntitiesQueryAsync(CloudTable employeeTable, string department)
{
        string filter = TableQuery.GenerateFilterCondition(
        "PartitionKey", QueryComparisons.Equal, department);
        TableQuery<EmployeeEntity> employeeQuery =
        new TableQuery<EmployeeEntity>().Where(filter);
        TableContinuationToken continuationToken = null;

        do
        {
        var employees = await employeeTable.ExecuteQuerySegmentedAsync(
                employeeQuery, continuationToken);
        foreach (var emp in employees)
        {
            ...
        }
        continuationToken = employees.ContinuationToken;
            } while (continuationToken != null);
}
```

In this asynchronous example, you can see the following changes from the synchronous version:

- The method signature now includes the `async` modifier, and returns a `Task` instance.
- Instead of calling the `ExecuteSegmented` method to retrieve results, the method now calls the `ExecuteSegmentedAsync` method. The method uses the `await` modifier to retrieve results asynchronously.

The client application can call this method multiple times, with different values for the `department` parameter. Each query runs on a separate thread.

There is no asynchronous version of the `Execute` method in the `TableQuery` class, because the `IEnumerable` interface doesn't support asynchronous enumeration.

You can also insert, update, and delete entities asynchronously. The following C# example shows a simple, synchronous method to insert or replace an employee entity:

```
private static void SimpleEmployeeUpsert(CloudTable employeeTable,
        EmployeeEntity employee)
{
        TableResult result = employeeTable
        .Execute(TableOperation.InsertOrReplace(employee));
        Console.WriteLine("HTTP Status: {0}", result.HttpStatusCode);
}
```

You can easily modify this code so that the update runs asynchronously, as follows:

```
private static async Task SimpleEmployeeUpsertAsync(CloudTable employeeTable,
        EmployeeEntity employee)
{
        TableResult result = await employeeTable
        .ExecuteAsync(TableOperation.InsertOrReplace(employee));
        Console.WriteLine("HTTP Status: {0}", result.HttpStatusCode);
}
```

In this asynchronous example, you can see the following changes from the synchronous version:

- The method signature now includes the `async` modifier, and returns a `Task` instance.
- Instead of calling the `Execute` method to update the entity, the method now calls the `ExecuteAsync` method. The method uses the `await` modifier to retrieve results asynchronously.

The client application can call multiple asynchronous methods like this one, and each method invocation runs on a separate thread.

Manage Azure Cosmos DB Table API resources using Azure Resource Manager templates

12/4/2020 • 4 minutes to read • Edit Online

APPLIES TO: Table API

In this article, you learn how to use Azure Resource Manager templates to help deploy and manage your Azure Cosmos DB accounts, databases, and containers.

This article has examples for Table API accounts only, to find examples for other API type accounts see: use Azure Resource Manager templates with Azure Cosmos DB's API for Cassandra, Gremlin, MongoDB, SQL articles.

IMPORTANT

- Account names are limited to 44 characters, all lowercase.
- To change the throughput values, redeploy the template with updated RU/s.
- When you add or remove locations to an Azure Cosmos account, you can't simultaneously modify other properties. These operations must be done separately.

To create any of the Azure Cosmos DB resources below, copy the following example template into a new json file. You can optionally create a parameters json file to use when deploying multiple instances of the same resource with different names and values. There are many ways to deploy Azure Resource Manager templates including, Azure portal, Azure CLI, Azure PowerShell and GitHub.

TIP

To enable shared throughput when using Table API, enable account-level throughput in the Azure portal.

Azure Cosmos account for Table with autoscale throughput

This template will create an Azure Cosmos account for Table API with one table with autoscale throughput. This template is also available for one-click deploy from Azure Quickstart Templates Gallery.

Deploy to Azure

```
{
    "$schema": "https://schema.management.azure.com/schemas/2019-04-01/deploymentTemplate.json#",
    "contentVersion": "1.0.0.0",
    "parameters": {
       "accountName": {
          "type": "string",
          "defaultValue": "[concat('table-', uniqueString(resourceGroup().id))]",
          "metadata": {
             "description": "Cosmos DB account name"
          }
       },
       "location": {
          "type": "string",
          "defaultValue": "[resourceGroup().location]",
          "metadata": {
             "description": "Location for the Cosmos DB account."
          }
```

```
        },
        "primaryRegion":{
           "type":"string",
           "metadata": {
              "description": "The primary replica region for the Cosmos DB account."
           }
        },
        "secondaryRegion":{
           "type":"string",
           "metadata": {
             "description": "The secondary replica region for the Cosmos DB account."
          }
        },
        "defaultConsistencyLevel": {
           "type": "string",
           "defaultValue": "Session",
           "allowedValues": [
                "Eventual",
                "ConsistentPrefix",
                "Session",
                "BoundedStaleness",
                "Strong"
             ],
           "metadata": {
              "description": "The default consistency level of the Cosmos DB account."
           }
        },
        "maxStalenessPrefix": {
           "type": "int",
           "defaultValue": 100000,
           "minValue": 10,
           "maxValue": 2147483647,
           "metadata": {
              "description": "Max stale requests. Required for BoundedStaleness. Valid ranges, Single Region:
10 to 1000000. Multi Region: 100000 to 1000000."
           }
        },
        "maxIntervalInSeconds": {
           "type": "int",
           "defaultValue": 300,
           "minValue": 5,
           "maxValue": 86400,
           "metadata": {
              "description": "Max lag time (seconds). Required for BoundedStaleness. Valid ranges, Single
Region: 5 to 84600. Multi Region: 300 to 86400."
           }
        },
        "automaticFailover": {
           "type": "bool",
           "defaultValue": true,
           "allowedValues": [
                true,
                false
             ],
           "metadata": {
              "description": "Enable automatic failover for regions. Ignored when Multi-Master is enabled"
           }
        },
        "tableName": {
           "type": "string",
           "metadata": {
              "description": "The name for the table"
           }
        },
         "autoscaleMaxThroughput": {
             "type": "int",
             "defaultValue": 4000,
             "minValue": 4000,
             "maxValue": 1000000,
```

```
            "metadata": {
                "description": "Maximum autoscale throughput for the table"
            }
        }
    },
    "variables": {
       "accountName": "[toLower(parameters('accountName'))]",
       "consistencyPolicy": {
          "Eventual": {
             "defaultConsistencyLevel": "Eventual"
          },
          "ConsistentPrefix": {
             "defaultConsistencyLevel": "ConsistentPrefix"
          },
          "Session": {
             "defaultConsistencyLevel": "Session"
          },
          "BoundedStaleness": {
             "defaultConsistencyLevel": "BoundedStaleness",
             "maxStalenessPrefix": "[parameters('maxStalenessPrefix')]",
             "maxIntervalInSeconds": "[parameters('maxIntervalInSeconds')]"
          },
          "Strong": {
             "defaultConsistencyLevel": "Strong"
          }
       },
       "locations": [
          {
             "locationName": "[parameters('primaryRegion')]",
             "failoverPriority": 0,
             "isZoneRedundant": false
          },
          {
             "locationName": "[parameters('secondaryRegion')]",
             "failoverPriority": 1,
             "isZoneRedundant": false
          }
       ]
    },
    "resources": [
       {
          "type": "Microsoft.DocumentDB/databaseAccounts",
          "name": "[variables('accountName')]",
          "apiVersion": "2020-04-01",
          "location": "[parameters('location')]",
          "kind": "GlobalDocumentDB",
          "properties": {
            "capabilities": [
               {
                  "name": "EnableTable"
               }
            ],
            "consistencyPolicy": "[variables('consistencyPolicy')[parameters('defaultConsistencyLevel')]]",
            "locations": "[variables('locations')]",
            "databaseAccountOfferType": "Standard",
            "enableAutomaticFailover": "[parameters('automaticFailover')]"
            }
       },
       {
          "type": "Microsoft.DocumentDB/databaseAccounts/tables",
          "name": "[concat(variables('accountName'), '/', parameters('tableName'))]",
          "apiVersion": "2020-04-01",
          "dependsOn": [
             "[resourceId('Microsoft.DocumentDB/databaseAccounts/', variables('accountName'))]"
            ],
          "properties":{
             "resource":{
                "id": "[parameters('tableName')]"
             },
```

```
            },
            "options": {
                "autoscaleSettings": {
                   "maxThroughput": "[parameters('autoscaleMaxThroughput')]"
                }
            }
          }
        }
    ]
 }
```

Azure Cosmos account for Table with standard provisioned throughput

This template will create an Azure Cosmos account for Table API with one table with standard throughput. This template is also available for one-click deploy from Azure Quickstart Templates Gallery.

Deploy to Azure

```
{
    "$schema": "https://schema.management.azure.com/schemas/2019-04-01/deploymentTemplate.json#",
    "contentVersion": "1.0.0.0",
    "parameters": {
        "accountName": {
            "type": "string",
            "defaultValue": "[concat('table-', uniqueString(resourceGroup().id))]",
            "metadata": {
                "description": "Cosmos DB account name"
            }
        },
        "location": {
            "type": "string",
            "defaultValue": "[resourceGroup().location]",
            "metadata": {
                "description": "Location for the Cosmos DB account."
            }
        },
        "primaryRegion":{
            "type":"string",
            "metadata": {
                "description": "The primary replica region for the Cosmos DB account."
            }
        },
        "secondaryRegion":{
            "type":"string",
            "metadata": {
              "description": "The secondary replica region for the Cosmos DB account."
          }
        },
        "defaultConsistencyLevel": {
            "type": "string",
            "defaultValue": "Session",
            "allowedValues": [ "Eventual", "ConsistentPrefix", "Session", "BoundedStaleness", "Strong" ],
            "metadata": {
                "description": "The default consistency level of the Cosmos DB account."
            }
        },
        "maxStalenessPrefix": {
            "type": "int",
            "defaultValue": 100000,
            "minValue": 10,
            "maxValue": 2147483647,
            "metadata": {
                "description": "Max stale requests. Required for BoundedStaleness. Valid ranges, Single
Region: 10 to 1000000. Multi Region: 100000 to 1000000."
            }
        }
```

```
        },
        "maxIntervalInSeconds": {
            "type": "int",
            "defaultValue": 300,
            "minValue": 5,
            "maxValue": 86400,
            "metadata": {
                "description": "Max lag time (seconds). Required for BoundedStaleness. Valid ranges, Single
Region: 5 to 84600. Multi Region: 300 to 86400."
            }
        },
        "automaticFailover": {
            "type": "bool",
            "defaultValue": true,
            "allowedValues": [ true, false ],
            "metadata": {
                "description": "Enable automatic failover for regions. Ignored when Multi-Master is enabled"
            }
        },
        "tableName": {
            "type": "string",
            "metadata": {
                "description": "The name for the table"
            }
        },
        "throughput": {
            "type": "int",
            "defaultValue": 400,
            "minValue": 400,
            "maxValue": 1000000,
            "metadata": {
                "description": "The throughput for the table"
            }
        }
    },
    "variables": {
        "accountName": "[toLower(parameters('accountName'))]",
        "consistencyPolicy": {
            "Eventual": {
                "defaultConsistencyLevel": "Eventual"
            },
            "ConsistentPrefix": {
                "defaultConsistencyLevel": "ConsistentPrefix"
            },
            "Session": {
                "defaultConsistencyLevel": "Session"
            },
            "BoundedStaleness": {
                "defaultConsistencyLevel": "BoundedStaleness",
                "maxStalenessPrefix": "[parameters('maxStalenessPrefix')]",
                "maxIntervalInSeconds": "[parameters('maxIntervalInSeconds')]"
            },
            "Strong": {
                "defaultConsistencyLevel": "Strong"
            }
        },
        "locations":
        [
            {
                "locationName": "[parameters('primaryRegion')]",
                "failoverPriority": 0,
                "isZoneRedundant": false
            },
            {
                "locationName": "[parameters('secondaryRegion')]",
                "failoverPriority": 1,
                "isZoneRedundant": false
            }
        ]
```

```
    },
    "resources":
    [
        {
            "type": "Microsoft.DocumentDB/databaseAccounts",
            "name": "[variables('accountName')]",
            "apiVersion": "2020-03-01",
            "location": "[parameters('location')]",
            "kind": "GlobalDocumentDB",
            "properties": {
                "capabilities": [{ "name": "EnableTable" }],
                "consistencyPolicy": "[variables('consistencyPolicy')
[parameters('defaultConsistencyLevel')]]",
                "locations": "[variables('locations')]",
                "databaseAccountOfferType": "Standard",
                "enableAutomaticFailover": "[parameters('automaticFailover')]"
            }
        },
        {
            "type": "Microsoft.DocumentDB/databaseAccounts/tables",
            "name": "[concat(variables('accountName'), '/', parameters('tableName'))]",
            "apiVersion": "2020-03-01",
            "dependsOn": [ "[resourceId('Microsoft.DocumentDB/databaseAccounts/', variables('accountName'))]"
],
            "properties":{
                "resource":{
                    "id": "[parameters('tableName')]"
                },
                "options": { "throughput": "[parameters('throughput')]" }
            }
        }
    ]
}
```

Next steps

Here are some additional resources:

- Azure Resource Manager documentation
- Azure Cosmos DB resource provider schema
- Azure Cosmos DB Quickstart templates
- Troubleshoot common Azure Resource Manager deployment errors

Azure PowerShell samples for Azure Cosmos DB Table API

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: Table API

The following table includes links to commonly used Azure PowerShell scripts for Azure Cosmos DB. Use the links on the right to navigate to API specific samples. Common samples are the same across all APIs. Reference pages for all Azure Cosmos DB PowerShell cmdlets are available in the Azure PowerShell Reference. Please check for updates to `Az.CosmosDB` regularly. You can also fork these PowerShell samples for Cosmos DB from our GitHub repository, Cosmos DB PowerShell Samples on GitHub.

Common Samples

TASK	DESCRIPTION
Update an account	Update a Cosmos DB account's default consistency level.
Update an account's regions	Update a Cosmos DB account's regions.
Change failover priority or trigger failover	Change the regional failover priority of an Azure Cosmos account or trigger a manual failover.
Account keys or connection strings	Get primary and secondary keys, connection strings or regenerate an account key of an Azure Cosmos DB account.
Create a Cosmos Account with IP Firewall	Create an Azure Cosmos DB account with IP Firewall enabled.

Table API Samples

TASK	DESCRIPTION
Create an account and table	Creates an Azure Cosmos account and table.
Create an account and table with autoscale	Creates an Azure Cosmos account and table autoscale.
List or get tables	List or get tables.
Throughput operations	Throughput operations for a table including get, update and migrate between autoscale and standard throughput.
Lock resources from deletion	Prevent resources from being deleted with resource locks.

Azure CLI samples for Azure Cosmos DB Table API

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: Table API

The following table includes links to sample Azure CLI scripts for Azure Cosmos DB. Use the links on the right to navigate to API specific samples. Common samples are the same across all APIs. Reference pages for all Azure Cosmos DB CLI commands are available in the Azure CLI Reference. Azure Cosmos DB CLI script samples can also be found in the Azure Cosmos DB CLI GitHub Repository.

These samples require Azure CLI version 2.12.1 or later. Run `az --version` to find the version. If you need to install or upgrade, see Install Azure CLI

Common Samples

These samples apply to all Azure Cosmos DB APIs

TASK	DESCRIPTION
Add or failover regions	Add a region, change failover priority, trigger a manual failover.
Account keys and connection strings	List account keys, read-only keys, regenerate keys and list connection strings.
Secure with IP firewall	Create a Cosmos account with IP firewall configured.
Secure new account with service endpoints	Create a Cosmos account and secure with service-endpoints.
Secure existing account with service endpoints	Update a Cosmos account to secure with service-endpoints when the subnet is eventually configured.

Table API Samples

TASK	DESCRIPTION
Create an Azure Cosmos account and table	Creates an Azure Cosmos DB account and table for Table API.
Create an Azure Cosmos account and table with autoscale	Creates an Azure Cosmos DB account and table with autoscale for Table API.
Throughput operations	Read, update and migrate between autoscale and standard throughput on a table.
Lock resources from deletion	Prevent resources from being deleted with resource locks.

Azure Cosmos DB Table .NET API: Download and release notes

11/2/2020 • 4 minutes to read • Edit Online

APPLIES TO: Table API

SDK download	NuGet
Quickstart	Azure Cosmos DB: Build an app with .NET and the Table API
Tutorial	Azure Cosmos DB: Develop with the Table API in .NET
Current supported framework	Microsoft .NET Framework 4.5.1

> **IMPORTANT**
> The .NET Framework SDK Microsoft.Azure.CosmosDB.Table is in maintenance mode and it will be deprecated soon. Please upgrade to the new .NET Standard library Microsoft.Azure.Cosmos.Table to continue to get the latest features supported by the Table API.

> If you created a Table API account during the preview, please create a new Table API account to work with the generally available Table API SDKs.

Release notes

2.1.2

- Bug fixes

2.1.0

- Bug fixes

2.0.0

- Added Multi-region write support
- Fixed NuGet package dependencies on Microsoft.Azure.DocumentDB, Microsoft.OData.Core, Microsoft.OData.Edm, Microsoft.Spatial

1.1.3

- Fixed NuGet package dependencies on Microsoft.Azure.Storage.Common and Microsoft.Azure.DocumentDB.
- Bug fixes on table serialization when JsonConvert.DefaultSettings are configured.

1.1.1

- Added validation for malformed ETAGs in Direct Mode.
- Fixed LINQ query bug in Gateway Mode.
- Synchronous APIs now run on the thread pool with SynchronizationContext.

1.1.0

- Add TableQueryMaxItemCount, TableQueryEnableScan, TableQueryMaxDegreeOfParallelism, and TableQueryContinuationTokenLimitInKb to TableRequestOptions
- Bug Fixes

1.0.0

- General availability release

0.9.0-preview

- Initial preview release

Release and Retirement dates

Microsoft provides notification at least **12 months** in advance of retiring an SDK in order to smooth the transition to a newer/supported version.

The `Microsoft.Azure.CosmosDB.Table` library is currently available for .NET Framework only, and is in maintenance mode and will be deprecated soon. New features and functionalities and optimizations are only added to the .NET Standard library Microsoft.Azure.Cosmos.Table, as such it is recommended that you upgrade to Microsoft.Azure.Cosmos.Table.

The WindowsAzure.Storage-PremiumTable preview package has been deprecated. The WindowsAzure.Storage-PremiumTable SDK will be retired on November 15, 2018, at which time requests to the retired SDK will not be permitted.

VERSION	RELEASE DATE	RETIREMENT DATE
2.1.2	September 16, 2019	
2.1.0	January 22, 2019	April 01, 2020
2.0.0	September 26, 2018	March 01, 2020
1.1.3	July 17, 2018	December 01, 2019
1.1.1	March 26, 2018	December 01, 2019
1.1.0	February 21, 2018	December 01, 2019
1.0.0	November 15, 2017	November 15, 2019
0.9.0-preview	November 11, 2017	November 11, 2019

Troubleshooting

If you get the error

```
Unable to resolve dependency 'Microsoft.Azure.Storage.Common'. Source(s) used: 'nuget.org',
'CliFallbackFolder', 'Microsoft Visual Studio Offline Packages', 'Microsoft Azure Service Fabric SDK'`
```

when attempting to use the Microsoft.Azure.CosmosDB.Table NuGet package, you have two options to fix the issue:

- Use Package Manage Console to install the Microsoft.Azure.CosmosDB.Table package and its dependencies. To do this, type the following in the Package Manager Console for your solution.

```
Install-Package Microsoft.Azure.CosmosDB.Table -IncludePrerelease
```

- Using your preferred NuGet package management tool, install the Microsoft.Azure.Storage.Common NuGet package before installing Microsoft.Azure.CosmosDB.Table.

FAQ

How will I be notified of the retiring SDK?

Microsoft will provide 12 month's advance notice before the end of support of the retiring SDK to facilitate a smooth transition to a supported SDK. We'll notify you through various communication channels: the Azure portal, Azure updates, and direct communication to assigned service administrators.

Can I author applications by using a to-be-retired Azure Cosmos DB SDK during the 12-month period?

Yes, you'll be able to author, deploy, and modify applications by using the to-be-retired Azure Cosmos DB SDK during the 12-month notice period. We recommend that you migrate to a newer supported version of the Azure Cosmos DB SDK during the 12-month notice period, as appropriate.

After the retirement date, what happens to applications that use the unsupported Azure Cosmos DB SDK?

After the retirement date, Azure Cosmos DB will no longer make bug fixes, add new features, or provide support to the retired SDK versions. If you prefer not to upgrade, requests sent from the retired versions of the SDK will continue to be served by the Azure Cosmos DB service.

Which SDK versions will have the latest features and updates?

New features and updates will be added only to the latest minor version of the latest supported major SDK version. We recommend that you always use the latest version to take advantage of new features, performance improvements, and bug fixes. If you're using an old, non-retired version of the SDK, your requests to Azure Cosmos DB will still function, but you won't have access to any new capabilities.

What should I do if I can't update my application before a cutoff date?

We recommend that you upgrade to the latest SDK as early as possible. After an SDK is tagged for retirement, you'll have 12 months to update your application. If you're not able to update by the retirement date, requests sent from the retired versions of the SDK will continue to be served by Azure Cosmos DB, so your running applications will continue to function. But Azure Cosmos DB will no longer make bug fixes, add new features, or provide support to the retired SDK versions.

If you have a support plan and require technical support, contact us by filing a support ticket.

See also

To learn more about the Azure Cosmos DB Table API, see Introduction to Azure Cosmos DB Table API.

Azure Cosmos DB Table .NET Standard API: Download and release notes

11/2/2020 • 4 minutes to read • Edit Online

APPLIES TO: Table API

SDK download	NuGet
Sample	Cosmos DB Table API .NET Sample
Quickstart	Quickstart
Tutorial	Tutorial
Current supported framework	Microsoft .NET Standard 2.0
Report Issue	Report Issue

Release notes for 2.0.0 series

2.0.0 series takes the dependency on Microsoft.Azure.Cosmos, with performance improvements and namespace consolidation to Cosmos DB endpoint.

2.0.0-preview

- initial preview of 2.0.0 Table SDK that takes the dependency on Microsoft.Azure.Cosmos, with performance improvements and namespace consolidation to Cosmos DB endpoint. The public API remains the same.

Release notes for 1.0.0 series

1.0.0 series takes the dependency on Microsoft.Azure.DocumentDB.Core.

1.0.8

- Add support to set TTL property if it's cosmosdb endpoint
- Honor retry policy upon timeout and task cancelled exception
- Fix intermittent task cancelled exception seen in asp .net applications
- Fix azure table storage retrieve from secondary endpoint only location mode
- Update `Microsoft.Azure.DocumentDB.Core` dependency version to 2.11.2 which fixes intermittent null reference exception
- Update `Odata.Core` dependency version to 7.6.4 which fixes compatibility conflict with azure shell

1.0.7

- Performance improvement by setting Table SDK default trace level to SourceLevels.Off, which can be opted in via app.config

1.0.5

- Introduce new config under TableClientConfiguration to use Rest Executor to communicate with Cosmos DB Table API

1.0.5-preview

- Bug fixes

1.0.4

- Bug fixes
- Provide HttpClientTimeout option for RestExecutorConfiguration.

1.0.4-preview

- Bug fixes
- Provide HttpClientTimeout option for RestExecutorConfiguration.

1.0.1

- Bug fixes

1.0.0

- General availability release

0.11.0-preview

- Changes were made to how CloudTableClient can be configured. It now takes an a TableClientConfiguration object during construction. TableClientConfiguration provides different properties to configure the client behavior depending on whether the target endpoint is Cosmos DB Table API or Azure Storage Table API.
- Added support to TableQuery to return results in sorted order on a custom column. This feature is only supported on Cosmos DB Table endpoints.
- Added support to expose RequestCharges on various result types. This feature is only supported on Cosmos DB Table endpoints.

0.10.1-preview

- Add support for SAS token, operations of TablePermissions, ServiceProperties, and ServiceStats against Azure Storage Table endpoints.

  > **NOTE**
  > Some functionalities in previous Azure Storage Table SDKs are not yet supported, such as client-side encryption.

0.10.0-preview

- Add support for core CRUD, batch, and query operations against Azure Storage Table endpoints.

  > **NOTE**
  > Some functionalities in previous Azure Storage Table SDKs are not yet supported, such as client-side encryption.

0.9.1-preview

- Azure Cosmos DB Table .NET Standard SDK is a cross-platform .NET library that provides efficient access to the Table data model on Cosmos DB. This initial release supports the full set of Table and Entity CRUD + Query functionalities with similar APIs as the Cosmos DB Table SDK For .NET Framework.

  > **NOTE**
  > Azure Storage Table endpoints are not yet supported in the 0.9.1-preview version.

Release and Retirement dates

Microsoft provides notification at least **12 months** in advance of retiring an SDK in order to smooth the transition

to a newer/supported version.

This cross-platform .NET Standard library Microsoft.Azure.Cosmos.Table will replace the .NET Framework library Microsoft.Azure.CosmosDB.Table.

2.0.0 series

VERSION	RELEASE DATE	RETIREMENT DATE
2.0.0-preview	August 22, 2019	---

1.0.0 series

VERSION	RELEASE DATE	RETIREMENT DATE
1.0.5	September 13, 2019	---
1.0.5-preview	August 20, 2019	---
1.0.4	August 12, 2019	---
1.0.4-preview	July 26, 2019	---
1.0.2-preview	May 2, 2019	---
1.0.1	April 19, 2019	---
1.0.0	March 13, 2019	---
0.11.0-preview	March 5, 2019	---
0.10.1-preview	January 22, 2019	---
0.10.0-preview	December 18, 2018	---
0.9.1-preview	October 18, 2018	---

FAQ

How will I be notified of the retiring SDK?

Microsoft will provide 12 month's advance notice before the end of support of the retiring SDK to facilitate a smooth transition to a supported SDK. We'll notify you through various communication channels: the Azure portal, Azure updates, and direct communication to assigned service administrators.

Can I author applications by using a to-be-retired Azure Cosmos DB SDK during the 12-month period?

Yes, you'll be able to author, deploy, and modify applications by using the to-be-retired Azure Cosmos DB SDK during the 12-month notice period. We recommend that you migrate to a newer supported version of the Azure Cosmos DB SDK during the 12-month notice period, as appropriate.

After the retirement date, what happens to applications that use the unsupported Azure Cosmos DB SDK?

After the retirement date, Azure Cosmos DB will no longer make bug fixes, add new features, or provide support

to the retired SDK versions. If you prefer not to upgrade, requests sent from the retired versions of the SDK will continue to be served by the Azure Cosmos DB service.

Which SDK versions will have the latest features and updates?

New features and updates will be added only to the latest minor version of the latest supported major SDK version. We recommend that you always use the latest version to take advantage of new features, performance improvements, and bug fixes. If you're using an old, non-retired version of the SDK, your requests to Azure Cosmos DB will still function, but you won't have access to any new capabilities.

What should I do if I can't update my application before a cutoff date?

We recommend that you upgrade to the latest SDK as early as possible. After an SDK is tagged for retirement, you'll have 12 months to update your application. If you're not able to update by the retirement date, requests sent from the retired versions of the SDK will continue to be served by Azure Cosmos DB, so your running applications will continue to function. But Azure Cosmos DB will no longer make bug fixes, add new features, or provide support to the retired SDK versions.

If you have a support plan and require technical support, contact us by filing a support ticket.

See also

To learn more about the Azure Cosmos DB Table API, see Introduction to Azure Cosmos DB Table API.

Azure Cosmos DB Table API for Java: Release notes and resources

11/2/2020 • 2 minutes to read • Edit Online

APPLIES TO: Table API

SDK download	Download Options
API documentation	Java API reference documentation
Contribute to SDK	GitHub

> **IMPORTANT**
> If you created a Table API account during the preview, please create a new Table API account to work with the generally available Table API SDKs.

Release notes

1.0.0

- General availability release

Release and retirement dates

Microsoft will provide notification at least **12 months** in advance of retiring an SDK in order to smooth the transition to a newer/supported version.

New features and functionality and optimizations are only added to the current SDK, as such it is recommended that you always upgrade to the latest SDK version as early as possible.

VERSION	RELEASE DATE	RETIREMENT DATE
1.0.0	November 15, 2017	---

FAQ

How will I be notified of the retiring SDK?

Microsoft will provide 12 month's advance notice before the end of support of the retiring SDK to facilitate a smooth transition to a supported SDK. We'll notify you through various communication channels: the Azure portal, Azure updates, and direct communication to assigned service administrators.

Can I author applications by using a to-be-retired Azure Cosmos DB SDK during the 12-month period?

Yes, you'll be able to author, deploy, and modify applications by using the to-be-retired Azure Cosmos DB SDK during the 12-month notice period. We recommend that you migrate to a newer supported version of the Azure Cosmos DB SDK during the 12-month notice period, as appropriate.

After the retirement date, what happens to applications that use the unsupported Azure Cosmos DB SDK?

After the retirement date, Azure Cosmos DB will no longer make bug fixes, add new features, or provide support to the retired SDK versions. If you prefer not to upgrade, requests sent from the retired versions of the SDK will continue to be served by the Azure Cosmos DB service.

Which SDK versions will have the latest features and updates?

New features and updates will be added only to the latest minor version of the latest supported major SDK version. We recommend that you always use the latest version to take advantage of new features, performance improvements, and bug fixes. If you're using an old, non-retired version of the SDK, your requests to Azure Cosmos DB will still function, but you won't have access to any new capabilities.

What should I do if I can't update my application before a cutoff date?

We recommend that you upgrade to the latest SDK as early as possible. After an SDK is tagged for retirement, you'll have 12 months to update your application. If you're not able to update by the retirement date, requests sent from the retired versions of the SDK will continue to be served by Azure Cosmos DB, so your running applications will continue to function. But Azure Cosmos DB will no longer make bug fixes, add new features, or provide support to the retired SDK versions.

If you have a support plan and require technical support, contact us by filing a support ticket.

See also

To learn more about Cosmos DB, see Microsoft Azure Cosmos DB service page.

Azure Cosmos DB Table API for Node.js: Release notes and resources

11/2/2020 • 2 minutes to read • Edit Online

APPLIES TO: Table API

SDK download	NPM
API documentation	Node.js API reference documentation
Contribute to SDK	GitHub

> **IMPORTANT**
> If you created a Table API account during the preview, please create a new Table API account to work with the generally available Table API SDKs.

Release notes

1.0.0

- General availability release

Release and retirement dates

Microsoft will provide notification at least **12 months** in advance of retiring an SDK in order to smooth the transition to a newer/supported version.

New features and functionality and optimizations are only added to the current SDK, as such it is recommended that you always upgrade to the latest SDK version as early as possible.

VERSION	RELEASE DATE	RETIREMENT DATE
1.0.0	November 15, 2017	---

FAQ

How will I be notified of the retiring SDK?

Microsoft will provide 12 month's advance notice before the end of support of the retiring SDK to facilitate a smooth transition to a supported SDK. We'll notify you through various communication channels: the Azure portal, Azure updates, and direct communication to assigned service administrators.

Can I author applications by using a to-be-retired Azure Cosmos DB SDK during the 12-month period?

Yes, you'll be able to author, deploy, and modify applications by using the to-be-retired Azure Cosmos DB SDK during the 12-month notice period. We recommend that you migrate to a newer supported version of the Azure Cosmos DB SDK during the 12-month notice period, as appropriate.

After the retirement date, what happens to applications that use the unsupported Azure Cosmos DB SDK?

After the retirement date, Azure Cosmos DB will no longer make bug fixes, add new features, or provide support to the retired SDK versions. If you prefer not to upgrade, requests sent from the retired versions of the SDK will continue to be served by the Azure Cosmos DB service.

Which SDK versions will have the latest features and updates?

New features and updates will be added only to the latest minor version of the latest supported major SDK version. We recommend that you always use the latest version to take advantage of new features, performance improvements, and bug fixes. If you're using an old, non-retired version of the SDK, your requests to Azure Cosmos DB will still function, but you won't have access to any new capabilities.

What should I do if I can't update my application before a cutoff date?

We recommend that you upgrade to the latest SDK as early as possible. After an SDK is tagged for retirement, you'll have 12 months to update your application. If you're not able to update by the retirement date, requests sent from the retired versions of the SDK will continue to be served by Azure Cosmos DB, so your running applications will continue to function. But Azure Cosmos DB will no longer make bug fixes, add new features, or provide support to the retired SDK versions.

If you have a support plan and require technical support, contact us by filing a support ticket.

See also

To learn more about Cosmos DB, see Microsoft Azure Cosmos DB service page.

Azure Cosmos DB Table API SDK for Python: Release notes and resources

11/2/2020 • 2 minutes to read • Edit Online

APPLIES TO: Table API

SDK download	PyPI
API documentation	Python API reference documentation
SDK installation instructions	Python SDK installation instructions
Contribute to SDK	GitHub
Current supported platform	Python 2.7 or Python 3.3, 3.4, 3.5, or 3.6

IMPORTANT

If you created a Table API account during the preview, please create a new Table API account to work with the generally available Table API SDKs.

Release notes

1.0.0

- General availability release

0.37.1

- Pre-release SDK

Release and retirement dates

Microsoft will provide notification at least **12 months** in advance of retiring an SDK in order to smooth the transition to a newer/supported version.

New features and functionality and optimizations are only added to the current SDK, as such it is recommended that you always upgrade to the latest SDK version as early as possible.

VERSION	RELEASE DATE	RETIREMENT DATE
1.0.0	November 15, 2017	---
0.37.1	October 05, 2017	---

FAQ

How will I be notified of the retiring SDK?

Microsoft will provide 12 month's advance notice before the end of support of the retiring SDK to facilitate a smooth transition to a supported SDK. We'll notify you through various communication channels: the Azure portal, Azure updates, and direct communication to assigned service administrators.

Can I author applications by using a to-be-retired Azure Cosmos DB SDK during the 12-month period?

Yes, you'll be able to author, deploy, and modify applications by using the to-be-retired Azure Cosmos DB SDK during the 12-month notice period. We recommend that you migrate to a newer supported version of the Azure Cosmos DB SDK during the 12-month notice period, as appropriate.

After the retirement date, what happens to applications that use the unsupported Azure Cosmos DB SDK?

After the retirement date, Azure Cosmos DB will no longer make bug fixes, add new features, or provide support to the retired SDK versions. If you prefer not to upgrade, requests sent from the retired versions of the SDK will continue to be served by the Azure Cosmos DB service.

Which SDK versions will have the latest features and updates?

New features and updates will be added only to the latest minor version of the latest supported major SDK version. We recommend that you always use the latest version to take advantage of new features, performance improvements, and bug fixes. If you're using an old, non-retired version of the SDK, your requests to Azure Cosmos DB will still function, but you won't have access to any new capabilities.

What should I do if I can't update my application before a cutoff date?

We recommend that you upgrade to the latest SDK as early as possible. After an SDK is tagged for retirement, you'll have 12 months to update your application. If you're not able to update by the retirement date, requests sent from the retired versions of the SDK will continue to be served by Azure Cosmos DB, so your running applications will continue to function. But Azure Cosmos DB will no longer make bug fixes, add new features, or provide support to the retired SDK versions.

If you have a support plan and require technical support, contact us by filing a support ticket.

See also

To learn more about Cosmos DB, see Microsoft Azure Cosmos DB service page.

Install and use the Azure Cosmos DB Emulator for local development and testing

12/4/2020 • 16 minutes to read • Edit Online

APPLIES TO: SQL API Cassandra API Gremlin API Table API Azure Cosmos DB API for MongoDB

The Azure Cosmos DB Emulator provides a local environment that emulates the Azure Cosmos DB service for development purposes. Using the Azure Cosmos DB Emulator, you can develop and test your application locally, without creating an Azure subscription or incurring any costs. When you're satisfied with how your application is working in the Azure Cosmos DB Emulator, you can switch to using an Azure Cosmos account in the cloud. This article describes how to install and use the emulator on Windows, Linux, macOS, and Windows docker environments.

Download the emulator

To get started, download and install the latest version of Azure Cosmos DB Emulator on your local computer. The emulator release notes article lists all the available versions and the feature updates that were made in each release.

Download the Azure Cosmos DB Emulator

You can develop applications using Azure Cosmos DB Emulator with the SQL, Cassandra, MongoDB, Gremlin, and Table API accounts. Currently the data explorer in the emulator fully supports viewing SQL data only; the data created using MongoDB, Gremlin/Graph and Cassandra client applications it is not viewable at this time. To learn more, see how to connect to the emulator endpoint from different APIs.

How does the emulator work?

The Azure Cosmos DB Emulator provides a high-fidelity emulation of the Azure Cosmos DB service. It supports equivalent functionality as the Azure Cosmos DB, which includes creating data, querying data, provisioning and scaling containers, and executing stored procedures and triggers. You can develop and test applications using the Azure Cosmos DB Emulator, and deploy them to Azure at global scale by updating the Azure Cosmos DB connection endpoint.

While emulation of the Azure Cosmos DB service is faithful, the emulator's implementation is different than the service. For example, the emulator uses standard OS components such as the local file system for persistence, and the HTTPS protocol stack for connectivity. Functionality that relies on the Azure infrastructure like global replication, single-digit millisecond latency for reads/writes, and tunable consistency levels are not applicable when you use the emulator.

You can migrate data between the Azure Cosmos DB Emulator and the Azure Cosmos DB service by using the Azure Cosmos DB Data Migration Tool.

Differences between the emulator and the cloud service

Because the Azure Cosmos DB Emulator provides an emulated environment that runs on the local developer workstation, there are some differences in functionality between the emulator and an Azure Cosmos account in the cloud:

- Currently the **Data Explorer** pane in the emulator fully supports SQL API clients only. The **Data**

Explorer view and operations for Azure Cosmos DB APIs such as MongoDB, Table, Graph, and Cassandra APIs are not fully supported.

- The emulator supports only a single fixed account and a well-known primary key. You can't regenerate key when using the Azure Cosmos DB Emulator, however you can change the default key by using the command-line option.
- With the emulator, you can create an Azure Cosmos account in provisioned throughput mode only; currently it doesn't support serverless mode.
- The emulator is not a scalable service and it doesn't support a large number of containers. When using the Azure Cosmos DB Emulator, by default, you can create up to 25 fixed size containers at 400 RU/s (only supported using Azure Cosmos DB SDKs), or 5 unlimited containers. For more information on how to change this value, see Set the PartitionCount value article.
- The emulator does not offer different Azure Cosmos DB consistency levels like the cloud service does.
- The emulator does not offer multi-region replication.
- Because the copy of your Azure Cosmos DB Emulator might not always be up to date with the most recent changes in the Azure Cosmos DB service, you should always refer to the Azure Cosmos DB capacity planner to accurately estimate the throughput (RUs) needs of your application.
- The emulator supports a maximum ID property size of 254 characters.

Install the emulator

Before you install the emulator, make sure you have the following hardware and software requirements:

- Software requirements:
  - Currently Windows Server 2016, 2019 or Windows 10 host OS are supported. The host OS with Active Directory enabled is currently not supported.
  - 64-bit operating system
- Minimum hardware requirements:
  - 2-GB RAM
  - 10-GB available hard disk space
- To install, configure, and run the Azure Cosmos DB Emulator, you must have administrative privileges on the computer. The emulator will add a certificate and also set the firewall rules in order to run its services. Therefore admin rights are necessary for the emulator to be able to execute such operations.

To get started, download and install the latest version of Azure Cosmos DB Emulator on your local computer. If you run into any issues when installing the emulator, see the emulator troubleshooting article to debug.

Depending upon your system requirements, you can run the emulator on Windows, Docker for Windows, Linux, or macOS as described in next sections of this article.

Check for emulator updates

Each version of emulator comes with a set of feature updates or bug fixes. To see the available versions, read the emulator release notes article.

After installation, if you have used the default settings, the data corresponding to the emulator is saved at %LOCALAPPDATA%\CosmosDBEmulator location. You can configure a different location by using the optional data path settings; that is the `/DataPath=PREFERRED_LOCATION` as the command-line parameter. The data created in one version of the Azure Cosmos DB Emulator is not guaranteed to be accessible when using a different version. If you need to persist your data for the long term, it is recommended that you store that data in an Azure Cosmos account, instead of the Azure Cosmos DB Emulator.

Use the emulator on Windows

The Azure Cosmos DB Emulator is installed at `C:\Program Files\Azure Cosmos DB Emulator` location by default. To start the Azure Cosmos DB Emulator on Windows, select the **Start** button or press the Windows key. Begin typing **Azure Cosmos DB Emulator**, and select the emulator from the list of applications.

When the emulator has started, you'll see an icon in the Windows taskbar notification area. It automatically opens the Azure Cosmos data explorer in your browser at this URL `https://localhost:8081/_explorer/index.html` URL.

You can also start and stop the emulator from the command-line or PowerShell commands. For more information, see the command-line tool reference article.

The Azure Cosmos DB Emulator by default runs on the local machine ("localhost") listening on port 8081. The address appears as `https://localhost:8081/_explorer/index.html`. If you close the explorer and would like to reopen it later, you can either open the URL in your browser or launch it from the Azure Cosmos DB Emulator in the Windows Tray Icon as shown below.

Use the emulator on Docker for Windows

You can run the Azure Cosmos DB Emulator on the Windows Docker container. See the Docker Hub for the docker pull command and GitHub for the `Dockerfile` and more information. Currently the emulator

does not work on Docker for Oracle Linux. Use the following instructions to run the emulator on Docker for Windows:

1. After you have Docker for Windows installed, switch to Windows containers by right-clicking the Docker icon on the toolbar and selecting **Switch to Windows containers**.

2. Next, pull the emulator image from Docker Hub by running the following command from your favorite shell.

```
docker pull mcr.microsoft.com/cosmosdb/windows/azure-cosmos-emulator
```

3. To start the image, run the following commands depending on the command line or the PowerShell environment:

   - Command line
   - PowerShell

```
md %LOCALAPPDATA%\CosmosDBEmulator\bind-mount

docker run --name azure-cosmosdb-emulator --memory 2GB --mount
"type=bind,source=%LOCALAPPDATA%\CosmosDBEmulator\bind-
mount,destination=C:\CosmosDB.Emulator\bind-mount" --interactive --tty -p 8081:8081 -p
8900:8900 -p 8901:8901 -p 8902:8902 -p 10250:10250 -p 10251:10251 -p 10252:10252 -p
10253:10253 -p 10254:10254 -p 10255:10255 -p 10256:10256 -p 10350:10350
mcr.microsoft.com/cosmosdb/windows/azure-cosmos-emulator
```

   Windows based Docker images might not be generally compatible with every Windows host OS. For instance, the default Azure Cosmos DB Emulator image is only compatible with Windows 10 and Windows Server 2016. If you need an image that is compatible with Windows Server 2019, run the following command instead:

```
docker run --name azure-cosmosdb-emulator --memory 2GB --mount
"type=bind,source=%hostDirectory%,destination=C:\CosmosDB.Emulator\bind-mount" --interactive -
-tty -p 8081:8081 -p 8900:8900 -p 8901:8901 -p 8902:8902 -p 10250:10250 -p 10251:10251 -p
10252:10252 -p 10253:10253 -p 10254:10254 -p 10255:10255 -p 10256:10256 -p 10350:10350
mcr.microsoft.com/cosmosdb/winsrv2019/azure-cosmos-emulator:latest
```

   > **NOTE**
   > When executing the `docker run` command, if you see a port conflict error (that is if the specified port is already in use), pass a custom port by altering the port numbers. For example, you can change the "-p 8081:8081" parameter to "-p 443:8081"

4. Now use the emulator endpoint and primary key from the response and import the TLS/SSL certificate into your host. To import the TLS/SSL certificate, run the following steps from an admin command prompt:

   - Command line
   - PowerShell

```
cd  %LOCALAPPDATA%\CosmosDBEmulator\bind-mount
powershell .\importcert.ps1
```

5. If you close the interactive shell after the emulator has started, it will shut down the emulator's container. To reopen the data explorer, navigate to the following URL in your browser. The emulator endpoint is provided in the response message shown above.

   ```
   https://<emulator endpoint provided in response>/_explorer/index.html
   ```

If you have a .NET client application running on a Linux docker container and if you are running Azure Cosmos DB Emulator on a host machine, use the instructions in the next section to import the certificate into the Linux docker container.

Regenerate the emulator certificates when running on a Docker container

When running the emulator in a Docker container, the certificates associated with the emulator are regenerated every time you stop and restart the respective container. Because of that you have to re-import the certificates after each container start. To work around this limitation, you can use a Docker compose file to bind the Docker container to a particular IP address and a container image.

For example, you can use the following configuration within the Docker compose file, make sure to format it per your requirement:

```
version: '2.4' # Do not upgrade to 3.x yet, unless you plan to use swarm/docker stack:
https://github.com/docker/compose/issues/4513

networks:
  default:
    external: false
    ipam:
      driver: default
      config:
        - subnet: "172.16.238.0/24"

services:

  # First create a directory that will hold the emulator traces and certificate to be imported
  # set hostDirectory=C:\emulator\bind-mount
  # mkdir %hostDirectory%

  cosmosdb:
    container_name: "azurecosmosemulator"
    hostname: "azurecosmosemulator"
    image: 'mcr.microsoft.com/cosmosdb/windows/azure-cosmos-emulator'
    platform: windows
    tty: true
    mem_limit: 3GB
    ports:
        - '8081:8081'
        - '8900:8900'
        - '8901:8901'
        - '8902:8902'
        - '10250:10250'
        - '10251:10251'
        - '10252:10252'
        - '10253:10253'
        - '10254:10254'
        - '10255:10255'
        - '10256:10256'
        - '10350:10350'
    networks:
      default:
        ipv4_address: 172.16.238.246
    volumes:
        - '${hostDirectory}:C:\CosmosDB.Emulator\bind-mount'
```

Use the emulator on Linux or macOS

Currently the Azure Cosmos DB Emulator can only be run on Windows. If you are using Linux or macOS, you can run the emulator in a Windows virtual machine hosted in a hypervisor such as Parallels or VirtualBox.

> **NOTE**
> Every time you restart the Windows virtual machine that is hosted in a hypervisor, you have to reimport the certificate because the IP address of the virtual machine changes. Importing the certificate isn't required in case you have configured the virtual machine to preserve the IP address.

Use the following steps to use the emulator on Linux or macOS environments:

1. Run the following command from the Windows virtual machine and make a note of the IPv4 address:

   ```
   ipconfig.exe
   ```

2. Within your application, change the endpoint URL to use the IPv4 address returned by `ipconfig.exe` instead of `localhost`.

3. From the Windows VM, launch the Azure Cosmos DB Emulator from the command line using the following options. For details on the parameters supported by the command line, see the emulator command-line tool reference:

   ```
   Microsoft.Azure.Cosmos.Emulator.exe /AllowNetworkAccess
   /Key=C2y6yDjf5/R+ob0N8A7Cgv30VRDJIWEHLM+4QDU5DE2nQ9nDuVTqobD4b8mGGyPMbIZnqyMsEcaGQy67XIw/Jw==
   ```

4. Finally, you need to resolve the certificate trust process between the application running on the Linux or Mac environment and the emulator. You can use one of the following two options to resolve the certificate:

   a. Import the emulator TLS/SSL certificate into the Linux or Mac environment or
   b. Disable the TLS/SSL validation in the application

Option 1: Import the emulator TLS/SSL certificate

The following sections show how to import the emulator TLS/SSL certificate into Linux and macOS environments.

Linux environment

If you are working on Linux, .NET relays on OpenSSL to do the validation:

1. Export the certificate in PFX format. The PFX option is available when choosing to export the private key.

2. Copy that PFX file into your Linux environment.

3. Convert the PFX file into a CRT file

   ```
   openssl pkcs12 -in YourPFX.pfx -clcerts -nokeys -out YourCTR.crt
   ```

4. Copy the CRT file to the folder that contains custom certificates in your Linux distribution. Commonly on Debian distributions, it is located on `/usr/local/share/ca-certificates/`.

```
cp YourCTR.crt /usr/local/share/ca-certificates/
```

5. Update the TLS/SSL certificates, which will update the `/etc/ssl/certs/` folder.

```
update-ca-certificates
```

macOS environment

Use the following steps if you are working on Mac:

1. Export the certificate in PFX format. The PFX option is available when choosing to export the private key.
2. Copy that PFX file into your Mac environment.
3. Open the *Keychain Access* application and import the PFX file.
4. Open the list of Certificates and identify the one with the name `localhost`.
5. Open the context menu for that particular item, select *Get Item* and under *Trust* > *When using this certificate* option, select *Always Trust*.

Option 2: Disable the SSL validation in the application

Disabling SSL validation is only recommended for development purposes and should not be done when running in a production environment. The following examples show how to disable SSL validation for .NET and Node.js applications.

- .NET Standard 2.1+
- .NET Standard 2.0
- Node.js

For any application running in a framework compatible with .NET Standard 2.1 or later, we can leverage the `CosmosClientOptions.HttpClientFactory`:

```
CosmosClientOptions cosmosClientOptions = new CosmosClientOptions()
{
    HttpClientFactory = () =>
    {
        HttpMessageHandler httpMessageHandler = new HttpClientHandler()
        {
            ServerCertificateCustomValidationCallback =
HttpClientHandler.DangerousAcceptAnyServerCertificateValidator
        };

        return new HttpClient(httpMessageHandler);
    },
    ConnectionMode = ConnectionMode.Gateway
};


CosmosClient client = new CosmosClient(endpoint, authKey, cosmosClientOptions);
```

Enable access to emulator on a local network

If you have multiple machines using a single network, and if you set up the emulator on one machine and want to access it from other machine. In such case, you need to enable access to the emulator on a local network.

You can run the emulator on a local network. To enable network access, specify the `/AllowNetworkAccess` option at the command-line, which also requires that you specify `/Key=key_string` or `/KeyFile=file_name`. You can use `/GenKeyFile=file_name` to generate a file with a random key upfront. Then you can pass that to `/KeyFile=file_name` or `/Key=contents_of_file`.

To enable network access for the first time, the user should shut down the emulator and delete the emulator's data directory *%LOCALAPPDATA%\CosmosDBEmulator*.

Authenticate connections when using emulator

As with Azure Cosmos DB in the cloud, every request that you make against the Azure Cosmos DB Emulator must be authenticated. The Azure Cosmos DB Emulator supports only secure communication via TLS. The Azure Cosmos DB Emulator supports a single fixed account and a well-known authentication key for primary key authentication. This account and key are the only credentials permitted for use with the Azure Cosmos DB Emulator. They are:

```
Account name: localhost:<port>
Account key: C2y6yDjf5/R+ob0N8A7Cgv30VRDJIWEHLM+4QDU5DE2nQ9nDuVTqobD4b8mGGyPMbIZnqyMsEcaGQy67XIw/Jw==
```

> **NOTE**
> The primary key supported by the Azure Cosmos DB Emulator is intended for use only with the emulator. You cannot use your production Azure Cosmos DB account and key with the Azure Cosmos DB Emulator.

> **NOTE**
> If you have started the emulator with the /Key option, then use the generated key instead of the default key `C2y6yDjf5/R+ob0N8A7Cgv30VRDJIWEHLM+4QDU5DE2nQ9nDuVTqobD4b8mGGyPMbIZnqyMsEcaGQy67XIw/Jw==`. For more information about /Key option, see Command-line tool reference.

Connect to different APIs with the emulator

SQL API

Once you have the Azure Cosmos DB Emulator running on your desktop, you can use any supported Azure Cosmos DB SDK or the Azure Cosmos DB REST API to interact with the emulator. The Azure Cosmos DB Emulator also includes a built-in data explorer that lets you create containers for SQL API or Azure Cosmos DB for Mongo DB API. By using the data explorer, you can view and edit items without writing any code.

```
// Connect to the Azure Cosmos DB Emulator running locally
CosmosClient client = new CosmosClient(
   "https://localhost:8081",
    "C2y6yDjf5/R+ob0N8A7Cgv30VRDJIWEHLM+4QDU5DE2nQ9nDuVTqobD4b8mGGyPMbIZnqyMsEcaGQy67XIw/Jw==");
```

Azure Cosmos DB's API for MongoDB

Once you have the Azure Cosmos DB Emulator running on your desktop, you can use the Azure Cosmos DB's API for MongoDB to interact with the emulator. Start the emulator from command prompt as an administrator with "/EnableMongoDbEndpoint". Then use the following connection string to connect to the MongoDB API account:

```
mongodb://localhost:C2y6yDjf5/R+ob0N8A7Cgv30VRDJIWEHLM+4QDU5DE2nQ9nDuVTqobD4b8mGGyPMbIZnqyMsEcaGQy67X
Iw/Jw==@localhost:10255/admin?ssl=true
```

Table API

Once you have the Azure Cosmos DB Emulator running on your desktop, you can use the Azure Cosmos DB Table API SDK to interact with the emulator. Start the emulator from command prompt as an administrator with "/EnableTableEndpoint". Next run the following code to connect to the table API account:

```
using Microsoft.WindowsAzure.Storage;
using Microsoft.WindowsAzure.Storage.Table;
using CloudTable = Microsoft.WindowsAzure.Storage.Table.CloudTable;
using CloudTableClient = Microsoft.WindowsAzure.Storage.Table.CloudTableClient;

string connectionString =
"DefaultEndpointsProtocol=http;AccountName=localhost;AccountKey=C2y6yDjf5/R+ob0N8A7Cgv30VRDJIWEHLM+4Q
DU5DE2nQ9nDuVTqobD4b8mGGyPMbIZnqyMsEcaGQy67XIw/Jw==;TableEndpoint=http://localhost:8902/;";

CloudStorageAccount account = CloudStorageAccount.Parse(connectionString);
CloudTableClient tableClient = account.CreateCloudTableClient();
CloudTable table = tableClient.GetTableReference("testtable");
table.CreateIfNotExists();
table.Execute(TableOperation.Insert(new DynamicTableEntity("partitionKey", "rowKey")));
```

Cassandra API

Start emulator from an administrator command prompt with "/EnableCassandraEndpoint". Alternatively you can also set the environment variable `AZURE_COSMOS_EMULATOR_CASSANDRA_ENDPOINT=true`.

1. Install Python 2.7
2. Install Cassandra CLI/CQLSH
3. Run the following commands in a regular command prompt window:

```
set Path=c:\Python27;%Path%
cd /d C:\sdk\apache-cassandra-3.11.3\bin
set SSL_VERSION=TLSv1_2
set SSL_VALIDATE=false
cqlsh localhost 10350 -u localhost -p
C2y6yDjf5/R+ob0N8A7Cgv30VRDJIWEHLM+4QDU5DE2nQ9nDuVTqobD4b8mGGyPMbIZnqyMsEcaGQy67XIw/Jw== --ssl
```

4. In the CQLSH shell, run the following commands to connect to the Cassandra endpoint:

```
CREATE KEYSPACE MyKeySpace WITH replication = {'class':'MyClass', 'replication_factor': 1};
DESCRIBE keyspaces;
USE mykeyspace;
CREATE table table1(my_id int PRIMARY KEY, my_name text, my_desc text);
INSERT into table1 (my_id, my_name, my_desc) values( 1, 'name1', 'description 1');
SELECT * from table1;
EXIT
```

Gremlin API

Start emulator from an administrator command promptwith "/EnableGremlinEndpoint". Alternatively you can also set the environment variable `AZURE_COSMOS_EMULATOR_GREMLIN_ENDPOINT=true`

1. Install apache-tinkerpop-gremlin-console-3.3.4.

2. From the emulator's data explorer create a database "db1" and a collection "coll1"; for the partition key, choose "/name"

3. Run the following commands in a regular command prompt window:

```
cd /d C:\sdk\apache-tinkerpop-gremlin-console-3.3.4-bin\apache-tinkerpop-gremlin-console-3.3.4

copy /y conf\remote.yaml conf\remote-localcompute.yaml
notepad.exe conf\remote-localcompute.yaml
  hosts: [localhost]
  port: 8901
  username: /dbs/db1/colls/coll1
  password:
C2y6yDjf5/R+ob0N8A7Cgv30VRDJIWEHLM+4QDU5DE2nQ9nDuVTqobD4b8mGGyPMbIZnqyMsEcaGQy67XIw/Jw==
  connectionPool: {
  enableSsl: false}
  serializer: { className:
org.apache.tinkerpop.gremlin.driver.ser.GraphSONMessageSerializerV1d0,
  config: { serializeResultToString: true  }}

bin\gremlin.bat
```

4. In the Gremlin shell, run the following commands to connect to the Gremlin endpoint:

```
:remote connect tinkerpop.server conf/remote-localcompute.yaml
:remote console
:> g.V()
:> g.addV('person1').property(id, '1').property('name', 'somename1')
:> g.addV('person2').property(id, '2').property('name', 'somename2')
:> g.V()
```

Uninstall the local emulator

Use the following steps to uninstall the emulator:

1. Exit all the open instances of the local emulator by right-clicking the **Azure Cosmos DB Emulator** icon on the system tray, and then select **Exit**. It may take a minute for all instances to exit.
2. In the Windows search box, type **Apps & features** and select **Apps & features (System settings)** result.
3. In the list of apps, scroll to the **Azure Cosmos DB Emulator**, select it, click **Uninstall**, then confirm and select **Uninstall** again.

Next steps

In this article, you've learned how to use the local emulator for free local development. You can now proceed to the next articles:

- Export the Azure Cosmos DB Emulator certificates for use with Java, Python, and Node.js apps
- Use command line parameters and PowerShell commands to control the emulator
- Debug issues with the emulator

Export the Azure Cosmos DB Emulator certificates for use with Java, Python, and Node.js apps

12/4/2020 • 4 minutes to read • Edit Online

APPLIES TO: SQL API Cassandra API Gremlin API Table API Azure Cosmos DB API for MongoDB

The Azure Cosmos DB Emulator provides a local environment that emulates the Azure Cosmos DB service for development purposes. Azure Cosmos Emulator supports only secure communication through TLS connections.

Certificates in the Azure Cosmos DB local emulator are generated the first time you run the emulator. There are two certificates. One of them is used to connect to the local emulator and the other is used to manage default encryption of the emulator data within the emulator. The certificate you want to export is the connection certificate with the friendly name "DocumentDBEmulatorCertificate".

When you use the emulator to develop apps in different languages such as Java, Python, or Node.js, you need to export the emulator certificate and import it into the required certificate store.

The .NET language and runtime uses the Windows Certificate Store to securely connect to the Azure Cosmos DB local emulator when the application is run on a Windows OS host. Other languages have their own method of managing and using certificates. For example, Java uses its own certificate store, Python uses socket wrappers, and Node.js uses tlsSocket.

This article demonstrates how to export the TLS/SSL certificates for use in different languages and runtime environments that do not integrate with the Windows Certificate Store. You can read more about the emulator in Use the Azure Cosmos DB Emulator for development and testing.

Export the Azure Cosmos DB TLS/SSL certificate

You need to export the emulator certificate to successfully use the emulator endpoint from languages and runtime environments that do not integrate with the Windows Certificate Store. You can export the certificate using the Windows Certificate Manager. Use the following step-by-step instructions to export the "DocumentDBEmulatorCertificate" certificate as a BASE-64 encoded X.509 (.cer) file:

1. Start the Windows Certificate manager by running certlm.msc and navigate to the Personal->Certificates folder and open the certificate with the friendly name **DocumentDbEmulatorCertificate**.

2. Click on **Details** then **OK**.

3. Click **Copy to File...**.

4. Click **Next**.

5. Click **No, do not export private key**, then click **Next**.

6. Click on **Base-64 encoded X.509 (.CER)** and then **Next**.

7. Give the certificate a name. In this case **documentdbemulatorcert** and then click **Next**.

8. Click **Finish**.

Use the certificate with Java apps

When running Java applications or MongoDB applications that uses a Java based client, it is easier to install the certificate into the Java default certificate store than passing the `-Djavax.net.ssl.trustStore=<keystore> -Djavax.net.ssl.trustStorePassword="<password>"` flags. For example, the included Java Demo application (`https://localhost:8081/_explorer/index.html`) depends on the default certificate store.

Follow the instructions in the Adding a Certificate to the Java Certificates Store to import the X.509 certificate into the default Java certificate store. Keep in mind you will be working in the *%JAVA_HOME%* directory when running keytool. After the certificate is imported into the certificate store, clients for SQL and Azure Cosmos DB's API for MongoDB will be able to connect to the Azure Cosmos DB Emulator.

Alternatively you can run the following bash script to import the certificate:

```
#!/bin/bash

# If emulator was started with /AllowNetworkAccess, replace the below with the actual IP address of it:
EMULATOR_HOST=localhost
EMULATOR_PORT=8081
EMULATOR_CERT_PATH=/tmp/cosmos_emulator.cert
openssl s_client -connect ${EMULATOR_HOST}:${EMULATOR_PORT} </dev/null | sed -ne '/-BEGIN CERTIFICATE-/,/-END
CERTIFICATE-/p' > $EMULATOR_CERT_PATH
# delete the cert if already exists
sudo $JAVA_HOME/bin/keytool -cacerts -delete -alias cosmos_emulator
# import the cert
sudo $JAVA_HOME/bin/keytool -cacerts -importcert -alias cosmos_emulator -file $EMULATOR_CERT_PATH
```

Once the "CosmosDBEmulatorCertificate" TLS/SSL certificate is installed, your application should be able to connect and use the local Azure Cosmos DB Emulator. If you have any issues, you can follow the Debugging SSL/TLS connections article. In most cases, the certificate may not be installed into the *%JAVA_HOME%/jre/lib/security/cacerts* store. For example, if you have multiple installed versions of Java your application may be using a different cacerts store than the one you updated.

Use the certificate with Python apps

When connecting to the emulator from Python apps, TLS verification is disabled. By default the Python SDK(version 2.0.0 or higher) for the SQL API will not try to use the TLS/SSL certificate when connecting to the local emulator. If however you want to use TLS validation, you can follow the examples in the Python socket wrappers documentation.

How to use the certificate in Node.js

When connecting to the emulator from Node.js SDKs, TLS verification is disabled. By default the Node.js SDK(version 1.10.1 or higher) for the SQL API will not try to use the TLS/SSL certificate when connecting to the local emulator. If however you want to use TLS validation, you can follow the examples in the Node.js documentation.

Rotate emulator certificates

You can force regenerate the emulator certificates by selecting **Reset Data** from the Azure Cosmos DB Emulator running in the Windows Tray. Note that this action will also wipe out all the data stored locally by the emulator.

If you have installed the certificate into the Java certificate store or used them elsewhere, you need to re-import them using the current certificates. Your application can't connect to the local emulator until you update the certificates.

Next steps

- Use command line parameters and PowerShell commands to control the emulator
- Debug issues with the emulator

Command-line and PowerShell reference for Azure Cosmos DB Emulator

12/4/2020 • 6 minutes to read • Edit Online

APPLIES TO: SQL API Cassandra API Gremlin API Table API Azure Cosmos DB API for MongoDB

The Azure Cosmos DB Emulator provides a local environment that emulates the Azure Cosmos DB service for local development purposes. After installing the emulator, you can control the emulator with command line and PowerShell commands. This article describes how to use the command-line and PowerShell commands to start and stop the emulator, configure options, and perform other operations. You have to run the commands from the installation location.

Manage the emulator with command-line syntax

```
Microsoft.Azure.Cosmos.Emulator.exe [/Shutdown] [/DataPath] [/Port] [/MongoPort] [/DirectPorts] [/Key]
[/EnableRateLimiting] [/DisableRateLimiting] [/NoUI] [/NoExplorer] [/EnableMongoDbEndpoint] [/?]
```

To view the list of options, type `Microsoft.Azure.Cosmos.Emulator.exe /?` at the command prompt.

OPTION	DESCRIPTION	COMMAND	ARGUMENTS
[No arguments]	Starts up the Azure Cosmos DB Emulator with default settings.	Microsoft.Azure.Cosmos.Emulator.exe	
[Help]	Displays the list of supported command-line arguments.	Microsoft.Azure.Cosmos.Emulator.exe /?	
GetStatus	Gets the status of the Azure Cosmos DB Emulator. The status is indicated by the exit code: 1 = Starting, 2 = Running, 3 = Stopped. A negative exit code indicates that an error occurred. No other output is produced.	Microsoft.Azure.Cosmos.Emulator.exe /GetStatus	
Shutdown	Shuts down the Azure Cosmos DB Emulator.	Microsoft.Azure.Cosmos.Emulator.exe /Shutdown	
DataPath	Specifies the path in which to store data files. Default value is %LocalAppdata%\CosmosDBEmulator.	Microsoft.Azure.Cosmos.Emulator.exe /DataPath=<datapath>	<datapath>: An accessible path
Port	Specifies the port number to use for the emulator. Default value is 8081.	Microsoft.Azure.Cosmos.Emulator.exe /Port=<port>	<port>: Single port number

OPTION	DESCRIPTION	COMMAND	ARGUMENTS
ComputePort	Specified the port number to use for the Compute Interop Gateway service. The Gateway's HTTP endpoint probe port is calculated as ComputePort + 79. Hence, ComputePort and ComputePort + 79 must be open and available. The default value is 8900.	Microsoft.Azure.Cosmos.Emulator.exe /ComputePort=<computeport>	<computeport>: Single port number
EnableMongoDbEndpoint=3.2	Enables MongoDB API 3.2	Microsoft.Azure.Cosmos.Emulator.exe /EnableMongoDbEndpoint=3.2	
EnableMongoDbEndpoint=3.6	Enables MongoDB API 3.6	Microsoft.Azure.Cosmos.Emulator.exe /EnableMongoDbEndpoint=3.6	
MongoPort	Specifies the port number to use for MongoDB compatibility API. Default value is 10255.	Microsoft.Azure.Cosmos.Emulator.exe /MongoPort=<mongoport>	<mongoport>: Single port number
EnableCassandraEndpoint	Enables Cassandra API	Microsoft.Azure.Cosmos.Emulator.exe /EnableCassandraEndpoint	
CassandraPort	Specifies the port number to use for the Cassandra endpoint. Default value is 10350.	Microsoft.Azure.Cosmos.Emulator.exe /CassandraPort=<cassandraport>	<cassandraport>: Single port number
EnableGremlinEndpoint	Enables Gremlin API	Microsoft.Azure.Cosmos.Emulator.exe /EnableGremlinEndpoint	
GremlinPort	Port number to use for the Gremlin Endpoint. Default value is 8901.	Microsoft.Azure.Cosmos.Emulator.exe /GremlinPort=<port>	<port>: Single port number
EnableTableEndpoint	Enables Azure Table API	Microsoft.Azure.Cosmos.Emulator.exe /EnableTableEndpoint	
TablePort	Port number to use for the Azure Table Endpoint. Default value is 8902.	Microsoft.Azure.Cosmos.Emulator.exe /TablePort=<port>	<port>: Single port number
KeyFile	Read authorization key from the specified file. Use the /GenKeyFile option to generate a keyfile	Microsoft.Azure.Cosmos.Emulator.exe /KeyFile=<file_name>	<file_name>: Path to the file

OPTION	DESCRIPTION	COMMAND	ARGUMENTS
ResetDataPath	Recursively removes all the files in the specified path. If you don't specify a path, it defaults to %LOCALAPPDATA%\CosmosDbEmulator	Microsoft.Azure.Cosmos.Emulator.exe /ResetDataPath=<path>	<path>: File path
StartTraces	Start collecting debug trace logs using LOGMAN.	Microsoft.Azure.Cosmos.Emulator.exe /StartTraces	
StopTraces	Stop collecting debug trace logs using LOGMAN.	Microsoft.Azure.Cosmos.Emulator.exe /StopTraces	
StartWprTraces	Start collecting debug trace logs using Windows Performance Recording tool.	Microsoft.Azure.Cosmos.Emulator.exe /StartWprTraces	
StopWprTraces	Stop collecting debug trace logs using Windows Performance Recording tool.	Microsoft.Azure.Cosmos.Emulator.exe /StopWprTraces	
FailOnSslCertificateNameMismatch	By default the emulator regenerates its self-signed TLS/SSL certificate, if the certificate's SAN does not include the emulator host's domain name, local IPv4 address, 'localhost', and '127.0.0.1'. With this option, the emulator will fail at startup instead. You should then use the /GenCert option to create and install a new self-signed TLS/SSL certificate.	Microsoft.Azure.Cosmos.Emulator.exe /FailOnSslCertificateNameMismatch	
GenCert	Generate and install a new self-signed TLS/SSL certificate. optionally including a comma-separated list of additional DNS names for accessing the emulator over the network.	Microsoft.Azure.Cosmos.Emulator.exe /GenCert=<dns-names>	<dns-names>: Optional comma-separated list of additional dns names
DirectPorts	Specifies the ports to use for direct connectivity. Defaults are 10251,10252,10253,10254.	Microsoft.Azure.Cosmos.Emulator.exe /DirectPorts:<directports>	<directports>: Comma-delimited list of 4 ports
Key	Authorization key for the emulator. Key must be the base-64 encoding of a 64-byte vector.	Microsoft.Azure.Cosmos.Emulator.exe /Key:<key>	<key>: Key must be the base-64 encoding of a 64-byte vector

OPTION	DESCRIPTION	COMMAND	ARGUMENTS
EnableRateLimiting	Specifies that request rate limiting behavior is enabled.	Microsoft.Azure.Cosmos.Emulator.exe /EnableRateLimiting	
DisableRateLimiting	Specifies that request rate limiting behavior is disabled.	Microsoft.Azure.Cosmos.Emulator.exe /DisableRateLimiting	
NoUI	Do not show the emulator user interface.	Microsoft.Azure.Cosmos.Emulator.exe /NoUI	
NoExplorer	Don't show data explorer on startup.	Microsoft.Azure.Cosmos.Emulator.exe /NoExplorer	
PartitionCount	Specifies the maximum number of partitioned containers. See Change the number of containers for more information.	Microsoft.Azure.Cosmos.Emulator.exe /PartitionCount= <partitioncount>	<partitioncount>: Maximum number of allowed single partition containers. Default value is 25. Maximum allowed is 250.
DefaultPartitionCount	Specifies the default number of partitions for a partitioned container.	Microsoft.Azure.Cosmos.Emulator.exe /DefaultPartitionCount= <defaultpartitioncount>	<defaultpartitioncount> Default value is 25.
AllowNetworkAccess	Enables access to the emulator over a network. You must also pass /Key= <key_string> or /KeyFile= <file_name> to enable network access.	Microsoft.Azure.Cosmos.Emulator.exe /AllowNetworkAccess /Key= <key_string> or Microsoft.Azure.Cosmos.Emulator.exe /AllowNetworkAccess /KeyFile=<file_name>	
NoFirewall	Don't adjust firewall rules when /AllowNetworkAccess option is used.	Microsoft.Azure.Cosmos.Emulator.exe /NoFirewall	
GenKeyFile	Generate a new authorization key and save to the specified file. The generated key can be used with the /Key or /KeyFile options.	Microsoft.Azure.Cosmos.Emulator.exe /GenKeyFile= <path to key file>	
Consistency	Set the default consistency level for the account.	Microsoft.Azure.Cosmos.Emulator.exe /Consistency= <consistency>	<consistency>: Value must be one of the following consistency levels: Session, Strong, Eventual, or BoundedStaleness. The default value is Session.
?	Show the help message.		

Manage the emulator with PowerShell

The emulator comes with a PowerShell module to start, stop, uninstall, and retrieve the status of the service. Run the following cmdlet to use the PowerShell module:

```
Import-Module "$env:ProgramFiles\Azure Cosmos DB Emulator\PSModules\Microsoft.Azure.CosmosDB.Emulator"
```

or place the `PSModules` directory on your `PSModulesPath` and import it as shown in the following command:

```
$env:PSModulesPath += "$env:ProgramFiles\Azure Cosmos DB Emulator\PSModules"
Import-Module Microsoft.Azure.CosmosDB.Emulator
```

Here is a summary of the commands for controlling the emulator from PowerShell:

`Get-CosmosDbEmulatorStatus`

Syntax

`Get-CosmosDbEmulatorStatus`

Remarks

Returns one of these ServiceControllerStatus values: ServiceControllerStatus.StartPending, ServiceControllerStatus.Running, or ServiceControllerStatus.Stopped.

`Start-CosmosDbEmulator`

Syntax

```
Start-CosmosDbEmulator [-DataPath <string>] [-DefaultPartitionCount <uint16>] [-DirectPort <uint16[]>] [-
MongoPort <uint16>] [-NoUI] [-NoWait] [-PartitionCount <uint16>] [-Port <uint16>] [<CommonParameters>]
```

Remarks

Starts the emulator. By default, the command waits until the emulator is ready to accept requests. Use the -NoWait option, if you wish the cmdlet to return as soon as it starts the emulator.

`Stop-CosmosDbEmulator`

Syntax

`Stop-CosmosDbEmulator [-NoWait]`

Remarks

Stops the emulator. By default, this command waits until the emulator is fully shut down. Use the -NoWait option, if you wish the cmdlet to return as soon as the emulator begins to shut down.

`Uninstall-CosmosDbEmulator`

Syntax

`Uninstall-CosmosDbEmulator [-RemoveData]`

Remarks

Uninstalls the emulator and optionally removes the full contents of $env:LOCALAPPDATA\CosmosDbEmulator. The cmdlet ensures the emulator is stopped before uninstalling it.

Change the number of default containers

By default, you can create up to 25 fixed size containers (only supported using Azure Cosmos DB SDKs), or 5 unlimited containers using the Azure Cosmos DB Emulator. By modifying the **PartitionCount** value, you can

create up to 250 fixed size containers or 50 unlimited containers, or any combination of the two that does not exceed 250 fixed size containers (where one unlimited container = 5 fixed size containers). However it's not recommended to set up the emulator to run with more than 200 fixed size containers. Because of the overhead that it adds to the disk IO operations, which result in unpredictable timeouts when using the endpoint APIs.

If you attempt to create a container after the current partition count has been exceeded, the emulator throws a ServiceUnavailable exception, with the following message.

> Sorry, we are currently experiencing high demand in this region, and cannot fulfill your request at this time. We work continuously to bring more and more capacity online, and encourage you to try again. ActivityId: 12345678-1234-1234-1234-123456789abc

To change the number of containers available in the Azure Cosmos DB Emulator, run the following steps:

1. Delete all local Azure Cosmos DB Emulator data by right-clicking the **Azure Cosmos DB Emulator** icon on the system tray, and then clicking **Reset Data...**.
2. Delete all emulator data in this folder `%LOCALAPPDATA%\CosmosDBEmulator`.
3. Exit all open instances by right-clicking the **Azure Cosmos DB Emulator** icon on the system tray, and then clicking **Exit**. It may take a minute for all instances to exit.
4. Install the latest version of the Azure Cosmos DB Emulator.
5. Launch the emulator with the PartitionCount flag by setting a value <= 250. For example: `C:\Program Files\Azure Cosmos DB Emulator> Microsoft.Azure.Cosmos.Emulator.exe /PartitionCount=100`.

Next steps

- Export the Azure Cosmos DB Emulator certificates for use with Java, Python, and Node.js apps
- Debug issues with the emulator

Troubleshoot issues when using the Azure Cosmos DB Emulator

12/4/2020 • 3 minutes to read • Edit Online

APPLIES TO: ✅ SQL API ✅ Cassandra API ✅ Gremlin API ✅ Table API ✅ Azure Cosmos DB API for MongoDB

The Azure Cosmos DB Emulator provides a local environment that emulates the Azure Cosmos DB service for development purposes. Use the tips in this article to help troubleshoot issues you encounter when installing or using the Azure Cosmos DB Emulator.

If you installed a new version of the emulator and are experiencing errors, ensure you reset your data. You can reset your data by right-clicking the Azure Cosmos DB Emulator icon on the system tray, and then clicking Reset Data.... If that does not fix the errors, you can uninstall the emulator and any older versions of the emulator if found, remove *C:\Program files\Azure Cosmos DB Emulator* directory and reinstall the emulator. See Uninstall the local emulator for instructions. Alternatively if resetting the data doesn't work, navigate to `%LOCALAPPDATA%\CosmosDBEmulator` location and delete the folder.

Troubleshoot corrupted windows performance counters

- If the Azure Cosmos DB Emulator crashes, collect the dump files from `%LOCALAPPDATA%\CrashDumps` folder, compress them, and open a support ticket from the Azure portal.
- If you experience crashes in `Microsoft.Azure.Cosmos.ComputeServiceStartupEntryPoint.exe`, this might be a symptom where the Performance Counters are in a corrupted state. Usually running the following command from an admin command prompt fixes the issue:

```
lodctr /R
```

Troubleshoot connectivity issues

- If you encounter a connectivity issue, collect trace files, compress them, and open a support ticket in the Azure portal.
- If you receive a **Service Unavailable** message, the emulator might be failing to initialize the network stack. Check to see if you have the Pulse secure client or Juniper networks client installed, as their network filter drivers may cause the problem. Uninstalling third-party network filter drivers typically fixes the issue. Alternatively, start the emulator with /DisableRIO, which will switch the emulator network communication to regular Winsock.
- If you encounter **"Forbidden","message":"Request is being made with a forbidden encryption in transit protocol or cipher. Check account SSL/TLS minimum allowed protocol setting..."** connectivity issues, this might be caused by global changes in the OS (for example Insider Preview Build 20170) or the browser settings that enable TLS 1.3 as default. Similar error might occur when using the SDK to execute a request against the Cosmos emulator, such as **Microsoft.Azure.Documents.DocumentClientException: Request is being made with a forbidden encryption in transit protocol or cipher. Check account SSL/TLS minimum allowed protocol setting**. This is expected at this time since Cosmos emulator only accepts and works with TLS 1.2 protocol. The recommended work-around is to change the settings and default to TLS 1.2; for instance, in IIS

Manager navigate to "Sites" -> "Default Web Sites" and locate the "Site Bindings" for port 8081 and edit them to disable TLS 1.3. Similar operation can be performed for the Web browser via the "Settings" options.

- While the emulator is running, if your computer goes to sleep mode or runs any OS updates, you might see a **Service is currently unavailable** message. Reset the emulator's data, by right-clicking on the icon that appears on the windows notification tray and select **Reset Data**.

Collect trace files

To collect debugging traces, run the following commands from an administrative command prompt:

1. Navigate to the path where the emulator is installed:

```
cd /d "%ProgramFiles%\Azure Cosmos DB Emulator"
```

2. Shut down the emulator and watch the system tray to make sure the program has shut down. It may take a minute to complete. You can also select **Exit** in the Azure Cosmos DB Emulator user interface.

```
Microsoft.Azure.Cosmos.Emulator.exe /shutdown
```

3. Start logging with the following command:

```
Microsoft.Azure.Cosmos.Emulator.exe /startwprtraces
```

4. Launch the emulator

```
Microsoft.Azure.Cosmos.Emulator.exe
```

5. Reproduce the problem. If the data explorer is not working, you only need to wait for the browser to open for a few seconds to catch the error.

6. Stop logging with the following command:

```
Microsoft.Azure.Cosmos.Emulator.exe /stopwprtraces
```

7. Navigate to `%ProgramFiles%\Azure Cosmos DB Emulator` path and find the *docdbemulator_000001.etl* file.

8. Open a support ticket in the Azure portal and include the .etl file along with repro steps.

Next steps

In this article, you've learned how to debug issues with the local emulator. You can now proceed to the next articles:

- Export the Azure Cosmos DB Emulator certificates for use with Java, Python, and Node.js apps
- Use command line parameters and PowerShell commands to control the emulator

Azure Cosmos DB Emulator - Release notes and download information

12/4/2020 • 3 minutes to read • Edit Online

APPLIES TO: SQL API Cassandra API Gremlin API Table API Azure Cosmos DB API for MongoDB

This article shows the Azure Cosmos DB Emulator release notes with a list of feature updates that were made in each release. It also lists the latest version of emulator to download and use.

Download

MSI download	Microsoft Download Center
Get started	Develop locally with Azure Cosmos Emulator

Release notes

2.11.9 (3 December 2020)

- This release addresses couple issues with the Azure Cosmos DB Emulator functionality in addition to the general content update reflecting the latest features and improvements in Azure Cosmos DB:
- Fix for an issue where large document payload requests fail when using Direct mode and Java client applications.
- Fix for a connectivity issue with MongoDB endpoint version 3.6 when targeted by .NET based applications.

2.11.8 (6 November 2020)

- This release includes an update for the Cosmos emulator Data Explorer and fixes an issue where TLS 1.3 clients try to open the Data Explorer.

2.11.6 (6 October 2020)

- This release addresses a concurrency related issue when multiple containers might be created at the same time. In such cases emulator's data is left in a corrupted state and following API requests to the emulator's endpoint could fail with "service unavailable" errors, requiring a restart and a reset of the emulator's local data.

2.11.5 (23 August 2020)

This release adds two new Cosmos emulator startup options:

- "/EnablePreview" - it enables preview features for the emulator. The preview features that are still under development and they can be accessed via CI and sample writing.
- "/EnableAadAuthentication" - it enables the emulator to accept custom Azure Active Directory tokens as an alternative to the Azure Cosmos primary keys. This feature is still under development; specific role assignments and other permission-related settings aren't currently supported.

2.11.2 (07 July 2020)

- This release changes how ETL traces required when troubleshooting the Cosmos emulator are collected. WPR (Windows Performance Runtime tools) is now the default tools for capturing ETL-based traces while old LOGMAN based capturing has been deprecated. This change is required in part because latest Windows

security updates had an unexpected impact on how LOGMAN works when executed through the Cosmos emulator.

2.11.1 (10 June 2020)

- This release fixes couple bugs related to emulator Data Explorer. In certain cases when using the emulator Data Explorer through a web browser, it fails to connect to the Cosmos emulator endpoint and all the related actions such as creating a database or a container will result in error. The second issue fixed is related to creating an item from a JSON file using Data Explorer upload action.

2.11.0

- This release introduces support for autoscale provisioned throughput. These new features include the ability to set a custom maximum provisioned throughput level in request units (RU/s), enable autoscale on existing databases and containers, and programmatic support through Azure Cosmos DB SDKs.
- Fix an issue while querying through large number of documents (over 1 GB) were the emulator will fail with internal error status code 500.

2.9.2

- This release fixes a bug while enabling support for MongoDb endpoint version 3.2. It also adds support for generating ETL traces for troubleshooting purposes using WPR instead of LOGMAN.

2.9.1

- This release fixes couple issues in the query API support and restores compatibility with older OSs such as Windows Server 2012.

2.9.0

- This release adds the option to set the consistency to consistent prefix and increase the maximum limits for users and permissions.

2.7.2

- This release adds MongoDB version 3.6 server support to the Cosmos Emulator. To start a MongoDB endpoint that target version 3.6 of the service, start the emulator from an Administrator command line with "/EnableMongoDBEndpoint=3.6" option.

2.7.0

- This release fixes a regression, which prevented users from executing queries against the SQL API account from the emulator when using .NET core or x86 .NET based clients.

2.4.6

- This release provides parity with the features in the Azure Cosmos service as of July 2019, with the exceptions noted in Develop locally with Azure Cosmos Emulator. It also fixes several bugs related to emulator shut down when invoked via command line and internal IP address overrides for SDK clients using direct mode connectivity.

2.4.3

- Disabled starting the MongoDB service by default. Only the SQL endpoint is enabled as default. The user must start the endpoint manually using the emulator's "/EnableMongoDbEndpoint" command-line option. Now, it's like all the other service endpoints, such as Gremlin, Cassandra, and Table.
- Fixed a bug in the emulator when starting with “/AllowNetworkAccess” where the Gremlin, Cassandra, and Table endpoints weren't properly handling requests from external clients.
- Add direct connection ports to the Firewall Rules settings.

2.4.0

- Fixed an issue with emulator failing to start when network monitoring apps, such as Pulse Client, are present on the host computer.

Set up a CI/CD pipeline with the Azure Cosmos DB Emulator build task in Azure DevOps

12/4/2020 • 6 minutes to read • Edit Online

APPLIES TO: SQL API Cassandra API Gremlin API Table API Azure Cosmos DB API for MongoDB

The Azure Cosmos DB Emulator provides a local environment that emulates the Azure Cosmos DB service for development purposes. The emulator allows you to develop and test your application locally, without creating an Azure subscription or incurring any costs.

The Azure Cosmos DB Emulator build task for Azure DevOps allows you to do the same in a CI environment. With the build task, you can run tests against the emulator as part of your build and release workflows. The task spins up a Docker container with the emulator already running and provides an endpoint that can be used by the rest of the build definition. You can create and start as many instances of the emulator as you need, each running in a separate container.

This article demonstrates how to set up a CI pipeline in Azure DevOps for an ASP.NET application that uses the Cosmos DB emulator build task to run tests. You can use a similar approach to set up a CI pipeline for a Node.js or a Python application.

Install the emulator build task

To use the build task, we first need to install it onto our Azure DevOps organization. Find the extension **Azure Cosmos DB Emulator** in the Marketplace and click **Get it free**.

Next, choose the organization in which to install the extension.

> **NOTE**
>
> To install an extension to an Azure DevOps organization, you must be an account owner or project collection administrator. If you do not have permissions, but you are an account member, you can request extensions instead. Learn more.

Create a build definition

Now that the extension is installed, sign in to your Azure DevOps organization and find your project from the projects dashboard. You can add a build pipeline to your project or modify an existing build pipeline. If you already have a build pipeline, you can skip ahead to Add the Emulator build task to a build definition.

1. To create a new build definition, navigate to the **Builds** tab in Azure DevOps. Select **+New.** > **New build pipeline**

2. Select the desired **source**, **Team project**, **Repository**, and the **Default branch for manual and scheduled builds**. After choosing the required options, select **Continue**

3. Finally, select the desired template for the build pipeline. We'll select the **ASP.NET** template in this tutorial. Now you have a build pipeline that you can set up to use the Azure Cosmos DB Emulator build task.

> **NOTE**
> The agent pool to be selected for this CI should have Docker for Windows installed unless the installation is done manually in a prior task as a part of the CI. See Microsoft hosted agents article for a selection of agent pools; we recommend to start with `Hosted VS2017`.

Azure Cosmos DB Emulator currently doesn't support hosted VS2019 agent pool. However, the emulator already comes with VS2019 installed and you use it by starting the emulator with the following PowerShell cmdlets. If you run into any issues when using the VS2019, reach out to the Azure DevOps team for help:

```
Import-Module "$env:ProgramFiles\Azure Cosmos DB Emulator\PSModules\Microsoft.Azure.CosmosDB.Emulator"
Start-CosmosDbEmulator
```

Add the task to a build pipeline

1. Before adding a task to the build pipeline, you should add an agent job. Navigate to your build pipeline, select the **...** and choose **Add an agent job**.

2. Next select the **+** symbol next to the agent job to add the emulator build task. Search for **cosmos** in the search box, select **Azure Cosmos DB Emulator** and add it to the agent job. The build task will start up a container with an instance of the Cosmos DB emulator already running on it. The Azure Cosmos DB Emulator task should be placed before any other tasks that expect the emulator to be in running state.

In this tutorial, you'll add the task to the beginning to ensure the emulator is available before our tests execute.

Add the task using YAML

This step is optional and it's only required if you are setting up the CI/CD pipeline by using a YAML task. In such cases, you can define the YAML task as shown in the following code:

```
- task: azure-cosmosdb.emulator-public-preview.run-cosmosdbemulatorcontainer.CosmosDbEmulator@2
  displayName: 'Run Azure Cosmos DB Emulator'

- script: yarn test
  displayName: 'Run API tests (Cosmos DB)'
  env:
    HOST: $(CosmosDbEmulator.Endpoint)
    # Hardcoded key for emulator, not a secret
    AUTH_KEY: C2y6yDjf5/R+ob0N8A7Cgv30VRDJIWEHLM+4QDU5DE2nQ9nDuVTqobD4b8mGGyPMbIZnqyMsEcaGQy67XIw/Jw==
    # The emulator uses a self-signed cert, disable TLS auth errors
    NODE_TLS_REJECT_UNAUTHORIZED: '0'
```

Configure tests to use the emulator

Now, we'll configure our tests to use the emulator. The emulator build task exports an environment variable – 'CosmosDbEmulator.Endpoint' – that any tasks further in the build pipeline can issue requests against.

In this tutorial, we'll use the Visual Studio Test task to run unit tests configured via a **.runsettings** file. To learn more about unit test setup, visit the documentation. The complete Todo application code sample that you use in this document is available on GitHub

Below is an example of a **.runsettings** file that defines parameters to be passed into an application's unit tests. Note the `authKey` variable used is the well-known key for the emulator. This `authKey` is the key expected by the emulator build task and should be defined in your **.runsettings** file.

```
<RunSettings>
    <TestRunParameters>
    <Parameter name="endpoint" value="https://localhost:8081" />
    <Parameter name="authKey"
value="C2y6yDjf5/R+ob0N8A7Cgv30VRDJIWEHLM+4QDU5DE2nQ9nDuVTqobD4b8mGGyPMbIZnqyMsEcaGQy67XIw/Jw==" />
    <Parameter name="database" value="ToDoListTest" />
    <Parameter name="collection" value="ItemsTest" />
  </TestRunParameters>
</RunSettings>
```

If you are setting up a CI/CD pipeline for an application that uses the Azure Cosmos DB's API for MongoDB, the connection string by default includes the port number 10255. However, this port is not currently open, as an alternate, you should use port 10250 to establish the connection. The Azure Cosmos DB's API for MongoDB connection string remains the same except the supported port number is 10250 instead of 10255.

These parameters `TestRunParameters` are referenced via a `TestContext` property in the application's test project. Here is an example of a test that runs against Cosmos DB.

```
namespace todo.Tests
{
    [TestClass]
    public class TodoUnitTests
    {
        public TestContext TestContext { get; set; }

        [TestInitialize()]
        public void Initialize()
        {
            string endpoint = TestContext.Properties["endpoint"].ToString();
            string authKey = TestContext.Properties["authKey"].ToString();
            Console.WriteLine("Using endpoint: ", endpoint);
            DocumentDBRepository<Item>.Initialize(endpoint, authKey);
        }
        [TestMethod]
        public async Task TestCreateItemsAsync()
        {
            var item = new Item
            {
                Id = "1",
                Name = "testName",
                Description = "testDescription",
                Completed = false,
                Category = "testCategory"
            };

            // Create the item
            await DocumentDBRepository<Item>.CreateItemAsync(item);
            // Query for the item
            var returnedItem = await DocumentDBRepository<Item>.GetItemAsync(item.Id, item.Category);
            // Verify the item returned is correct.
            Assert.AreEqual(item.Id, returnedItem.Id);
            Assert.AreEqual(item.Category, returnedItem.Category);
        }

        [TestCleanup()]
        public void Cleanup()
        {
            DocumentDBRepository<Item>.Teardown();
        }
    }
}
```

Navigate to the Execution Options in the Visual Studio Test task. In the **Settings file** option, specify that the tests are configured using the **.runsettings** file. In the **Override test run parameters** option, add in `-endpoint $(CosmosDbEmulator.Endpoint)`. Doing so will configure the Test task to refer to the endpoint of the emulator build task, instead of the one defined in the **.runsettings** file.

Run the build

Now, **Save and queue** the build.

Once the build has started, observe the Cosmos DB emulator task has begun pulling down the Docker image with the emulator installed.

After the build completes, observe that your tests pass, all running against the Cosmos DB emulator from the build task!

Next steps

To learn more about using the emulator for local development and testing, see Use the Azure Cosmos DB Emulator for local development and testing.

To export emulator TLS/SSL certificates, see Export the Azure Cosmos DB Emulator certificates for use with Java, Python, and Node.js

Manage Azure Cosmos DB resources by using Azure Storage Explorer

12/4/2020 • 10 minutes to read • Edit Online

APPLIES TO: SQL API Cassandra API Gremlin API Table API Azure Cosmos DB API for MongoDB

You can use Azure Storage explorer to connect to Azure Cosmos DB. It lets you connect to Azure Cosmos DB accounts hosted on Azure and sovereign clouds from Windows, macOS, or Linux.

Use the same tool to manage your different Azure entities in one place. You can manage Azure Cosmos DB entities, manipulate data, update stored procedures and triggers along with other Azure entities like storage blobs and queues. Azure Storage Explorer supports Cosmos accounts configured for SQL, MongoDB, Graph, and Table APIs.

> **NOTE**
>
> The Azure Cosmos DB integration with Storage Explorer has been deprecated. Any existing functionality will not be removed for a minimum of one year from this release. You should use the Azure Portal, Azure Portal desktop app or the standalone Azure Cosmos DB Explorer instead. The alternative options contain many new features that aren't currently supported in Storage Explorer.

Prerequisites

A Cosmos account with a SQL API or an Azure Cosmos DB API for MongoDB. If you don't have an account, you can create one in the Azure portal. See Azure Cosmos DB: Build a SQL API web app with .NET and the Azure portal for more information.

Installation

To install the newest Azure Storage Explorer bits, see Azure Storage Explorer. We support Windows, Linux, and macOS versions.

Connect to an Azure subscription

1. After you install **Azure Storage Explorer**, select the **plug-in** icon on the left pane.

2. Select **Add an Azure Account**, and then select **Sign-in**.

3. In the **Azure Sign-in** dialog box, select **Sign in**, and then enter your Azure credentials.

4. Select your subscription from the list, and then select **Apply**.

The Explorer pane updates and shows the accounts in the selected subscription.

Your **Cosmos DB account** is connected to your Azure subscription.

Use a connection string to connect to Azure Cosmos DB

You can use a connection string to connect to an Azure Cosmos DB. This method only supports SQL and Table APIs. Follow these steps to connect with a connection string:

1. Find **Local and Attached** in the left tree, right-click **Cosmos DB Accounts**, and then select **Connect to Cosmos DB**.

2. In the **Connect to Cosmos DB** window:

   a. Select the API from the drop-down menu.

   b. Paste your connection string in the **Connection string** box. For how to retrieve the primary connection string, see Get the connection string.

   c. Enter an **Account label**, and then select **Next** to check the summary.

   d. Select **Connect** to connect the Azure Cosmos DB account.

> **NOTE**
>
> If the Azure Storage Explorer shows that the Azure Cosmos DB connection string is in an invalid format, make sure that the connection string has a semicolon (`;`) at the end. An example of a valid Azure Cosmos DB connection string would be:
> `AccountEndpoint=https://accountname.documents.azure.com:443;AccountKey=accountkey==;`

Use a local emulator to connect to Azure Cosmos DB

Use the following steps to connect to an Azure Cosmos DB with an emulator. This method only supports SQL accounts.

1. Install Cosmos DB Emulator, and then open it. For how to install the emulator, see Cosmos DB Emulator.
2. Find **Local and Attached** in the left tree, right-click **Cosmos DB Accounts**, and then select **Connect to Cosmos DB Emulator**.

3. In the **Connect to Cosmos DB** window:
   a. Paste your connection string in the **Connection string** box. For information on retrieving the primary connection string, see Get the connection string.
   b. Enter an **Account label**, and then select **Next** to check the summary.
   c. Select **Connect** to connect the Azure Cosmos DB account.

Azure Cosmos DB resource management

Use the following operations to manage an Azure Cosmos DB account:

- Open the account in the Azure portal.
- Add the resource to the Quick Access list.
- Search and refresh resources.
- Create and delete databases.
- Create and delete collections.
- Create, edit, delete, and filter documents.
- Manage stored procedures, triggers, and user-defined functions.

Quick access tasks

You can right-click a subscription on the Explorer pane to perform many quick action tasks, for example:

- Right-click an Azure Cosmos DB account or database, and then select **Open in Portal** to manage the resource in the browser on the Azure portal.

- Right-click an Azure Cosmos DB account, database, or collection, and then select **Add to Quick Access** to add it to the Quick Access menu.
- Select **Search from Here** to enable keyword search under the selected path.

Database and collection management

Create a database

1. Right-click the Azure Cosmos DB account, and then select **Create Database**.

2. Enter the database name, and then press **Enter** to complete.

Delete a database

1. Right-click the database, and then select **Delete Database**.

2. Select **Yes** in the pop-up window. The database node is deleted, and the Azure Cosmos DB account refreshes automatically.

Create a collection

1. Right-click your database, and then select **Create Collection**.

2. In the Create Collection window, enter the requested information, like **Collection ID** and **Storage capacity**, and so on. Select **OK** to finish.

3. Select **Unlimited** so you can specify a partition key, then select **OK** to finish.

> **NOTE**
>
> If a partition key is used when you create a collection, once creation is completed, you can't change the partition key value on the collection.

Delete a collection

- Right-click the collection, select **Delete Collection**, and then select **Yes** in the pop-up window.

  The collection node is deleted, and the database refreshes automatically.

Document management

Create and modify documents

- Open **Documents** on the left pane, select **New Document**, edit the contents on the right pane, and then select **Save**.

- You can also update an existing document, and then select **Save**. To discard changes, select **Discard**.

Delete a document

- Select the **Delete** button to delete the selected document.

Query for documents

- To edit the document filter, enter a SQL query, and then select **Apply**.

Graph management

Create and modify a vertex

- To create a new vertex, open **Graph** from the left pane, select **New Vertex**, edit the contents, and then select **OK**.
- To modify an existing vertex, select the pen icon on the right pane.

Delete a graph

- To delete a vertex, select the recycle bin icon beside the vertex name.

Filter for graph

- To edit the graph filter, enter a gremlin query, and then select **Apply Filter**.

Table management

Create and modify a table

- To create a new table:

  1. On the left pane, open **Entities**, and then select **Add**.
  2. In the **Add Entity** dialog box, edit the content.
  3. Select the **Add Property** button to add a property.
  4. Select **Insert**.

- To modify a table, select **Edit**, modify the content, and then select **Update**.

Import and export table

- To import, select the **Import** button, and then choose an existing table.
- To export, select the **Export** button, and then choose a destination.

Delete entities

- Select the entities, and then select the **Delete** button.

Query a table

- Select the **Query** button, input a query condition, and then select the **Execute Query** button. To close the query pane, select the **Close Query** button.

Manage stored procedures, triggers, and UDFs

- To create a stored procedure:

  1. In the left tree, right-click **Stored Procedures**, and then select **Create Stored Procedure**.

  2. Enter a name in the left, enter the stored procedure scripts on the right pane, and then select **Create**.

- To edit an existing stored procedure, double-click the procedure, make the update, and then select **Update** to save. You can also select **Discard** to cancel the change.

- The operations for **Triggers** and **UDF** are similar to **Stored Procedures**.

Troubleshooting

The following are solutions to common issues that arise when you use Azure Cosmos DB in Storage Explorer.

Sign in issues

First, restart your application to see if that fixes the problem. If the problem persists, continue troubleshooting.

Self-signed certificate in certificate chain

There are a few reasons you might be seeing this error, the two most common ones are:

- You're behind a *transparent proxy*. Someone, like your IT department, intercepts HTTPS traffic, decrypts it, and then encrypts it by using a self-signed certificate.
- You're running software, such as antivirus software. The software injects a self-signed TLS/SSL certificate into the HTTPS messages you receive.

When Storage Explorer finds a self-signed certificate, it doesn't know if the HTTPS message it receives is tampered with. If you have a copy of the self-signed certificate, you can tell Storage Explorer to trust it. If you're unsure of who injected the certificate, then you can follow these steps to try to find out:

1. Install OpenSSL:
   - Windows: Any of the light versions are OK.
   - macOS and Linux: Should be included with your operating system.
2. Run OpenSSL:
   - Windows: Go to the install directory, then **/bin/**, then double-click **openssl.exe**.
   - Mac and Linux: Execute **openssl** from a terminal.
3. Execute `s_client -showcerts -connect microsoft.com:443`.
4. Look for self-signed certificates. If you're unsure, which are self-signed, then look for anywhere that the subject ("s:") and issuer ("i:") are the same.
5. If you find any self-signed certificates, copy and paste everything from and including **-----BEGIN CERTIFICATE-----** to **-----END CERTIFICATE-----** to a new .CER file for each one.
6. Open Storage Explorer, and then go to **Edit** > **SSL Certificates** > **Import Certificates**. Use the file picker to find, select, and then open the .CER files you created.

If you don't find any self-signed certificates, you can send feedback for more help.

Unable to retrieve subscriptions

If you're unable to retrieve your subscriptions after you sign in, try these suggestions:

- Verify that your account has access to the subscriptions. To do this, sign in to the Azure portal.
- Make sure you signed in to the correct environment:
  - Azure
  - Azure China
  - Azure Germany
  - Azure US Government
  - Custom Environment/Azure Stack
- If you're behind a proxy, make sure that the Storage Explorer proxy is properly configured.
- Remove the account, and then add it again.
- Delete the following files from your home directory (such as: C:\Users\ContosoUser), and then add the account again:
  - .adalcache
  - .devaccounts
  - .extaccounts
- Press the F12 key to open the developer console. Watch the console for any error messages when you sign in.

Unable to see the authentication page

If you're unable to see the authentication page:

- Depending on the speed of your connection, it might take a while for the sign-in page to load. Wait at least one minute before you close the authentication dialog box.
- If you're behind a proxy, make sure that the Storage Explorer proxy is properly configured.
- On the developer tools console (F12), watch the responses to see if you can find any clue for why authentication isn't working.

Can't remove an account

If you're unable to remove an account, or if the reauthenticate link doesn't do anything:

- Delete the following files from your home directory, and then add the account again:
  - .adalcache
  - .devaccounts
  - .extaccounts
- If you want to remove SAS attached Storage resources, delete:
  - %AppData%/StorageExplorer folder for Windows
  - /Users/<your_name>/Library/Application SUpport/StorageExplorer for macOS
  - ~/.config/StorageExplorer for Linux

> **NOTE**
>
> If you delete these files, **you must reenter all your credentials**.

HTTP/HTTPS proxy issue

You can't list Azure Cosmos DB nodes in the left tree when you configure an HTTP/HTTPS proxy in ASE. You can use Azure Cosmos DB data explorer in the Azure portal as a work-around.

"Development" node under "Local and Attached" node issue

There's no response after you select the **Development** node under the **Local and Attached** node in the left tree. The behavior is expected.

Attach an Azure Cosmos DB account in the Local and Attached node error

If you see the following error after you attach an Azure Cosmos DB account in **Local and Attached** node, then make sure you're using the correct connection string.

Expand Azure Cosmos DB node error

You might see the following error when you try to expand nodes in the left tree.

Try these suggestions:

- Check if the Azure Cosmos DB account is in provision progress. Try again when the account is being created successfully.
- If the account is under the **Quick Access** or **Local and Attached** nodes, check if the account is deleted. If so, you need to manually remove the node.

Next steps

- Watch this video to see how to use Azure Cosmos DB in Azure Storage Explorer: Use Azure Cosmos DB in Azure Storage Explorer.
- Learn more about Storage Explorer and connect more services in Get started with Storage Explorer.

Work with data using Azure Cosmos DB Explorer

12/4/2020 • 3 minutes to read • Edit Online

APPLIES TO: SQL API Cassandra API Gremlin API Table API Azure Cosmos DB API for MongoDB

Azure Cosmos DB Explorer is a standalone web-based interface that allows you to view and manage the data stored in Azure Cosmos DB. Azure Cosmos DB Explorer is equivalent to the existing **Data Explorer** tab that is available in Azure portal when you create an Azure Cosmos DB account. The key advantages of Azure Cosmos DB Explorer over the existing Data explorer are:

- You have a full screen real-estate to view your data, run queries, stored procedures, triggers, and view their results.
- You can provide temporary or permanent read or read-write access to your database account and its collections to other users who do not have access to Azure portal or subscription.
- You can share the query results with other users who do not have access to Azure portal or subscription.

Access Azure Cosmos DB Explorer

1. Sign in to Azure portal.
2. From **All resources**, find and navigate to your Azure Cosmos DB account, select Keys, and copy the **Primary Connection String**.
3. Go to https://cosmos.azure.com/, paste the connection string and select **Connect**. By using the connection string, you can access the Azure Cosmos DB Explorer without any time limits.

   If you want to provide other users temporary access to your Azure Cosmos DB account, you can do so by using the read-write and read access URLs.
4. Open the **Data Explorer** blade, select **Open Full Screen**. From the pop-up dialog, you can view two access URLs – **Read-Write** and **Read**. These URLs allow you to share your Azure Cosmos DB account temporarily with other users. Access to the account expires in 24 hours after which you can reconnect by using a new access URL or the connection string.

   Read-Write – When you share the Read-Write URL with other users, they can view and modify the databases, collections, queries, and other resources associated with that specific account.

   Read - When you share the read-only URL with other users, they can view the databases, collections, queries, and other resources associated with that specific account. For example, if you want to share results of a query with your teammates who don't have access to Azure portal or your Azure Cosmos DB account, you can provide them with this URL.

   Choose the type of access you'd like to open the account with and click **Open**. After you open the explorer, the experience is same as you had with the Data Explorer tab in Azure portal.

Known issues

Currently the **Open Full Screen** experience that allows you to share temporary read-write or read access is not yet supported for Azure Cosmos DB Gremlin and Table API accounts. You can still view your Gremlin and Table API accounts by passing the connection string to Azure Cosmos DB Explorer.

Currently, viewing documents that contain a UUID is not supported in Data Explorer. This does not affect loading collections, only viewing individual documents or queries that include these documents. To view and manage these documents, users should continue to use the tool that was originally used to create these documents.

Customers receiving HTTP-401 errors may be due to insufficient Azure RBAC permissions for the customer's Azure account, particularly if the account has a custom role. Any custom roles must have `Microsoft.DocumentDB/databaseAccounts/listKeys/*` action to use Data Explorer if signing in using their Azure Active Directory credentials.

Next steps

Now that you have learned how to get started with Azure Cosmos DB Explorer to manage your data, next you can:

- Start defining queries using SQL syntax and perform server side programming by using stored procedures, UDFs, triggers.

Configure and use Azure Synapse Link for Azure Cosmos DB

12/4/2020 • 8 minutes to read • Edit Online

APPLIES TO: SQL API Azure Cosmos DB API for MongoDB

Azure Synapse Link for Azure Cosmos DB is a cloud-native hybrid transactional and analytical processing (HTAP) capability that enables you to run near real-time analytics over operational data in Azure Cosmos DB. Synapse Link creates a tight seamless integration between Azure Cosmos DB and Azure Synapse Analytics.

Azure Synapse Link is available for Azure Cosmos DB SQL API containers or for Azure Cosmos DB API for Mongo DB collections. Use the following steps to run analytical queries with the Azure Synapse Link for Azure Cosmos DB:

- Enable Synapse Link for your Azure Cosmos DB accounts
- Create an analytical store enabled Azure Cosmos DB container
- Connect your Azure Cosmos DB database to a Synapse workspace
- Query the analytical store using Synapse Spark
- Query the analytical store using serverless SQL pool
- Use serverless SQL pool to analyze and visualize data in Power BI

Enable Azure Synapse Link for Azure Cosmos DB accounts

Azure portal

1. Sign into the Azure portal.
2. Create a new Azure account, or select an existing Azure Cosmos DB account.
3. Navigate to your Azure Cosmos DB account and open the **Features** pane.
4. Select **Synapse Link** from the features list.

5. Next it prompts you to enable synapse link on your account. Select **Enable**. This process can take 1 to 5 minutes to complete.

6. Your account is now enabled to use Synapse Link. Next see how to create analytical store enabled containers to automatically start replicating your operational data from the transactional store to the analytical store.

> **NOTE**
> Turning on Synapse Link does not turn on the analytical store automatically. Once you enable Synapse Link on the Cosmos DB account, enable analytical store on containers when you create them, to start replicating your operation data to analytical store.

Create an Azure Cosmos container with analytical store

You can turn on analytical store on an Azure Cosmos container while creating the container. You can use the Azure portal or configure the `analyticalTTL` property during container creation by using the Azure Cosmos DB SDKs.

> **NOTE**
> Currently you can enable analytical store for **new** containers (both in new and existing accounts). You can migrate data from your exisitng containers to new containers using Azure Cosmos DB migration tools.

Azure portal

1. Sign in to the Azure portal or the Azure Cosmos DB Explorer.
2. Navigate to your Azure Cosmos DB account and open the **Data Explorer** tab.
3. Select **New Container** and enter a name for your database, container, partition key and throughput details. Turn on the **Analytical store** option. After you enable the analytical store, it creates a container with `AnalyicalTTL` property set to the default value of -1 (infinite retention). This analytical store that retains all the historical versions of records.

4. If you have previously not enabled Synapse Link on this account, it will prompt you to do so because it's a pre-requisite to create an analytical store enabled container. If prompted, select **Enable Synapse Link**. This process can take 1 to 5 minutes to complete.
5. Select **OK**, to create an analytical store enabled Azure Cosmos container.
6. After the container is created, verify that analytical store has been enabled by clicking **Settings**, right below Documents in Data Explorer, and check if the **Analytical Store Time to Live** option is turned on.

.NET SDK

The following code creates a container with analytical store by using the .NET SDK. Set the analytical TTL property to the required value. For the list of allowed values, see the analytical TTL supported values article:

```
// Create a container with a partition key, and analytical TTL configured to  -1 (infinite retention)
string containerId = “myContainerName”;
int analyticalTtlInSec = -1;
ContainerProperties cpInput = new ContainerProperties()
            {
Id = containerId,
PartitionKeyPath = "/id",
AnalyticalStorageTimeToLiveInSeconds = analyticalTtlInSec,
};
 await this. cosmosClient.GetDatabase("myDatabase").CreateContainerAsync(cpInput);
```

Java V4 SDK

The following code creates a container with analytical store by using the Java V4 SDK. Set the `AnalyticalStoreTimeToLiveInSeconds` property to the required value:

```
// Create a container with a partition key and  analytical TTL configured to  -1 (infinite retention)
CosmosContainerProperties containerProperties = new CosmosContainerProperties("myContainer",
"/myPartitionKey");

containerProperties.setAnalyticalStoreTimeToLiveInSeconds(-1);

container = database.createContainerIfNotExists(containerProperties, 400).block().getContainer();
```

Python V4 SDK

Python 2.7 and Azure Cosmos DB SDK 4.1.0 are the minimum versions required, and the SDK is only compatible with the SQL API.

The first step is to make sure that you are using at least version 4.1.0 of the Azure Cosmos DB Python SDK:

```
import azure.cosmos as cosmos

print (cosmos.__version__)
```

The next step creates a container with analytical store by using the Azure Cosmos DB Python SDK:

```
# Azure Cosmos DB Python SDK, for SQL API only.
# Creating an analytical store enabled container.

import azure.cosmos.cosmos_client as cosmos_client
import azure.cosmos.exceptions as exceptions
from azure.cosmos.partition_key import PartitionKey

HOST = 'your-cosmos-db-account-URI'
KEY = 'your-cosmos-db-account-key'
DATABASE = 'your-cosmos-db-database-name'
CONTAINER = 'your-cosmos-db-container-name'

client = cosmos_client.CosmosClient(HOST,  KEY )
# setup database for this sample.
# If doesn't exist, creates a new one with the name informed above.
try:
    db = client.create_database(DATABASE)

except exceptions.CosmosResourceExistsError:
    db = client.get_database_client(DATABASE)

# Creating the container with analytical store enabled, using the name informed above.
# If a container with the same name exists, an error is returned.
#
# The 3 options for the analytical_storage_ttl parameter are:
# 1) 0 or Null or not informed (Not enabled).
# 2) -1 (The data will be stored in analytical store infinitely).
# 3) Any other number is the actual ttl, in seconds.

try:
    container = db.create_container(
        id=CONTAINER,
        partition_key=PartitionKey(path='/id', kind='Hash'),analytical_storage_ttl=-1
    )
    properties = container.read()
    print('Container with id \'{0}\' created'.format(container.id))
    print('Partition Key - \'{0}\''.format(properties['partitionKey']))

except exceptions.CosmosResourceExistsError:
    print('A container with already exists')
```

Update the analytical store time to live

After the analytical store is enabled with a particular TTL value, you can update it to a different valid value later. You can update the value by using the Azure portal or SDKs. For information on the various Analytical TTL config options, see the analytical TTL supported values article.

Azure portal

If you created an analytical store enabled container through the Azure portal, it contains a default analytical TTL of -1. Use the following steps to update this value:

1. Sign in to the Azure portal or the Azure Cosmos DB Explorer.
2. Navigate to your Azure Cosmos DB account and open the **Data Explorer** tab.
3. Select an existing container that has analytical store enabled. Expand it and modify the following values:

- Open the **Scale & Settings** window.
- Under **Setting** find,** Analytical Storage Time to Live**.
- Select **On (no default)** or select **On** and set a TTL value
- Click **Save** to save the changes.

.NET SDK

The following code shows how to update the TTL for analytical store by using the .NET SDK:

```
// Get the container, update AnalyticalStorageTimeToLiveInSeconds
ContainerResponse containerResponse = await client.GetContainer("database", "container").ReadContainerAsync();
// Update analytical store TTL
containerResponse.Resource. AnalyticalStorageTimeToLiveInSeconds = 60 * 60 * 24 * 180  // Expire analytical
store data in 6 months;
await client.GetContainer("database", "container").ReplaceContainerAsync(containerResponse.Resource);
```

Java V4 SDK

The following code shows how to update the TTL for analytical store by using the Java V4 SDK:

```
CosmosContainerProperties containerProperties = new CosmosContainerProperties("myContainer",
"/myPartitionKey");

// Update analytical store TTL to expire analytical store data in 6 months;
containerProperties.setAnalyticalStoreTimeToLiveInSeconds (60 * 60 * 24 * 180 );

// Update container settings
container.replace(containerProperties).block();
```

Connect to a Synapse workspace

Use the instructions in Connect to Azure Synapse Link on how to access an Azure Cosmos DB database from Azure Synapse Analytics Studio with Azure Synapse Link.

Query analytical store using Apache Spark for Azure Synapse Analytics

Use the instructions in the Query Azure Cosmos DB analytical store article on how to query with Synapse Spark. That article gives some examples on how you can interact with the analytical store from Synapse gestures. Those gestures are visible when you right-click on a container. With gestures, you can quickly generate code and tweak it to your needs. They are also perfect for discovering data with a single click.

Query the analytical store using serverless SQL pool in Azure Synapse Analytics

Serverless SQL pool allows you to query and analyze data in your Azure Cosmos DB containers that are enabled with Azure Synapse Link. You can analyze data in near real-time without impacting the performance of your transactional workloads. It offers a familiar T-SQL syntax to query data from the analytical store and integrated connectivity to a wide range of BI and ad-hoc querying tools via the T-SQL interface. To learn more, see the Query analytical store using serverless SQL pool article.

Use serverless SQL pool to analyze and visualize data in Power BI

You can build a serverless SQL pool database and views over Synapse Link for Azure Cosmos DB. Later you can query the Azure Cosmos containers and then build a model with Power BI over those views to reflect that query. To learn more, see how to use Serverless SQL pool to analyze Azure Cosmos DB data with Synapse Link article.

Azure Resource Manager template

The Azure Resource Manager template creates a Synapse Link enabled Azure Cosmos DB account for SQL API. This template creates a Core (SQL) API account in one region with a container configured with analytical TTL enabled, and an option to use manual or autoscale throughput. To deploy this template, click on **Deploy to Azure** on the readme page.

Getting started with Azure Synpase Link - Samples

You can find samples to get started with Azure Synapse Link on GitHub. These showcase end-to-end solutions with IoT and retail scenarios. You can also find the samples corresponding to Azure Cosmos DB API for MongoDB in the same repo under the MongoDB folder.

Next steps

To learn more, see the following docs:

- Azure Synapse Link for Azure Cosmos DB.
- Azure Cosmos DB analytical store overview.
- Frequently asked questions about Synapse Link for Azure Cosmos DB.
- Apache Spark in Azure Synapse Analytics.
- Serverless SQL pool runtime support in Azure Synapse Analytics.

Connect to Azure Synapse Link for Azure Cosmos DB

12/4/2020 • 2 minutes to read • Edit Online

This article describes how to access an Azure Cosmos DB database from Azure Synapse Analytics Studio with Azure Synapse Link.

Prerequisites

Before you connect an Azure Cosmos DB database to your workspace, you'll need an:

- Existing Azure Cosmos DB database, or create a new account by following the steps in Quickstart: Manage an Azure Cosmos DB account.
- Existing Azure Synapse workspace, or create a new workspace by following the steps in Quickstart: Create a Synapse workspace.

> **IMPORTANT**
>
> Azure Synapse Link for Azure Cosmos DB is currently supported for workspaces that don't have a managed virtual network enabled.

Enable Synapse Link on an Azure Cosmos DB database account

To run large-scale analytics into Azure Cosmos DB without affecting your operational performance, we recommend enabling Synapse Link for Azure Cosmos DB. Synapse Link brings HTAP capability to a container and built-in support in Azure Synapse.

Go to Synapse Studio

From your Azure Synapse workspace, select **Launch Synapse Studio**. On the Synapse Studio home page, select **Data**, which takes you to the Data Object Explorer.

Connect an Azure Cosmos DB database to an Azure Synapse workspace

Connecting an Azure Cosmos DB database is done as a linked service. With an Azure Cosmos DB linked service, you can browse and explore data, read, and write from Apache Spark for Azure Synapse Analytics or SQL into Azure Cosmos DB.

From the Data Object Explorer, you can directly connect to an Azure Cosmos DB database by following these steps:

1. Select the + icon near **Data**.
2. Select **Connect to external data**.
3. Select the API that you want to connect to, for example, **SQL API** or **API for MongoDB**.
4. Select **Continue**.
5. Use a friendly name to name the linked service. The name will appear in the Data Object Explorer and is used by Azure Synapse runtimes to connect to the database and containers.
6. Select the **Azure Cosmos DB account name** and the **database name**.
7. (Optional) If no region is specified, Azure Synapse runtime operations will be routed toward the nearest region where the analytical store is enabled. You can also manually set the region you want your users to use to access

the Azure Cosmos DB analytical store. Select **Additional connection properties**, and then select **New**. Under **Property Name**, enter **PreferredRegions**. Set the **Value** to the region you want, for example, **WestUS2**. (There are no spaces between the words and the number.)

8. Select **Create**.

Azure Cosmos DB databases appear on the **Linked** tab under the **Azure Cosmos DB** section. With Azure Cosmos DB, you can differentiate an HTAP-enabled container from an OLTP-only container through the following icons:

OLTP-only container:

HTAP-enabled container:

Quickly interact with code-generated actions

By right-clicking into a container, you have a list of gestures that will trigger a Spark or SQL runtime. Writing into a container will happen through the Transactional Store of Azure Cosmos DB and will consume Request Units.

Next steps

- Learn what is supported between Azure Synapse and Azure Cosmos DB
- Learn how to query the analytical store with Spark

Interact with Azure Cosmos DB using Apache Spark in Azure Synapse Link

12/4/2020 • 7 minutes to read • Edit Online

In this article, you'll learn how to interact with Azure Cosmos DB using Synapse Apache Spark. With its full support for Scala, Python, SparkSQL, and C#, Synapse Apache Spark is central to analytics, data engineering, data science, and data exploration scenarios in Azure Synapse Link for Azure Cosmos DB.

The following capabilities are supported while interacting with Azure Cosmos DB:

- Synapse Apache Spark allows you to analyze data in your Azure Cosmos DB containers that are enabled with Azure Synapse Link in near real-time without impacting the performance of your transactional workloads. The following two options are available to query the Azure Cosmos DB analytical store from Spark:
  - Load to Spark DataFrame
  - Create Spark table
- Synapse Apache Spark also allows you to ingest data into Azure Cosmos DB. It is important to note that data is always ingested into Azure Cosmos DB containers through the transactional store. When Synapse Link is enabled, any new inserts, updates, and deletes are then automatically synced to the analytical store.
- Synapse Apache Spark also supports Spark structured streaming with Azure Cosmos DB as a source as well as a sink.

The following sections walk you through the syntax of above capabilities. Gestures in Azure Synapse Analytics workspace are designed to provide an easy out-of-the-box experience to get started. Gestures are visible when you right-click on an Azure Cosmos DB container in the **Data** tab of the Synapse workspace. With gestures, you can quickly generate code and tailor it to your needs. Gestures are also perfect for discovering data with a single click.

Query Azure Cosmos DB analytical store

Before you learn about the two possible options to query Azure Cosmos DB analytical store, loading to Spark DataFrame and creating Spark table, it is worth exploring the differences in experience so you can choose the option that works for your needs.

The difference in experience is around whether underlying data changes in the Azure Cosmos DB container should be automatically reflected in the analysis performed in Spark. When either a Spark DataFrame is registered or a Spark table is created against a container's analytical store, metadata around the current snapshot of data in the analytical store is fetched to Spark for efficient pushdown of subsequent analysis. It is important to note that since Spark follows a lazy evaluation policy, unless an action is invoked on the Spark DataFrame or a SparkSQL query is executed against the Spark table, actual data is not fetched from the underlying container's analytical store.

In the case of **loading to Spark DataFrame**, the fetched metadata is cached through the lifetime of the Spark session and hence subsequent actions invoked on the DataFrame are evaluated against the snapshot of the analytical store at the time of DataFrame creation.

On the other hand, in the case of **creating a Spark table**, the metadata of the analytical store state is not cached in Spark and is reloaded on every SparkSQL query execution against the Spark table.

Thus, you can choose between loading to Spark DataFrame and creating a Spark table based on whether you want your Spark analysis to be evaluated against a fixed snapshot of the analytical store or against the latest snapshot of the analytical store respectively.

> **NOTE**
> To query the Azure Cosmos DB API of Mongo DB accounts, learn more about the full fidelity schema representation in the analytical store and the extended property names to be used.

Load to Spark DataFrame

In this example, you'll create a Spark DataFrame that points to the Azure Cosmos DB analytical store. You can then perform additional analysis by invoking Spark actions against the DataFrame. This operation doesn't impact the transactional store.

The syntax in **Python** would be the following:

```
# To select a preferred list of regions in a multi-region Azure Cosmos DB account, add
.option("spark.cosmos.preferredRegions", "<Region1>,<Region2>")

df = spark.read.format("cosmos.olap")\
    .option("spark.synapse.linkedService", "<enter linked service name>")\
    .option("spark.cosmos.container", "<enter container name>")\
    .load()
```

The equivalent syntax in **Scala** would be the following:

```
// To select a preferred list of regions in a multi-region Azure Cosmos DB account, add
option("spark.cosmos.preferredRegions", "<Region1>,<Region2>")

val df_olap = spark.read.format("cosmos.olap").
    option("spark.synapse.linkedService", "<enter linked service name>").
    option("spark.cosmos.container", "<enter container name>").
    load()
```

Create Spark table

In this example, you'll create a Spark table that points the Azure Cosmos DB analytical store. You can then perform additional analysis by invoking SparkSQL queries against the table. This operation neither impacts the transactional store nor does it incur any data movement. If you decide to delete this Spark table, the underlying Azure Cosmos DB container and the corresponding analytical store will not be affected.

This scenario is convenient to reuse Spark tables through third-party tools and provide accessibility to the underlying data for the run-time.

The syntax to create a Spark table is as follows:

```
%%sql
-- To select a preferred list of regions in a multi-region Azure Cosmos DB account, add
spark.cosmos.preferredRegions '<Region1>,<Region2>' in the config options

create table call_center using cosmos.olap options (
    spark.synapse.linkedService '<enter linked service name>',
    spark.cosmos.container '<enter container name>'
)
```

> **NOTE**
> If you have scenarios where the schema of the underlying Azure Cosmos DB container changes over time; and if you want the updated schema to automatically reflect in the queries against the Spark table, you can achieve this by setting the `spark.cosmos.autoSchemaMerge` option to `true` in the Spark table options.

Write Spark DataFrame to Azure Cosmos DB container

In this example, you'll write a Spark DataFrame into an Azure Cosmos DB container. This operation will impact the performance of transactional workloads and consume request units provisioned on the Azure Cosmos DB container or the shared database.

The syntax in **Python** would be the following:

```
# Write a Spark DataFrame into an Azure Cosmos DB container
# To select a preferred list of regions in a multi-region Azure Cosmos DB account, add
.option("spark.cosmos.preferredRegions", "<Region1>,<Region2>")

YOURDATAFRAME.write.format("cosmos.oltp")\
    .option("spark.synapse.linkedService", "<enter linked service name>")\
    .option("spark.cosmos.container", "<enter container name>")\
    .option("spark.cosmos.write.upsertEnabled", "true")\
    .mode('append')\
    .save()
```

The equivalent syntax in **Scala** would be the following:

```
// To select a preferred list of regions in a multi-region Azure Cosmos DB account, add
option("spark.cosmos.preferredRegions", "<Region1>,<Region2>")

import org.apache.spark.sql.SaveMode

df.write.format("cosmos.oltp").
    option("spark.synapse.linkedService", "<enter linked service name>").
    option("spark.cosmos.container", "<enter container name>").
    option("spark.cosmos.write.upsertEnabled", "true").
    mode(SaveMode.Overwrite).
    save()
```

Load streaming DataFrame from container

In this gesture, you'll use Spark Streaming capability to load data from a container into a dataframe. The data will be stored in the primary data lake account (and file system) you connected to the workspace.

> **NOTE**
>
> If you are looking to reference external libraries in Synapse Apache Spark, learn more here. For instance, if you are looking to ingest a Spark DataFrame to a container of Cosmos DB API for Mongo DB, you can leverage the Mongo DB connector for Spark here.

Load streaming DataFrame from Azure Cosmos DB container

In this example, you'll use Spark's structured streaming capability to load data from an Azure Cosmos DB container into a Spark streaming DataFrame using the change feed functionality in Azure Cosmos DB. The checkpoint data used by Spark will be stored in the primary data lake account (and file system) that you connected to the workspace.

If the folder */localReadCheckpointFolder* isn't created (in the example below), it will be automatically created. This operation will impact the performance of transactional workloads and consume Request Units provisioned on the Azure Cosmos DB container or shared database.

The syntax in **Python** would be the following:

```
# To select a preferred list of regions in a multi-region Azure Cosmos DB account, add
.option("spark.cosmos.preferredRegions", "<Region1>,<Region2>")

dfStream = spark.readStream\
    .format("cosmos.oltp")\
    .option("spark.synapse.linkedService", "<enter linked service name>")\
    .option("spark.cosmos.container", "<enter container name>")\
    .option("spark.cosmos.changeFeed.readEnabled", "true")\
    .option("spark.cosmos.changeFeed.startFromTheBeginning", "true")\
    .option("spark.cosmos.changeFeed.checkpointLocation", "/localReadCheckpointFolder")\
    .option("spark.cosmos.changeFeed.queryName", "streamQuery")\
    .load()
```

The equivalent syntax in **Scala** would be the following:

```
// To select a preferred list of regions in a multi-region Azure Cosmos DB account, add
.option("spark.cosmos.preferredRegions", "<Region1>,<Region2>")

val dfStream = spark.readStream.
    format("cosmos.oltp").
    option("spark.synapse.linkedService", "<enter linked service name>").
    option("spark.cosmos.container", "<enter container name>").
    option("spark.cosmos.changeFeed.readEnabled", "true").
    option("spark.cosmos.changeFeed.startFromTheBeginning", "true").
    option("spark.cosmos.changeFeed.checkpointLocation", "/localReadCheckpointFolder").
    option("spark.cosmos.changeFeed.queryName", "streamQuery").
    load()
```

Write streaming DataFrame to Azure Cosmos DB container

In this example, you'll write a streaming DataFrame into an Azure Cosmos DB container. This operation will impact the performance of transactional workloads and consume Request Units provisioned on the Azure Cosmos DB container or shared database. If the folder */localWriteCheckpointFolder* isn't created (in the example below), it will be automatically created.

The syntax in **Python** would be the following:

```
# To select a preferred list of regions in a multi-region Azure Cosmos DB account, add
.option("spark.cosmos.preferredRegions", "<Region1>,<Region2>")

streamQuery = dfStream\
        .writeStream\
        .format("cosmos.oltp")\
        .outputMode("append")\
        .option("checkpointLocation", "/localWriteCheckpointFolder")\
        .option("spark.synapse.linkedService", "<enter linked service name>")\
        .option("spark.cosmos.container", "<enter container name>")\
        .option("spark.cosmos.connection.mode", "gateway")\
        .start()

streamQuery.awaitTermination()
```

The equivalent syntax in **Scala** would be the following:

```
// To select a preferred list of regions in a multi-region Azure Cosmos DB account, add
.option("spark.cosmos.preferredRegions", "<Region1>,<Region2>")

val query = dfStream.
            writeStream.
            format("cosmos.oltp").
            outputMode("append").
            option("checkpointLocation", "/localWriteCheckpointFolder").
            option("spark.synapse.linkedService", "<enter linked service name>").
            option("spark.cosmos.container", "<enter container name>").
            option("spark.cosmos.connection.mode", "gateway").
            start()

query.awaitTermination()
```

External library management

In this example, you'll learn how to reference external libraries from JAR files when using Spark notebooks in Synpase Apache Spark workspaces. You can place the JAR files in a container in the primary data lake account that you connected to the workspace and then add the following `%configure` statement in your Spark notebook:

```
%%configure -f
{
    "jars": [
        "abfss://<storage container name>@<data lake account name>.dfs.core.windows.net/<path to jar>"
    ]
}
```

If you are looking to submit remote Spark job definitions to a serverless Apache Spark pool, you can learn how to reference external libraries by following this tutorial.

Next steps

- Samples to get started with Azure Synapse Link on GitHub
- Learn what is supported in Azure Synapse Link for Azure Cosmos DB
- Connect to Synapse Link for Azure Cosmos DB

Query Azure Cosmos DB data with a serverless SQL pool in Azure Synapse Link Preview

12/4/2020 • 16 minutes to read • Edit Online

> **IMPORTANT**
> Serverless SQL pool support for Azure Synapse Link for Azure Cosmos DB is currently in preview. This preview version is provided without a service level agreement, and it's not recommended for production workloads. For more information, see Supplemental terms of use for Microsoft Azure previews.

A serverless SQL pool allows you to analyze data in your Azure Cosmos DB containers that are enabled with Azure Synapse Link in near real time without affecting the performance of your transactional workloads. It offers a familiar T-SQL syntax to query data from the analytical store and integrated connectivity to a wide range of business intelligence (BI) and ad-hoc querying tools via the T-SQL interface.

For querying Azure Cosmos DB, the full SELECT surface area is supported through the OPENROWSET function, which includes the majority of SQL functions and operators. You can also store results of the query that reads data from Azure Cosmos DB along with data in Azure Blob Storage or Azure Data Lake Storage by using create external table as select (CETAS). You can't currently store serverless SQL pool query results to Azure Cosmos DB by using CETAS.

In this article, you'll learn how to write a query with a serverless SQL pool that will query data from Azure Cosmos DB containers that are enabled with Azure Synapse Link. You can then learn more about building serverless SQL pool views over Azure Cosmos DB containers and connecting them to Power BI models in this tutorial.

> **IMPORTANT**
> This tutorial uses a container with an Azure Cosmos DB well-defined schema. The query experience that the serverless SQL pool provides for an Azure Cosmos DB full fidelity schema is temporary behavior that will change based on the preview feedback. Don't rely on the result set schema of the `OPENROWSET` function without the `WITH` clause that reads data from a container with a full fidelity schema because the query experience might be aligned with and change based on the well-defined schema. You can post your feedback in the Azure Synapse Analytics feedback forum. You can also contact the Azure Synapse Link product team to provide feedback.

Overview

Serverless SQL pool enables you to query Azure Cosmos DB analytical storage using `OPENROWSET` function.

- `OPENROWSET` with inline key. This syntax can be used to query Azure Cosmos DB collections without need to prepare credentials.
- `OPENROWSET` that referenced credential that contains the Cosmos DB account key. This syntax can be used to create views on Azure Cosmos DB collections.

- OPENROWSET with key
- OPENROWSET with credential

To support querying and analyzing data in an Azure Cosmos DB analytical store, a serverless SQL pool uses the following `OPENROWSET` syntax:

```
OPENROWSET(
       'CosmosDB',
       '<Azure Cosmos DB connection string>',
       <Container name>
    )  [ < with clause > ] AS alias
```

The Azure Cosmos DB connection string specifies the Azure Cosmos DB account name, database name, database account master key, and an optional region name to the `OPENROWSET` function.

The connection string has the following format:

```
'account=<database account name>;database=<database name>;region=<region name>;key=<database account master key>'
```

The Azure Cosmos DB container name is specified without quotation marks in the `OPENROWSET` syntax. If the container name has any special characters, for example, a dash (-), the name should be wrapped within square brackets (`[]`) in the `OPENROWSET` syntax.

> **IMPORTANT**
> Make sure that you're using some UTF-8 database collation, for example, `Latin1_General_100_CI_AS_SC_UTF8`, because string values in an Azure Cosmos DB analytical store are encoded as UTF-8 text. A mismatch between text encoding in the file and collation might cause unexpected text conversion errors. You can easily change default collation of the current database by using the T-SQL statement `alter database current collate Latin1_General_100_CI_AI_SC_UTF8`.

> **NOTE**
> A serverless SQL pool doesn't support querying an Azure Cosmos DB transactional store.

Sample dataset

The examples in this article are based on data from the European Centre for Disease Prevention and Control (ECDC) COVID-19 Cases and COVID-19 Open Research Dataset (CORD-19), doi:10.5281/zenodo.3715505.

You can see the license and the structure of data on these pages. You can also download sample data for the ECDC and CORD-19 datasets.

To follow along with this article showcasing how to query Azure Cosmos DB data with a serverless SQL pool, make sure that you create the following resources:

- An Azure Cosmos DB database account that's Azure Synapse Link enabled.
- An Azure Cosmos DB database named `covid`.
- Two Azure Cosmos DB containers named `EcdcCases` and `Cord19` loaded with the preceding sample datasets.

Explore Azure Cosmos DB data with automatic schema inference

The easiest way to explore data in Azure Cosmos DB is by using the automatic schema inference capability. By omitting the `WITH` clause from the `OPENROWSET` statement, you can instruct the serverless SQL pool to autodetect (infer) the schema of the analytical store of the Azure Cosmos DB container.

- OPENROWSET with key
- OPENROWSET with credential

```
SELECT TOP 10 *
FROM OPENROWSET(
       'CosmosDB',
       'account=MyCosmosDbAccount;database=covid;region=westus2;key=C0Sm0sDbKey==',
       EcdcCases) as documents
```

In the preceding example, we instructed the serverless SQL pool to connect to the `covid` database in the Azure Cosmos DB account `MyCosmosDbAccount` authenticated by using the Azure Cosmos DB key (the dummy in the preceding example). We then accessed the `EcdcCases` container's analytical store in the `West US 2` region. Since there's no projection of specific properties, the `OPENROWSET` function will return all properties from the Azure Cosmos DB items.

Assuming that the items in the Azure Cosmos DB container have `date_rep`, `cases`, and `geo_id` properties, the results of this query are shown in the following table:

DATE_REP	CASES	GEO_ID
2020-08-13	254	RS
2020-08-12	235	RS
2020-08-11	163	RS

If you need to explore data from the other container in the same Azure Cosmos DB database, you can use the same connection string and reference the required container as the third parameter:

```
SELECT TOP 10 *
FROM OPENROWSET(
       'CosmosDB',
       'account=MyCosmosDbAccount;database=covid;region=westus2;key=C0Sm0sDbKey==',
       Cord19) as cord19
```

Explicitly specify schema

While automatic schema inference capability in `OPENROWSET` provides a simple, easy-to-use querience, your business scenarios might require you to explicitly specify the schema to read-only relevant properties from the Azure Cosmos DB data.

The `OPENROWSET` function enables you to explicitly specify what properties you want to read from the data in the container and to specify their data types.

Let's imagine that we've imported some data from the ECDC COVID dataset with the following structure into Azure Cosmos DB:

```
{"date_rep":"2020-08-13","cases":254,"countries_and_territories":"Serbia","geo_id":"RS"}
{"date_rep":"2020-08-12","cases":235,"countries_and_territories":"Serbia","geo_id":"RS"}
{"date_rep":"2020-08-11","cases":163,"countries_and_territories":"Serbia","geo_id":"RS"}
```

These flat JSON documents in Azure Cosmos DB can be represented as a set of rows and columns in Synapse SQL. The `OPENROWSET` function enables you to specify a subset of properties that you want to read and the exact column types in the `WITH` clause:

- OPENROWSET with key
- OPENROWSET with credential

```
SELECT TOP 10 *
FROM OPENROWSET(
      'CosmosDB',
      'account=MyCosmosDbAccount;database=covid;region=westus2;key=C0Sm0sDbKey==',
       EcdcCases
    ) with ( date_rep varchar(20), cases bigint, geo_id varchar(6) ) as rows
```

The result of this query might look like the following table:

DATE_REP	CASES	GEO_ID
2020-08-13	254	RS
2020-08-12	235	RS

DATE_REP	CASES	GEO_ID
2020-08-11	163	RS

For more information about the SQL types that should be used for Azure Cosmos DB values, see the rules for SQL type mappings at the end of the article.

Create view

Once you identify the schema, you can prepare a view on top of your Azure Cosmos DB data. You should place your Azure Cosmos DB account key in a separate credential and reference this credential from `OPENROWSET` function. Do not keep your account key in the view definition.

```
CREATE CREDENTIAL MyCosmosDbAccountCredential
WITH IDENTITY = 'SHARED ACCESS SIGNATURE', SECRET = 'C0Sm0sDbKey==';
GO
CREATE OR ALTER VIEW EcdcCases
AS SELECT *
FROM OPENROWSET(
      PROVIDER = 'CosmosDB',
      CONNECTION = 'account=MyCosmosDbAccount;database=covid;region=westus2',
      OBJECT = 'EcdcCases',
      SERVER_CREDENTIAL = 'MyCosmosDbAccountCredential'
    ) with ( date_rep varchar(20), cases bigint, geo_id varchar(6) ) as rows
```

Do not use `OPENROWSET` without explicitly defined schema because it might impact your performance. Make sure that you use the smallest possible sizes for your columns (for example VARCHAR(100) instead of default VARCHAR(8000)). You should use some UTF-8 collation as default database collation or set it as explicit column collation to avoid UTF-8 conversion issue. Collation `Latin1_General_100_BIN2_UTF8` provides best performance when yu filter data using some string columns.

Query nested objects and arrays

With Azure Cosmos DB, you can represent more complex data models by composing them as nested objects or arrays. The autosync capability of Azure Synapse Link for Azure Cosmos DB manages the schema representation in the analytical store out of the box, which includes handling nested data types that allow for rich querying from the serverless SQL pool.

For example, the CORD-19 dataset has JSON documents that follow this structure:

```
{
    "paper_id": <str>,                   # 40-character sha1 of the PDF
    "metadata": {
        "title": <str>,
        "authors": <array of objects>    # list of author dicts, in order
        ...
     }
     ...
}
```

The nested objects and arrays in Azure Cosmos DB are represented as JSON strings in the query result when the `OPENROWSET` function reads them. One option to read the values from these complex types as SQL columns is to use SQL JSON functions:

```
SELECT
    title = JSON_VALUE(metadata, '$.title'),
    authors = JSON_QUERY(metadata, '$.authors'),
    first_author_name = JSON_VALUE(metadata, '$.authors[0].first')
FROM
    OPENROWSET(
      'CosmosDB',
      'account=MyCosmosDbAccount;database=covid;region=westus2;key=C0Sm0sDbKey==',
       Cord19
    WITH ( metadata varchar(MAX) ) AS docs;
```

The result of this query might look like the following table:

TITLE	AUTHORS	FIRST_AUTOR_NAME
Supplementary Information An eco-epidemi...	`[{"first":"Julien","last":"Mélade","suffix":"Julien","affiliation":{"laboratory":"Centre de Recher…`	

As an alternative option, you can also specify the paths to nested values in the objects when you use the `WITH` clause:

```
SELECT
    *
FROM
    OPENROWSET(
      'CosmosDB',
      'account=MyCosmosDbAccount;database=covid;region=westus2;key=C0Sm0sDbKey==',
       Cord19
    WITH ( title varchar(1000) '$.metadata.title',
           authors varchar(max) '$.metadata.authors'
    ) AS docs;
```

Learn more about analyzing complex data types in Azure Synapse Link and nested structures in a serverless SQL pool.

> **IMPORTANT**
>
> If you see unexpected characters in your text like `MÃƒÂ©lade` instead of `Mélade`, then your database collation isn't set to UTF-8 collation. Change collation of the database to UTF-8 collation by using a SQL statement like `ALTER DATABASE MyLdw COLLATE LATIN1_GENERAL_100_CI_AS_SC_UTF8`.

Flatten nested arrays

Azure Cosmos DB data might have nested subarrays like the author's array from a CORD-19 dataset:

```
{
    "paper_id": <str>,                      # 40-character sha1 of the PDF
    "metadata": {
        "title": <str>,
        "authors": [                        # list of author dicts, in order
            {
                "first": <str>,
                "middle": <list of str>,
                "last": <str>,
                "suffix": <str>,
                "affiliation": <dict>,
                "email": <str>
            },
            ...
        ],
        ...
}
```

In some cases, you might need to "join" the properties from the top item (metadata) with all elements of the array

(authors). A serverless SQL pool enables you to flatten nested structures by applying the `OPENJSON` function on the nested array:

```
SELECT
    *
FROM
    OPENROWSET(
      'CosmosDB',
      'account=MyCosmosDbAccount;database=covid;region=westus2;key=C0Sm0sDbKey==',
       Cord19
    ) WITH ( title varchar(1000) '$.metadata.title',
             authors varchar(max) '$.metadata.authors' ) AS docs
      CROSS APPLY OPENJSON ( authors )
                  WITH (
                       first varchar(50),
                       last varchar(50),
                       affiliation nvarchar(max) as json
                  ) AS a
```

The result of this query might look like the following table:

TITLE	AUTHORS	FIRST	LAST	AFFILIATION
Supplementary Information An eco-epidemi...	`[{"first":"Julien","last":"Mélade","suffix":"","affiliation": {"laboratory":"Centre de Recher…`	Julien	Mélade	`{"laboratory":"Centre de Recher…`
Supplementary Information An eco-epidemi...	`[{"first":"Nicolas","last":"Ali","suffix":"","affiliation": {"laboratory":"","institution":"U…`	Nicolas	Ali	`{"laboratory":"","institutio`
Supplementary Information An eco-epidemi...	`[{"first":"Beza","last":"Ramazindrazana","suffix":"","affiliation": {"laboratory":"Centre de Recher…`	Beza	Ramazindrazana	`{"laboratory":"Centre de Recher…`
Supplementary Information An eco-epidemi...	`[{"first":"Olivier","last":"Flores","suffix":"","affiliation": {"laboratory":"UMR C53 CIRAD, …`	Olivier	Flores	`{"laboratory":"UMR C53 CIRAD, …`

IMPORTANT

If you see unexpected characters in your text like `MÃƒÂ©lade` instead of `Mélade`, then your database collation isn't set to UTF-8 collation. Change collation of the database to UTF-8 collation by using a SQL statement like `ALTER DATABASE MyLdw COLLATE LATIN1_GENERAL_100_CI_AS_SC_UTF8`.

Azure Cosmos DB to SQL type mappings

Although Azure Cosmos DB transactional store is schema-agnostic, the analytical store is schematized to optimize for analytical query performance. With the autosync capability of Azure Synapse Link, Azure Cosmos DB manages the schema representation in the analytical store out of the box, which includes handling nested data types. Since a serverless SQL pool queries the analytical store, it's important to understand how to map Azure Cosmos DB input data types to SQL data types.

Azure Cosmos DB accounts of SQL (Core) API support JSON property types of number, string, Boolean, null, nested object, or array. You would need to choose SQL types that match these JSON types if you're using the `WITH` clause in `OPENROWSET`. The following table shows the SQL column types that should be used for different property types in Azure Cosmos DB.

AZURE COSMOS DB PROPERTY TYPE	SQL COLUMN TYPE
Boolean	bit

AZURE COSMOS DB PROPERTY TYPE	SQL COLUMN TYPE
Integer	bigint
Decimal	float
String	varchar (UTF-8 database collation)
Date time (ISO-formatted string)	varchar(30)
Date time (UNIX timestamp)	bigint
Null	`any SQL type`
Nested object or array	varchar(max) (UTF-8 database collation), serialized as JSON text

Full fidelity schema

Azure Cosmos DB full fidelity schema records both values and their best match types for every property in a container. The `OPENROWSET` function on a container with full fidelity schema provides both the type and the actual value in each cell. Let's assume that the following query reads the items from a container with full fidelity schema:

```
SELECT *
FROM OPENROWSET(
      'CosmosDB',
      'account=MyCosmosDbAccount;database=covid;region=westus2;key=C0Sm0sDbKey==',
       EcdcCases
    ) as rows
```

The result of this query will return types and values formatted as JSON text:

DATE_REP	CASES	GEO_ID
{"date":"2020-08-13"}	{"int32":"254"}	{"string":"RS"}
{"date":"2020-08-12"}	{"int32":"235"}	{"string":"RS"}
{"date":"2020-08-11"}	{"int32":"316"}	{"string":"RS"}
{"date":"2020-08-10"}	{"int32":"281"}	{"string":"RS"}
{"date":"2020-08-09"}	{"int32":"295"}	{"string":"RS"}
{"string":"2020/08/08"}	{"int32":"312"}	{"string":"RS"}
{"date":"2020-08-07"}	{"float64":"339.0"}	{"string":"RS"}

For every value, you can see the type identified in an Azure Cosmos DB container item. Most of the values for the `date_rep` property contain `date` values, but some of them are incorrectly stored as strings in Azure Cosmos DB. Full fidelity schema will return both correctly typed `date` values and incorrectly formatted `string` values. The number of cases is information stored as an `int32` value, but there's one value that's entered as a decimal number. This value has the `float64` type. If there are some values that exceed the largest `int32` number, they would be stored as the `int64` type. All `geo_id` values in this example are stored as `string` types.

> **IMPORTANT**
> The `OPENROWSET` function without a `WITH` clause exposes both values with expected types and the values with incorrectly entered types. This function is designed for data exploration and not for reporting. Don't parse JSON values returned from this function to build reports. Use an explicit WITH clause to create your reports. You should clean up the values that have incorrect types in the Azure Cosmos DB container to apply corrections in the full fidelity analytical store.

If you need to query Azure Cosmos DB accounts of the Mongo DB API kind, you can learn more about the full fidelity schema representation in the analytical store and the extended property names to be used in What is Azure Cosmos DB Analytical Store (preview)?.

Query items with full fidelity schema

While querying full fidelity schema, you need to explicitly specify the SQL type and the expected Azure Cosmos DB property type in the `WITH` clause. Don't use `OPENROWSET` without a `WITH` clause in the reports because the format of the result set might be changed in the preview based on feedback.

In the following example, we'll assume that `string` is the correct type for the `geo_id` property and `int32` is the correct type for the `cases` property:

```
SELECT geo_id, cases = SUM(cases)
FROM OPENROWSET(
      'CosmosDB'
      'account=MyCosmosDbAccount;database=covid;region=westus2;key=C0Sm0sDbKey==',
       EcdcCases
    ) WITH ( geo_id VARCHAR(50) '$.geo_id.string',
             cases INT '$.cases.int32'
    ) as rows
GROUP BY geo_id
```

Values for `geo_id` and `cases` that have other types will be returned as `NULL` values. This query will reference only the `cases` with the specified type in the expression (`cases.int32`).

If you have values with other types (`cases.int64`, `cases.float64`) that can't be cleaned in an Azure Cosmos DB container, you would need to explicitly reference them in a `WITH` clause and combine the results. The following query aggregates both `int32`, `int64`, and `float64` stored in the `cases` column:

```
SELECT geo_id, cases = SUM(cases_int) + SUM(cases_bigint) + SUM(cases_float)
FROM OPENROWSET(
      'CosmosDB',
      'account=MyCosmosDbAccount;database=covid;region=westus2;key=C0Sm0sDbKey==',
       EcdcCases
    ) WITH ( geo_id VARCHAR(50) '$.geo_id.string',
             cases_int INT '$.cases.int32',
             cases_bigint BIGINT '$.cases.int64',
             cases_float FLOAT '$.cases.float64'
    ) as rows
GROUP BY geo_id
```

In this example, the number of cases is stored either as `int32`, `int64`, or `float64` values. All values must be extracted to calculate the number of cases per country.

Known issues

- The query experience that serverless SQL pool provides for Azure Cosmos DB full fidelity schema is temporary behavior that will be changed based on preview feedback. Don't rely on the schema that the `OPENROWSET` function without the `WITH` clause provides during public preview because the query experience might be aligned with well-defined schema based on customer feedback. To provide feedback, contact the Azure Synapse Link product team.
- A serverless SQL pool won't return a compile-time error if the `OPENROWSET` column collation doesn't have UTF-8 encoding. You can easily change default collation for all `OPENROWSET` functions running in the current database by using

the T-SQL statement `alter database current collate Latin1_General_100_CI_AI_SC_UTF8`.

Possible errors and troubleshooting actions are listed in the following table.

ERROR	ROOT CAUSE
Syntax errors: - Incorrect syntax near "Openrowset" - `...` is not a recognized BULK OPENROWSET provider option. - Incorrect syntax near `...`	Possible root causes: - Not using CosmosDB as the first parameter. - Using a string literal instead of an identifier in the third parameter. - Not specifying the third parameter (container name).
There was an error in the CosmosDB connection string.	- The account, database, or key isn't specified. - There's some option in a connection string that isn't recognized. - A semicolon (`;`) is placed at the end of a connection string.
Resolving CosmosDB path has failed with the error "Incorrect account name" or "Incorrect database name."	The specified account name, database name, or container can't be found, or analytical storage hasn't been enabled to the specified collection.
Resolving CosmosDB path has failed with the error "Incorrect secret value" or "Secret is null or empty."	The account key isn't valid or is missing.
Column `column name` of the type `type name` isn't compatible with the external data type `type name`.	The specified column type in the `WITH` clause doesn't match the type in the Azure Cosmos DB container. Try to change the column type as it's described in the section Azure Cosmos DB to SQL type mappings, or use the `VARCHAR` type.
Column contains `NULL` values in all cells.	Possibly a wrong column name or path expression in the `WITH` clause. The column name (or path expression after the column type) in the `WITH` clause must match some property name in the Azure Cosmos DB collection. Comparison is *case sensitive*. For example, `productCode` and `ProductCode` are different properties.

You can report suggestions and issues on the Azure Synapse Analytics feedback page.

Next steps

For more information, see the following articles:

- Use Power BI and serverless SQL pool with Azure Synapse Link
- Create and use views in a serverless SQL pool
- Tutorial on building serverless SQL pool views over Azure Cosmos DB and connecting them to Power BI models via DirectQuery

Use Power BI and serverless Synapse SQL pool (preview) to analyze Azure Cosmos DB data with Synapse Link

12/4/2020 • 4 minutes to read • Edit Online

APPLIES TO: SQL API Azure Cosmos DB API for MongoDB

In this article, you learn how to build a serverless SQL pool database and views over Synapse Link for Azure Cosmos DB. You will query the Azure Cosmos DB containers and then build a model with Power BI over those views to reflect that query.

In this scenario, you will use dummy data about Surface product sales in a partner retail store. You will analyze the revenue per store based on the proximity to large households and the impact of advertising for a specific week. In this article, you create two views named **RetailSales** and **StoreDemographics** and a query between them. You can get the sample product data from this GitHub repo.

> **IMPORTANT**
>
> Synapse serverless SQL pool support for Azure Synapse Link for Azure Cosmos DB is currently in preview. This preview version is provided without a service level agreement, and it's not recommended for production workloads. For more information, see Supplemental terms of use for Microsoft Azure previews.

Prerequisites

Make sure to create the following resources before you start:

- Create an Azure Cosmos DB account of kind SQL(core) or MongoDB.
- Enable Azure Synapse Link for your Azure Cosmos account
- Create a database within the Azure Cosmos account and two containers that have analytical store enabled.
- Load products data into the Azure Cosmos containers as described in this batch data ingestion notebook.
- Create a Synapse workspace named **SynapseLinkBI**.
- Connect the Azure Cosmos database to the Synapse workspace.

Create a database and views

From the Synapse workspace go the **Develop** tab, select the **+** icon, and select **SQL Script**.

Every workspace comes with a serverless SQL endpoint. After creating a SQL script, from the tool bar on the top connect to **Built-in**.

Create a new database, named **RetailCosmosDB**, and a SQL view over the Synapse Link enabled containers. The following command shows how to create a database:

```
--Createdatabase
CreatedatabaseRetailCosmosDB
```

Next, create multiple views across different Synapse Link enabled Azure Cosmos containers. Views will allow you to use T-SQL to join and query Azure Cosmos DB data sitting in different containers. Make sure to select the **RetailCosmosDB** database when creating the views.

The following scripts show how to create views on each container. For simplicity, let's use the automatic schema inference feature of serverless SQL pool over Synapse Link enabled containers:

RetailSales view:

```
--Create view for RetailSales container
CREATEVIEW RetailSales
AS
SELECT  *
FROM OPENROWSET (
    'CosmosDB', N'account=<Your Azure Cosmos account name>;database=<Your Azure Cosmos database name>;region=
<Your Azure Cosmos DB Region>;key=<Your Azure Cosmos DB key here>',RetailSales)
AS q1
```

Make sure to insert your Azure Cosmos DB region and the primary key in the previous SQL script. All the characters in the region name should be in lower case without spaces. Unlike the other parameters of the `OPENROWSET` command, the container name parameter should be specified without quotes around it.

StoreDemographics view:

```
--Create view for StoreDemographics container
CREATEVIEWStoreDemographics
AS
SELECT  *
FROM OPENROWSET (
    'CosmosDB', N'account=<Your Azure Cosmos account name>;database=<Your Azure Cosmos database name>;region=
<Your Azure Cosmos DB Region>;key=<Your Azure Cosmos DB key here>', StoreDemographics)
AS q1
```

Now run the SQL script by selecting the **Run** command.

Query the views

Now that the two views are created, let's define the query to join those two views as follows:

```
SELECT
sum(p.[revenue]) as revenue
,p.[advertising]
,p.[storeId]
,p.[weekStarting]
,q.[largeHH]
 FROM [dbo].[RetailSales] as p
INNER JOIN [dbo].[StoreDemographics] as q ON q.[storeId] = p.[storeId]
GROUP BY p.[advertising], p.[storeId], p.[weekStarting], q.[largeHH]
```

Select **Run** that gives the following table as result:

View | Table | Chart | Export results

Search

Revenue	Advertising	StoreId	WeekStarting	LargeHH
31502	0	100	1499904000000	0.2164
28973	0	100	1505347200000	0.2164
53282	0	100	1516838400000	0.2164

Model views over containers with Power BI

Next open the Power BI desktop and connect to the serverless SQL endpoint by using the following steps:

1. Open the Power BI Desktop application. Select **Get data** and select **more**.
2. Choose **Azure Synapse Analytics (SQL DW)** from the list of connection options.
3. Enter the name of the SQL endpoint where the database is located. Enter `SynapseLinkBI-ondemand.sql.azuresynapse.net` within the **Server** field. In this example, **SynapseLinkBI** is name of the workspace. Replace it if you have given a different name to your workspace. Select **Direct Query** for data connectivity mode and then **OK**.
4. Select the preferred authentication method such as Azure AD.
5. Select the **RetailCosmosDB** database and the **RetailSales**, **StoreDemographics** views.
6. Select **Load** to load the two views into the direct query mode.
7. Select **Model** to create a relationship between the two views through the **storeId** column.
8. Drag the **StoreId** column from the **RetailSales** view towards the **StoreId** column in the **StoreDemographics** view.
9. Select the Many to one (*:1) relationship because there are multiple rows with the same store ID in the

RetailSales view. **StoreDemographics** has only one store ID row (it is a dimension table).

Now navigate to the **report** window and create a report to compare the relative importance of household size to the average revenue per store based on the scattered representation of revenue and LargeHH index:

1. Select **Scatter chart**.
2. Drag and drop **LargeHH** from the **StoreDemographics** view into the X-axis.
3. Drag and drop **Revenue** from **RetailSales** view into the Y-axis. Select **Average** to get the average sales per product per store and per week.
4. Drag and drop the **productCode** from **RetailSales** view into the legend to select a specific product line. After you choose these options, you should see a graph like the following screenshot:

Next steps

Use T-SQL to query Azure Cosmos DB data using Azure Synapse Link

Use serverless SQL pool to analyze Azure Open Datasets and visualize the results in Azure Synapse Studio

Configure IP firewall in Azure Cosmos DB

11/2/2020 • 10 minutes to read • Edit Online

APPLIES TO: SQL API Cassandra API Gremlin API Table API Azure Cosmos DB API for MongoDB

To secure the data stored in your account, Azure Cosmos DB supports a secret based authorization model that utilizes a strong Hash-based Message Authentication Code (HMAC). Additionally, Azure Cosmos DB supports IP-based access controls for inbound firewall support. This model is similar to the firewall rules of a traditional database system and provides an additional level of security to your account. With firewalls, you can configure your Azure Cosmos account to be accessible only from an approved set of machines and/or cloud services. Access to data stored in your Azure Cosmos database from these approved sets of machines and services will still require the caller to present a valid authorization token.

IP access control

By default, your Azure Cosmos account is accessible from internet, as long as the request is accompanied by a valid authorization token. To configure IP policy-based access control, the user must provide the set of IP addresses or IP address ranges in CIDR (Classless Inter-Domain Routing) form to be included as the allowed list of client IPs to access a given Azure Cosmos account. Once this configuration is applied, any requests originating from machines outside this allowed list receive 403 (Forbidden) response. When using IP firewall, it is recommended to allow Azure portal to access your account. Access is required to allow use of data explorer as well as to retrieve metrics for your account that show up on the Azure portal. When using data explorer, in addition to allowing Azure portal to access your account, you also need to update your firewall settings to add your current IP address to the firewall rules. Note that firewall changes may take up to 15min to propagate.

You can combine IP-based firewall with subnet and VNET access control. By combining them, you can limit access to any source that has a public IP and/or from a specific subnet within VNET. To learn more about using subnet and VNET-based access control see Access Azure Cosmos DB resources from virtual networks.

To summarize, authorization token is always required to access an Azure Cosmos account. If IP firewall and VNET Access Control List (ACLs) are not set up, the Azure Cosmos account can be accessed with the authorization token. After the IP firewall or VNET ACLs or both are set up on the Azure Cosmos account, only requests originating from the sources you have specified (and with the authorization token) get valid responses.

You can secure the data stored in your Azure Cosmos DB account by using IP firewalls. Azure Cosmos DB supports IP-based access controls for inbound firewall support. You can set an IP firewall on the Azure Cosmos DB account by using one of the following ways:

- From the Azure portal
- Declaratively by using an Azure Resource Manager template
- Programmatically through the Azure CLI or Azure PowerShell by updating the **ipRangeFilter** property

Configure an IP firewall by using the Azure portal

To set the IP access control policy in the Azure portal, go to the Azure Cosmos DB account page and select **Firewall and virtual networks** on the navigation menu. Change the **Allow access from** value to **Selected networks**, and then select **Save**.

When IP access control is turned on, the Azure portal provides the ability to specify IP addresses, IP address ranges, and switches. Switches enable access to other Azure services and the Azure portal. The following sections give details about these switches.

> **NOTE**
>
> After you enable an IP access control policy for your Azure Cosmos DB account, all requests to your Azure Cosmos DB account from machines outside the allowed list of IP address ranges are rejected. Browsing the Azure Cosmos DB resources from the portal is also blocked to ensure the integrity of access control.

Allow requests from the Azure portal

When you enable an IP access control policy programmatically, you need to add the IP address for the Azure portal to the **ipRangeFilter** property to maintain access. The portal IP addresses are:

REGION	IP ADDRESS
Germany	51.4.229.218
China	139.217.8.252
US Gov	52.244.48.71
All other regions	104.42.195.92,40.76.54.131,52.176.6.30,52.169.50.45,52.187.184.26

You can enable requests to access the Azure portal by selecting the **Allow access from Azure portal** option, as shown in the following screenshot:

Allow requests from global Azure datacenters or other sources within Azure

If you access your Azure Cosmos DB account from services that don't provide a static IP (for example, Azure Stream Analytics and Azure Functions), you can still use the IP firewall to limit access. You can enable access from other sources within the Azure by selecting the **Accept connections from within Azure datacenters** option, as shown in the following screenshot:

When you enable this option, the IP address `0.0.0.0` is added to the list of allowed IP addresses. The `0.0.0.0` IP address restricts requests to your Azure Cosmos DB account from Azure datacenter IP range. This setting does not allow access for any other IP ranges to your Azure Cosmos DB account.

> **NOTE**
>
> This option configures the firewall to allow all requests from Azure, including requests from the subscriptions of other customers deployed in Azure. The list of IPs allowed by this option is wide, so it limits the effectiveness of a firewall policy. Use this option only if your requests don't originate from static IPs or subnets in virtual networks. Choosing this option automatically allows access from the Azure portal because the Azure portal is deployed in Azure.

Requests from your current IP

To simplify development, the Azure portal helps you identify and add the IP of your client machine to the allowed list. Apps running your machine can then access your Azure Cosmos DB account.

The portal automatically detects the client IP address. It might be the client IP address of your machine, or the IP address of your network gateway. Make sure to remove this IP address before you take your workloads to production.

To add your current IP to the list of IPs, select **Add my current IP**. Then select **Save**.

Requests from cloud services

In Azure, cloud services are a common way for hosting middle-tier service logic by using Azure Cosmos DB. To enable access to your Azure Cosmos DB account from a cloud service, you must add the public IP address of the cloud service to the allowed list of IP addresses associated with your Azure Cosmos DB account by configuring the IP access control policy. This ensures that all role instances of cloud services have access to your Azure Cosmos DB account.

You can retrieve IP addresses for your cloud services in the Azure portal, as shown in the following screenshot:

When you scale out your cloud service by adding role instances, those new instances will automatically have access to the Azure Cosmos DB account because they're part of the same cloud service.

Requests from virtual machines

You can also use virtual machines or virtual machine scale sets to host middle-tier services by using Azure Cosmos DB. To configure your Cosmos DB account such that it allows access from virtual machines, you must configure the public IP address of the virtual machine and/or virtual machine scale set as one of the allowed IP addresses for your Azure Cosmos DB account by configuring the IP access control policy.

You can retrieve IP addresses for virtual machines in the Azure portal, as shown in the following screenshot:

When you add virtual machine instances to the group, they automatically receive access to your Azure Cosmos DB account.

Requests from the internet

When you access your Azure Cosmos DB account from a computer on the internet, the client IP address or IP address range of the machine must be added to the allowed list of IP addresses for your account.

Configure an IP firewall by using a Resource Manager template

To configure access control to your Azure Cosmos DB account, make sure that the Resource Manager template specifies the **ipRules** property with an array of allowed IP ranges. If configuring IP Firewall to an already deployed Cosmos account, ensure the `locations` array matches what is currently deployed. You cannot simultaneously modify the `locations` array and other properties. For more information and samples of Azure Resource Manager templates for Azure Cosmos DB see, Azure Resource Manager templates for Azure Cosmos DB

> **IMPORTANT**
>
> The **ipRules** property has been introduced with API version 2020-04-01. Previous versions exposed an **ipRangeFilter** property instead, which is a list of comma-separated IP addresses.

The example below shows how the **ipRules** property is exposed in API version 2020-04-01 or later:

```
{
  "type": "Microsoft.DocumentDB/databaseAccounts",
  "name": "[variables('accountName')]",
  "apiVersion": "2020-04-01",
  "location": "[parameters('location')]",
  "kind": "GlobalDocumentDB",
  "properties": {
    "consistencyPolicy": "[variables('consistencyPolicy')[parameters('defaultConsistencyLevel')]]",
    "locations": "[variables('locations')]",
    "databaseAccountOfferType": "Standard",
    "enableAutomaticFailover": "[parameters('automaticFailover')]",
    "ipRules": [
      {
        "ipAddressOrRange": "40.76.54.131"
      },
      {
        "ipAddressOrRange": "52.176.6.30"
      },
      {
        "ipAddressOrRange": "52.169.50.45"
      },
      {
        "ipAddressOrRange": "52.187.184.26"
      }
    ]
  }
}
```

Here's the same example for any API version prior to 2020-04-01:

```
{
  "type": "Microsoft.DocumentDB/databaseAccounts",
  "name": "[variables('accountName')]",
  "apiVersion": "2019-08-01",
  "location": "[parameters('location')]",
  "kind": "GlobalDocumentDB",
  "properties": {
    "consistencyPolicy": "[variables('consistencyPolicy')[parameters('defaultConsistencyLevel')]]",
    "locations": "[variables('locations')]",
    "databaseAccountOfferType": "Standard",
    "enableAutomaticFailover": "[parameters('automaticFailover')]",
    "ipRangeFilter":"40.76.54.131,52.176.6.30,52.169.50.45,52.187.184.26"
  }
}
```

Configure an IP access control policy by using the Azure CLI

The following command shows how to create an Azure Cosmos DB account with IP access control:

```
# Create a Cosmos DB account with default values and IP Firewall enabled
resourceGroupName='MyResourceGroup'
accountName='mycosmosaccount'
ipRangeFilter='192.168.221.17,183.240.196.255,40.76.54.131'

# Make sure there are no spaces in the comma-delimited list of IP addresses or CIDR ranges.
az cosmosdb create \
    -n $accountName \
    -g $resourceGroupName \
    --locations regionName='West US 2' failoverPriority=0 isZoneRedundant=False \
    --locations regionName='East US 2' failoverPriority=1 isZoneRedundant=False \
    --ip-range-filter $ipRangeFilter
```

Configure an IP access control policy by using PowerShell

The following script shows how to create an Azure Cosmos DB account with IP access control:

```
# Create a Cosmos DB account with default values and IP Firewall enabled
$resourceGroupName = "myResourceGroup"
$accountName = "mycosmosaccount"
$ipRules = @("192.168.221.17","183.240.196.255","40.76.54.131")

$locations = @(
    @{ "locationName"="West US 2"; "failoverPriority"=0; "isZoneRedundant"=False },
    @{ "locationName"="East US 2"; "failoverPriority"=1, "isZoneRedundant"=False }
)

# Make sure there are no spaces in the comma-delimited list of IP addresses or CIDR ranges.
$CosmosDBProperties = @{
    "databaseAccountOfferType"="Standard";
    "locations"=$locations;
    "ipRules"=$ipRules
}

New-AzResource -ResourceType "Microsoft.DocumentDb/databaseAccounts" `
    -ApiVersion "2020-04-01" -ResourceGroupName $resourceGroupName `
    -Name $accountName -PropertyObject $CosmosDBProperties
```

Troubleshoot issues with an IP access control policy

You can troubleshoot issues with an IP access control policy by using the following options:

Azure portal

By enabling an IP access control policy for your Azure Cosmos DB account, you block all requests to your account from machines outside the allowed list of IP address ranges. To enable portal data-plane operations like browsing containers and querying documents, you need to explicitly allow Azure portal access by using the **Firewall** pane in the portal.

SDKs

When you access Azure Cosmos DB resources by using SDKs from machines that are not in the allowed list, a generic **403 Forbidden** response is returned with no additional details. Verify the allowed IP list for your account, and make sure that the correct policy configuration is applied to your Azure Cosmos DB account.

Source IPs in blocked requests

Enable diagnostic logging on your Azure Cosmos DB account. These logs show each request and response. The firewall-related messages are logged with a 403 return code. By filtering these messages, you can see the source IPs for the blocked requests. See Azure Cosmos DB diagnostic logging.

Requests from a subnet with a service endpoint for Azure Cosmos DB enabled

Requests from a subnet in a virtual network that has a service endpoint for Azure Cosmos DB enabled sends the virtual network and subnet identity to Azure Cosmos DB accounts. These requests don't have the public IP of the source, so IP filters reject them. To allow access from specific subnets in virtual networks, add an access control list as outlined in How to configure virtual network and subnet-based access for your Azure Cosmos DB account. It can take up to 15 minutes for firewall rules to apply.

Private IP addresses in list of allowed addresses

Creating or updating an Azure Cosmos account with a list of allowed addresses containing private IP addresses will fail. Make sure that no private IP address is specified in the list.

Next steps

To configure a virtual network service endpoint for your Azure Cosmos DB account, see the following articles:

- Virtual network and subnet access control for your Azure Cosmos DB account
- Configure virtual network and subnet-based access for your Azure Cosmos DB account

Configure access to Azure Cosmos DB from virtual networks (VNet)

12/4/2020 • 13 minutes to read • Edit Online

APPLIES TO: SQL API Cassandra API Gremlin API Table API Azure Cosmos DB API for MongoDB

You can configure the Azure Cosmos account to allow access only from a specific subnet of virtual network (VNet). By enabling Service endpoint to access Azure Cosmos DB on the subnet within a virtual network, the traffic from that subnet is sent to Azure Cosmos DB with the identity of the subnet and Virtual Network. Once the Azure Cosmos DB service endpoint is enabled, you can limit access to the subnet by adding it to your Azure Cosmos account.

By default, an Azure Cosmos account is accessible from any source if the request is accompanied by a valid authorization token. When you add one or more subnets within VNets, only requests originating from those subnets will get a valid response. Requests originating from any other source will receive a 403 (Forbidden) response.

You can configure Azure Cosmos DB accounts to allow access from only a specific subnet of an Azure virtual network. To limit access to an Azure Cosmos DB account with connections from a subnet in a virtual network:

1. Enable the subnet to send the subnet and virtual network identity to Azure Cosmos DB. You can achieve this by enabling a service endpoint for Azure Cosmos DB on the specific subnet.
2. Add a rule in the Azure Cosmos DB account to specify the subnet as a source from which the account can be accessed.

> **NOTE**
>
> When a service endpoint for your Azure Cosmos DB account is enabled on a subnet, the source of the traffic that reaches Azure Cosmos DB switches from a public IP to a virtual network and subnet. The traffic switching applies for any Azure Cosmos DB account that's accessed from this subnet. If your Azure Cosmos DB accounts have an IP-based firewall to allow this subnet, requests from the service-enabled subnet no longer match the IP firewall rules, and they're rejected.
>
> To learn more, see the steps outlined in the Migrating from an IP firewall rule to a virtual network access control list section of this article.

The following sections describe how to configure a virtual network service endpoint for an Azure Cosmos DB account.

Configure a service endpoint by using the Azure portal

Configure a service endpoint for an existing Azure virtual network and subnet

1. From the **All resources** blade, find the Azure Cosmos DB account that you want to secure.
2. Select **Firewalls and virtual networks** from the settings menu, and choose to allow access from **Selected networks**.
3. To grant access to an existing virtual network's subnet, under **Virtual networks**, select **Add existing Azure virtual network**.
4. Select the **Subscription** from which you want to add an Azure virtual network. Select the Azure **Virtual**

networks and **Subnets** that you want to provide access to your Azure Cosmos DB account. Next, select **Enable** to enable selected networks with service endpoints for "Microsoft.AzureCosmosDB". When it's complete, select **Add**.

5. After the Azure Cosmos DB account is enabled for access from a virtual network, it will allow traffic from only this chosen subnet. The virtual network and subnet that you added should appear as shown in the following screenshot:

> **NOTE**
>
> To enable virtual network service endpoints, you need the following subscription permissions:
>
> - Subscription with virtual network: Network contributor
> - Subscription with Azure Cosmos DB account: DocumentDB account contributor
> - If your virtual network and Azure Cosmos DB account are in different subscriptions, make sure that the subscription that has virtual network also has `Microsoft.DocumentDB` resource provider registered. To register a resource provider, see Azure resource providers and types article.

Here are the directions for registering subscription with resource provider.

Configure a service endpoint for a new Azure virtual network and subnet

1. From the **All resources** blade, find the Azure Cosmos DB account that you want to secure.

2. Select **Firewalls and Azure virtual networks** from the settings menu, and choose to allow access from **Selected networks**.

3. To grant access to a new Azure virtual network, under **Virtual networks**, select **Add new virtual network**.

4. Provide the details required to create a new virtual network, and then select **Create**. The subnet will be created with a service endpoint for "Microsoft.AzureCosmosDB" enabled.

If your Azure Cosmos DB account is used by other Azure services like Azure Cognitive Search, or is accessed from Stream analytics or Power BI, you allow access by selecting **Accept connections from within global Azure datacenters**.

To ensure that you have access to Azure Cosmos DB metrics from the portal, you need to enable **Allow access from Azure portal** options. To learn more about these options, see the Configure an IP firewall article. After you enable access, select **Save** to save the settings.

Remove a virtual network or subnet

1. From the **All resources** blade, find the Azure Cosmos DB account for which you assigned service endpoints.

2. Select **Firewalls and virtual networks** from the settings menu.

3. To remove a virtual network or subnet rule, select **...** next to the virtual network or subnet, and select **Remove**.

4. Select **Save** to apply your changes.

Configure a service endpoint by using Azure PowerShell

> **NOTE**
> This article has been updated to use the new Azure PowerShell Az module. You can still use the AzureRM module, which will continue to receive bug fixes until at least December 2020. To learn more about the new Az module and AzureRM compatibility, see Introducing the new Azure PowerShell Az module. For Az module installation instructions, see Install Azure PowerShell.

Use the following steps to configure a service endpoint to an Azure Cosmos DB account by using Azure PowerShell:

1. Install Azure PowerShell and sign in.

2. Enable the service endpoint for an existing subnet of a virtual network.

```
$resourceGroupName = "<Resource group name>"
$vnetName = "<Virtual network name>"
$subnetName = "<Subnet name>"
$subnetPrefix = "<Subnet address range>"
$serviceEndpoint = "Microsoft.AzureCosmosDB"

Get-AzVirtualNetwork `
   -ResourceGroupName $resourceGroupName `
   -Name $vnetName | Set-AzVirtualNetworkSubnetConfig `
   -Name $subnetName `
   -AddressPrefix $subnetPrefix `
   -ServiceEndpoint $serviceEndpoint | Set-AzVirtualNetwork
```

> **NOTE**
> When you're using PowerShell or the Azure CLI, be sure to specify the complete list of IP filters and virtual network ACLs in parameters, not just the ones that need to be added.

3. Get virtual network information.

```
$vnet = Get-AzVirtualNetwork `
   -ResourceGroupName $resourceGroupName `
   -Name $vnetName

$subnetId = $vnet.Id + "/subnets/" + $subnetName
```

4. Prepare a Cosmos DB Virtual Network Rule

```
$vnetRule = New-AzCosmosDBVirtualNetworkRule `
   -Id $subnetId
```

5. Update Azure Cosmos DB account properties with the new Virtual Network endpoint configuration:

```
$accountName = "<Cosmos DB account name>"

Update-AzCosmosDBAccount `
   -ResourceGroupName $resourceGroupName `
   -Name $accountName `
   -EnableVirtualNetwork $true `
   -VirtualNetworkRuleObject @($vnetRule)
```

6. Run the following command to verify that your Azure Cosmos DB account is updated with the virtual

network service endpoint that you configured in the previous step:

```
$account = Get-AzCosmosDBAccount `
   -ResourceGroupName $resourceGroupName `
   -Name $accountName

$account.IsVirtualNetworkFilterEnabled
$account.VirtualNetworkRules
```

Configure a service endpoint by using the Azure CLI

Azure Cosmos accounts can be configured for service endpoints when they are created or updated at a later time if the subnet is already configured for them. Service endpoints can also be enabled on the Cosmos account where the subnet is not yet configured for them and then will begin to work when the subnet is configured later. This flexibility allows for administrators who do not have access to both the Cosmos account and virtual network resources to make their configurations independent of each other.

Create a new Cosmos account and connect it to a back end subnet for a new virtual network

In this example the virtual network and subnet is created with service endpoints enabled for both when they are created.

```
# Create an Azure Cosmos Account with a service endpoint connected to a backend subnet

# Resource group and Cosmos account variables
resourceGroupName='MyResourceGroup'
location='West US 2'
accountName='mycosmosaccount'

# Variables for a new Virtual Network with two subnets
vnetName='myVnet'
frontEnd='FrontEnd'
backEnd='BackEnd'

# Create a resource group
az group create -n $resourceGroupName -l $location

# Create a virtual network with a front-end subnet
az network vnet create \
   -n $vnetName \
   -g $resourceGroupName \
   --address-prefix 10.0.0.0/16 \
   --subnet-name $frontEnd \
   --subnet-prefix 10.0.1.0/24

# Create a back-end subnet with service endpoints enabled for Cosmos DB
az network vnet subnet create \
   -n $backEnd \
   -g $resourceGroupName \
   --address-prefix 10.0.2.0/24 \
   --vnet-name $vnetName \
   --service-endpoints Microsoft.AzureCosmosDB

svcEndpoint=$(az network vnet subnet show -g $resourceGroupName -n $backEnd --vnet-name $vnetName --query
'id' -o tsv)

# Create a Cosmos DB account with default values and service endpoints
az cosmosdb create \
   -n $accountName \
   -g $resourceGroupName \
   --enable-virtual-network true \
   --virtual-network-rules $svcEndpoint
```

Connect and configure a Cosmos account to a back end subnet independently

This sample is intended to show how to connect an Azure Cosmos account to an existing new virtual network where the subnet is not yet configured for service endpoints. This is done by using the `--ignore-missing-vnet-service-endpoint` parameter. This allows the configuration for the Cosmos account to complete without error before the configuration to the virtual network's subnet is complete. Once the subnet configuration is complete, the Cosmos account will then be accessible through the configured subnet.

```
# Create an Azure Cosmos Account with a service endpoint connected to a backend subnet
# that is not yet enabled for service endpoints.

# Resource group and Cosmos account variables
resourceGroupName='MyResourceGroup'
location='West US 2'
accountName='mycosmosaccount'

# Variables for a new Virtual Network with two subnets
vnetName='myVnet'
frontEnd='FrontEnd'
backEnd='BackEnd'

# Create a resource group
az group create -n $resourceGroupName -l $location

# Create a virtual network with a front-end subnet
az network vnet create \
   -n $vnetName \
   -g $resourceGroupName \
   --address-prefix 10.0.0.0/16 \
   --subnet-name $frontEnd \
   --subnet-prefix 10.0.1.0/24

# Create a back-end subnet but without configuring service endpoints (--service-endpoints
Microsoft.AzureCosmosDB)
az network vnet subnet create \
   -n $backEnd \
   -g $resourceGroupName \
   --address-prefix 10.0.2.0/24 \
   --vnet-name $vnetName

svcEndpoint=$(az network vnet subnet show -g $resourceGroupName -n $backEnd --vnet-name $vnetName --query
'id' -o tsv)

# Create a Cosmos DB account with default values
az cosmosdb create -n $accountName -g $resourceGroupName

# Add the virtual network rule but ignore the missing service endpoint on the subnet
az cosmosdb network-rule add \
   -n $accountName \
   -g $resourceGroupName \
   --virtual-network $vnetName \
   --subnet svcEndpoint \
   --ignore-missing-vnet-service-endpoint true

read -p'Press any key to now configure the subnet for service endpoints'

az network vnet subnet update \
   -n $backEnd \
   -g $resourceGroupName \
   --vnet-name $vnetName \
   --service-endpoints Microsoft.AzureCosmosDB
```

Port range when using direct mode

When you're using service endpoints with an Azure Cosmos account through a direct mode connection, you

need to ensure that the TCP port range from 10000 to 20000 is open.

Migrating from an IP firewall rule to a virtual network ACL

To migrate an Azure Cosmos DB account from using IP firewall rules to using virtual network service endpoints, use the following steps.

After an Azure Cosmos DB account is configured for a service endpoint for a subnet, requests from that subnet are sent to Azure Cosmos DB with virtual network and subnet source information instead of a source public IP address. These requests will no longer match an IP filter configured on the Azure Cosmos DB account, which is why the following steps are necessary to avoid downtime.

Before proceeding, enable the Azure Cosmos DB service endpoint on the virtual network and subnet using the step shown above in "Enable the service endpoint for an existing subnet of a virtual network".

1. Get virtual network and subnet information:

```
$resourceGroupName = "myResourceGroup"
$accountName = "mycosmosaccount"
$vnetName = "myVnet"
$subnetName = "mySubnet"

$vnet = Get-AzVirtualNetwork `
   -ResourceGroupName $resourceGroupName `
   -Name $vnetName

$subnetId = $vnet.Id + "/subnets/" + $subnetName
```

2. Prepare a new Virtual Network rule object for the Azure Cosmos DB account:

```
$vnetRule = New-AzCosmosDBVirtualNetworkRule `
   -Id $subnetId
```

3. Update the Azure Cosmos DB account to enable service endpoint access from the subnet:

```
Update-AzCosmosDBAccount `
   -ResourceGroupName $resourceGroupName `
   -Name $accountName `
   -EnableVirtualNetwork $true `
   -VirtualNetworkRuleObject @($vnetRule)
```

4. Repeat the previous steps for all Azure Cosmos DB accounts accessed from the subnet.

5. Remove the IP firewall rule for the subnet from the Azure Cosmos DB account's Firewall rules.

Frequently asked questions

Here are some frequently asked questions about configuring access from virtual networks:

Are Notebooks and Mongo/Cassandra Shell currently compatible with Virtual Network enabled accounts?

At the moment the Mongo shell and Cassandra shell integrations in the Cosmos DB Data Explorer, and the Jupyter Notebooks service, are not supported with VNET access. This is currently in active development.

Can I specify both virtual network service endpoint and IP access control policy on an Azure Cosmos account?

You can enable both the virtual network service endpoint and an IP access control policy (also known as firewall) on your Azure Cosmos account. These two features are complementary and collectively ensure isolation and security of your Azure Cosmos account. Using IP firewall ensures that static IPs can access your account.

How do I limit access to subnet within a virtual network?

There are two steps required to limit access to Azure Cosmos account from a subnet. First, you allow traffic from subnet to carry its subnet and virtual network identity to Azure Cosmos DB. This is done by enabling service endpoint for Azure Cosmos DB on the subnet. Next is adding a rule in the Azure Cosmos account specifying this subnet as a source from which account can be accessed.

Will virtual network ACLs and IP Firewall reject requests or connections?

When IP firewall or virtual network access rules are added, only requests from allowed sources get valid responses. Other requests are rejected with a 403 (Forbidden). It is important to distinguish Azure Cosmos account's firewall from a connection level firewall. The source can still connect to the service and the connections themselves aren't rejected.

My requests started getting blocked when I enabled service endpoint to Azure Cosmos DB on the subnet. What happened?

Once service endpoint for Azure Cosmos DB is enabled on a subnet, the source of the traffic reaching the account switches from public IP to virtual network and subnet. If your Azure Cosmos account has IP-based firewall only, traffic from service enabled subnet would no longer match the IP firewall rules and therefore be rejected. Go over the steps to seamlessly migrate from IP-based firewall to virtual network-based access control.

Are additional Azure RBAC permissions needed for Azure Cosmos accounts with VNET service endpoints?

After you add the VNet service endpoints to an Azure Cosmos account, to make any changes to the account settings, you need access to the `Microsoft.Network/virtualNetworks/subnets/joinViaServiceEndpoint/action` action for all the VNETs configured on your Azure Cosmos account. This permission is required because the authorization process validates access to resources (such as database and virtual network resources) before evaluating any properties.

The authorization validates permission for VNet resource action even if the user doesn't specify the VNET ACLs using Azure CLI. Currently, the Azure Cosmos account's control plane supports setting the complete state of the Azure Cosmos account. One of the parameters to the control plane calls is `virtualNetworkRules`. If this parameter is not specified, the Azure CLI makes a get database call to retrieves the `virtualNetworkRules` and uses this value in the update call.

Do the peered virtual networks also have access to Azure Cosmos account?

Only virtual network and their subnets added to Azure Cosmos account have access. Their peered VNets cannot access the account until the subnets within peered virtual networks are added to the account.

What is the maximum number of subnets allowed to access a single Cosmos account?

Currently, you can have at most 256 subnets allowed for an Azure Cosmos account.

Can I enable access from VPN and Express Route?

For accessing Azure Cosmos account over Express route from on premises, you would need to enable Microsoft peering. Once you put IP firewall or virtual network access rules, you can add the public IP addresses used for Microsoft peering on your Azure Cosmos account IP firewall to allow on premises services access to Azure Cosmos account.

Do I need to update the Network Security Groups (NSG) rules?

NSG rules are used to limit connectivity to and from a subnet with virtual network. When you add service endpoint for Azure Cosmos DB to the subnet, there is no need to open outbound connectivity in NSG for your Azure Cosmos account.

Are service endpoints available for all VNets?

No, Only Azure Resource Manager virtual networks can have service endpoint enabled. Classic virtual networks don't support service endpoints.

Can I "Accept connections from within public Azure datacenters" when service endpoint access is enabled for

Azure Cosmos DB?

This is required only when you want your Azure Cosmos DB account to be accessed by other Azure first party services like Azure Data factory, Azure Cognitive Search or any service that is deployed in given Azure region.

Next steps

- To configure a firewall for Azure Cosmos DB, see the Firewall support article.

Configure Azure Private Link for an Azure Cosmos account

11/2/2020 • 18 minutes to read • Edit Online

APPLIES TO: SQL API Cassandra API Gremlin API Table API Azure Cosmos DB API for MongoDB

By using Azure Private Link, you can connect to an Azure Cosmos account via a private endpoint. The private endpoint is a set of private IP addresses in a subnet within your virtual network. You can then limit access to an Azure Cosmos account over private IP addresses. When Private Link is combined with restricted NSG policies, it helps reduce the risk of data exfiltration. To learn more about private endpoints, see the Azure Private Link article.

> **NOTE**
> Private Link doesn't prevent your Azure Cosmos endpoints from being resolved by public DNS. Filtering of incoming requests happens at application level, not transport or network level.

Private Link allows users to access an Azure Cosmos account from within the virtual network or from any peered virtual network. Resources mapped to Private Link are also accessible on-premises over private peering through VPN or Azure ExpressRoute.

You can connect to an Azure Cosmos account configured with Private Link by using the automatic or manual approval method. To learn more, see the Approval workflow section of the Private Link documentation.

This article describes the steps to create a private endpoint. It assumes that you're using the automatic approval method.

Create a private endpoint by using the Azure portal

Use the following steps to create a private endpoint for an existing Azure Cosmos account by using the Azure portal:

1. From the **All resources** pane, choose an Azure Cosmos account.
2. Select **Private Endpoint Connections** from the list of settings, and then select **Private endpoint**:

3. In the **Create a private endpoint - Basics** pane, enter or select the following details:

SETTING	VALUE
Project details	
Subscription	Select your subscription.
Resource group	Select a resource group.
Instance details	
Name	Enter any name for your private endpoint. If this name is taken, create a unique one.
Region	Select the region where you want to deploy Private Link. Create the private endpoint in the same location where your virtual network exists.

4. Select **Next: Resource**.

5. In **Create a private endpoint - Resource**, enter or select this information:

SETTING	VALUE
Connection method	Select **Connect to an Azure resource in my directory**. You can then choose one of your resources to set up Private Link. Or you can connect to someone else's resource by using a resource ID or alias that they've shared with you.

SETTING	VALUE
Subscription	Select your subscription.
Resource type	Select **Microsoft.AzureCosmosDB/databaseAccounts**.
Resource	Select your Azure Cosmos account.
Target sub-resource	Select the Azure Cosmos DB API type that you want to map. This defaults to only one choice for the SQL, MongoDB, and Cassandra APIs. For the Gremlin and Table APIs, you can also choose **Sql** because these APIs are interoperable with the SQL API.

6. Select **Next: Configuration**.
7. In **Create a private endpoint - Configuration**, enter or select this information:

SETTING	VALUE
Networking	
Virtual network	Select your virtual network.
Subnet	Selectyour subnet.
Private DNS Integration	
Integrate with private DNS zone	Select **Yes**. To connect privately with your private endpoint, you need a DNS record. We recommend that you integrate your private endpoint with a private DNS zone. You can also use your own DNS servers or create DNS records by using the host files on your virtual machines.
Private DNS Zone	Select **privatelink.documents.azure.com**. The private DNS zone is determined automatically. You can't change it by using the Azure portal.

8. Select **Review + create**. On the **Review + create** page, Azure validates your configuration.
9. When you see the **Validation passed** message, select **Create**.

When you have approved Private Link for an Azure Cosmos account, in the Azure portal, the **All networks** option in the **Firewall and virtual networks** pane is unavailable.

The following table shows the mapping between different Azure Cosmos account API types, supported sub-resources, and the corresponding private zone names. You can also access the Gremlin and Table API accounts through the SQL API, so there are two entries for these APIs.

AZURE COSMOS ACCOUNT API TYPE	SUPPORTED SUB-RESOURCES (OR GROUP IDS)	PRIVATE ZONE NAME
Sql	Sql	privatelink.documents.azure.com
Cassandra	Cassandra	privatelink.cassandra.cosmos.azure.com
Mongo	MongoDB	privatelink.mongo.cosmos.azure.com
Gremlin	Gremlin	privatelink.gremlin.cosmos.azure.com
Gremlin	Sql	privatelink.documents.azure.com
Table	Table	privatelink.table.cosmos.azure.com
Table	Sql	privatelink.documents.azure.com

Fetch the private IP addresses

After the private endpoint is provisioned, you can query the IP addresses. To view the IP addresses from the Azure portal:

1. Select **All resources**.
2. Search for the private endpoint that you created earlier. In this case, it's **cdbPrivateEndpoint3**.
3. Select the **Overview** tab to see the DNS settings and IP addresses.

Multiple IP addresses are created per private endpoint:

- One for the global (region-agnostic) endpoint of the Azure Cosmos account
- One for each region where the Azure Cosmos account is deployed

Create a private endpoint by using Azure PowerShell

Run the following PowerShell script to create a private endpoint named "MyPrivateEndpoint" for an existing Azure Cosmos account. Replace the variable values with the details for your environment.

```
$SubscriptionId = "<your Azure subscription ID>"
# Resource group where the Azure Cosmos account and virtual network resources are located
$ResourceGroupName = "myResourceGroup"
# Name of the Azure Cosmos account
$CosmosDbAccountName = "mycosmosaccount"

# API type of the Azure Cosmos account: Sql, MongoDB, Cassandra, Gremlin, or Table
$CosmosDbApiType = "Sql"
# Name of the existing virtual network
$VNetName = "myVnet"
# Name of the target subnet in the virtual network
$SubnetName = "mySubnet"
# Name of the private endpoint to create
$PrivateEndpointName = "MyPrivateEndpoint"
# Location where the private endpoint can be created. The private endpoint should be created in the same
location where your subnet or the virtual network exists
$Location = "westcentralus"

$cosmosDbResourceId =
"/subscriptions/$($SubscriptionId)/resourceGroups/$($ResourceGroupName)/providers/Microsoft.DocumentDB/databas
eAccounts/$($CosmosDbAccountName)"

$privateEndpointConnection = New-AzPrivateLinkServiceConnection -Name "myConnectionPS" -PrivateLinkServiceId
$cosmosDbResourceId -GroupId $CosmosDbApiType

$virtualNetwork = Get-AzVirtualNetwork -ResourceGroupName  $ResourceGroupName -Name $VNetName

$subnet = $virtualNetwork | Select -ExpandProperty subnets | Where-Object  {$_.Name -eq $SubnetName}

$privateEndpoint = New-AzPrivateEndpoint -ResourceGroupName $ResourceGroupName -Name $PrivateEndpointName -
Location "westcentralus" -Subnet  $subnet -PrivateLinkServiceConnection $privateEndpointConnection
```

Integrate the private endpoint with a private DNS zone

After you create the private endpoint, you can integrate it with a private DNS zone by using the following PowerShell script:

```
Import-Module Az.PrivateDns
$zoneName = "privatelink.documents.azure.com"
$zone = New-AzPrivateDnsZone -ResourceGroupName $ResourceGroupName `
  -Name $zoneName

$link  = New-AzPrivateDnsVirtualNetworkLink -ResourceGroupName $ResourceGroupName `
  -ZoneName $zoneName `
  -Name "myzonelink" `
  -VirtualNetworkId $virtualNetwork.Id

$pe = Get-AzPrivateEndpoint -Name $PrivateEndpointName `
  -ResourceGroupName $ResourceGroupName

$networkInterface = Get-AzResource -ResourceId $pe.NetworkInterfaces[0].Id `
  -ApiVersion "2019-04-01"

foreach ($ipconfig in $networkInterface.properties.ipConfigurations) {
foreach ($fqdn in $ipconfig.properties.privateLinkConnectionProperties.fqdns) {
Write-Host "$($ipconfig.properties.privateIPAddress) $($fqdn)"
$recordName = $fqdn.split('.',2)[0]
$dnsZone = $fqdn.split('.',2)[1]
New-AzPrivateDnsRecordSet -Name $recordName `
  -RecordType A -ZoneName $zoneName  `
  -ResourceGroupName $ResourceGroupName -Ttl 600 `
  -PrivateDnsRecords (New-AzPrivateDnsRecordConfig `
  -IPv4Address $ipconfig.properties.privateIPAddress)
}
}
```

Fetch the private IP addresses

After the private endpoint is provisioned, you can query the IP addresses and the FQDN mapping by using the following PowerShell script:

```
$pe = Get-AzPrivateEndpoint -Name MyPrivateEndpoint -ResourceGroupName myResourceGroup
$networkInterface = Get-AzNetworkInterface -ResourceId $pe.NetworkInterfaces[0].Id
foreach ($IPConfiguration in $networkInterface.IpConfigurations)
{
    Write-Host $IPConfiguration.PrivateIpAddress ":" $IPConfiguration.PrivateLinkConnectionProperties.Fqdns
}
```

Create a private endpoint by using Azure CLI

Run the following Azure CLI script to create a private endpoint named "myPrivateEndpoint" for an existing Azure Cosmos account. Replace the variable values with the details for your environment.

```
# Resource group where the Azure Cosmos account and virtual network resources are located
ResourceGroupName="myResourceGroup"

# Subscription ID where the Azure Cosmos account and virtual network resources are located
SubscriptionId="<your Azure subscription ID>"

# Name of the existing Azure Cosmos account
CosmosDbAccountName="mycosmosaccount"

# API type of your Azure Cosmos account: Sql, MongoDB, Cassandra, Gremlin, or Table
CosmosDbApiType="Sql"

# Name of the virtual network to create
VNetName="myVnet"

# Name of the subnet to create
SubnetName="mySubnet"

# Name of the private endpoint to create
PrivateEndpointName="myPrivateEndpoint"

# Name of the private endpoint connection to create
PrivateConnectionName="myConnection"

az network vnet create \
 --name $VNetName \
 --resource-group $ResourceGroupName \
 --subnet-name $SubnetName

az network vnet subnet update \
 --name $SubnetName \
 --resource-group $ResourceGroupName \
 --vnet-name $VNetName \
 --disable-private-endpoint-network-policies true

az network private-endpoint create \
    --name $PrivateEndpointName \
    --resource-group $ResourceGroupName \
    --vnet-name $VNetName  \
    --subnet $SubnetName \
    --private-connection-resource-id
"/subscriptions/$SubscriptionId/resourceGroups/$ResourceGroupName/providers/Microsoft.DocumentDB/databaseAccou
nts/$CosmosDbAccountName" \
    --group-ids $CosmosDbApiType \
    --connection-name $PrivateConnectionName
```

Integrate the private endpoint with a private DNS zone

After you create the private endpoint, you can integrate it with a private DNS zone by using the following Azure CLI script:

```
zoneName="privatelink.documents.azure.com"

az network private-dns zone create --resource-group $ResourceGroupName \
   --name  $zoneName

az network private-dns link vnet create --resource-group $ResourceGroupName \
   --zone-name  $zoneName\
   --name myzonelink \
   --virtual-network $VNetName \
   --registration-enabled false

#Query for the network interface ID
networkInterfaceId=$(az network private-endpoint show --name $PrivateEndpointName --resource-group
$ResourceGroupName --query 'networkInterfaces[0].id' -o tsv)

# Copy the content for privateIPAddress and FQDN matching the Azure Cosmos account
az resource show --ids $networkInterfaceId --api-version 2019-04-01 -o json

#Create DNS records
az network private-dns record-set a create --name recordSet1 --zone-name privatelink.documents.azure.com --
resource-group $ResourceGroupName
az network private-dns record-set a add-record --record-set-name recordSet2 --zone-name
privatelink.documents.azure.com --resource-group $ResourceGroupName -a <Private IP Address>
```

Create a private endpoint by using a Resource Manager template

You can set up Private Link by creating a private endpoint in a virtual network subnet. You achieve this by using an Azure Resource Manager template.

Use the following code to create a Resource Manager template named "PrivateEndpoint_template.json." This template creates a private endpoint for an existing Azure Cosmos SQL API account in an existing virtual network.

```
{
    "$schema": "http://schema.management.azure.com/schemas/2015-01-01/deploymentTemplate.json#",
    "contentVersion": "1.0.0.0",
    "parameters": {
        "location": {
          "type": "string",
          "defaultValue": "[resourceGroup().location]",
          "metadata": {
            "description": "Location for all resources."
          }
        },
        "privateEndpointName": {
            "type": "string"
        },
        "resourceId": {
            "type": "string"
        },
        "groupId": {
            "type": "string"
        },
        "subnetId": {
            "type": "string"
        }
    },
    "resources": [
        {
            "name": "[parameters('privateEndpointName')]",
            "type": "Microsoft.Network/privateEndpoints",
            "apiVersion": "2019-04-01",
            "location": "[parameters('location')]",
            "properties": {
                "subnet": {
                    "id": "[parameters('subnetId')]"
                },
                "privateLinkServiceConnections": [
                    {
                        "name": "MyConnection",
                        "properties": {
                            "privateLinkServiceId": "[parameters('resourceId')]",
                            "groupIds": [ "[parameters('groupId')]" ],
                            "requestMessage": ""
                        }
                    }
                ]
            }
        }
    ],
    "outputs": {
        "privateEndpointNetworkInterface": {
          "type": "string",
          "value": "[reference(concat('Microsoft.Network/privateEndpoints/',
parameters('privateEndpointName'))).networkInterfaces[0].id]"
        }
    }
}
```

Define the parameters file for the template

Create a parameters file for the template, and name it "PrivateEndpoint_parameters.json." Add the following code to the parameters file:

```
{
    "$schema": "https://schema.management.azure.com/schemas/2015-01-01/deploymentParameters.json#",
    "contentVersion": "1.0.0.0",
    "parameters": {
        "privateEndpointName": {
            "value": ""
        },
        "resourceId": {
            "value": ""
        },
        "groupId": {
            "value": ""
        },
        "subnetId": {
            "value": ""
        }
    }
}
```

Deploy the template by using a PowerShell script

Create a PowerShell script by using the following code. Before you run the script, replace the subscription ID, resource group name, and other variable values with the details for your environment.

```
### This script creates a private endpoint for an existing Azure Cosmos account in an existing virtual network

## Step 1: Fill in these details. Replace the variable values with the details for your environment.
$SubscriptionId = "<your Azure subscription ID>"
# Resource group where the Azure Cosmos account and virtual network resources are located
$ResourceGroupName = "myResourceGroup"
# Name of the Azure Cosmos account
$CosmosDbAccountName = "mycosmosaccount"
# API type of the Azure Cosmos account. It can be one of the following: "Sql", "MongoDB", "Cassandra",
"Gremlin", "Table"
$CosmosDbApiType = "Sql"
# Name of the existing virtual network
$VNetName = "myVnet"
# Name of the target subnet in the virtual network
$SubnetName = "mySubnet"
# Name of the private endpoint to create
$PrivateEndpointName = "myPrivateEndpoint"

$cosmosDbResourceId =
"/subscriptions/$($SubscriptionId)/resourceGroups/$($ResourceGroupName)/providers/Microsoft.DocumentDB/databas
eAccounts/$($CosmosDbAccountName)"
$VNetResourceId =
"/subscriptions/$($SubscriptionId)/resourceGroups/$($ResourceGroupName)/providers/Microsoft.Network/virtualNet
works/$($VNetName)"
$SubnetResourceId = "$($VNetResourceId)/subnets/$($SubnetName)"
$PrivateEndpointTemplateFilePath = "PrivateEndpoint_template.json"
$PrivateEndpointParametersFilePath = "PrivateEndpoint_parameters.json"

## Step 2: Sign in to your Azure account and select the target subscription.
Login-AzAccount
Select-AzSubscription -SubscriptionId $subscriptionId

## Step 3: Make sure private endpoint network policies are disabled in the subnet.
$VirtualNetwork= Get-AzVirtualNetwork -Name "$VNetName" -ResourceGroupName "$ResourceGroupName"
($virtualNetwork | Select -ExpandProperty subnets | Where-Object  {$_.Name -eq "$SubnetName"}
).PrivateEndpointNetworkPolicies = "Disabled"
$virtualNetwork | Set-AzVirtualNetwork

## Step 4: Create the private endpoint.
Write-Output "Deploying private endpoint on $($resourceGroupName)"
$deploymentOutput = New-AzResourceGroupDeployment -Name "PrivateCosmosDbEndpointDeployment" `
    -ResourceGroupName $resourceGroupName `
    -TemplateFile $PrivateEndpointTemplateFilePath `
    -TemplateParameterFile $PrivateEndpointParametersFilePath `
    -SubnetId $SubnetResourceId `
    -ResourceId $CosmosDbResourceId `
    -GroupId $CosmosDbApiType `
    -PrivateEndpointName $PrivateEndpointName

$deploymentOutput
```

In the PowerShell script, the `GroupId` variable can contain only one value. That value is the API type of the account. Allowed values are: `Sql`, `MongoDB`, `Cassandra`, `Gremlin`, and `Table`. Some Azure Cosmos account types are accessible through multiple APIs. For example:

- A Gremlin API account can be accessed from both Gremlin and SQL API accounts.
- A Table API account can be accessed from both Table and SQL API accounts.

For those accounts, you must create one private endpoint for each API type. The corresponding API type is specified in the `GroupId` array.

After the template is deployed successfully, you can see an output similar to what the following image shows. The `provisioningState` value is `Succeeded` if the private endpoints are set up correctly.

```
Deploying private endpoint on cdbrg

ResourceGroupName       : cdbrg
OnErrorDeployment       :
DeploymentName          : PrivateCosmosDbEndpointDeployment
CorrelationId           : [redacted]
ProvisioningState       : Succeeded
Timestamp               : 10/22/2019 6:03:14 PM
Mode                    : Incremental
TemplateLink            :
TemplateLinkString      :
DeploymentDebugLogLevel :
Parameters              : {[location, Microsoft.Azure.Commands.ResourceManager.Cmdlets.SdkModels.DeploymentVariable], [privateEndpointName,
                          Microsoft.Azure.Commands.ResourceManager.Cmdlets.SdkModels.DeploymentVariable], [resourceId,
                          Microsoft.Azure.Commands.ResourceManager.Cmdlets.SdkModels.DeploymentVariable], [groupId,
                          Microsoft.Azure.Commands.ResourceManager.Cmdlets.SdkModels.DeploymentVariable]...}
ParametersString        :
                          Name                  Type                      Value
                          ====================  ========================  ==========
                          location              String                    eastus
                          privateEndpointName   String                    cdbPrivateEndpoint3
                          resourceId            String                    /subscriptions/[redacted]/resourceGroups/cdbr
                          g/providers/Microsoft.DocumentDB/databaseAccounts/sqlcdb2
                          groupId               String                    Sql
                          subnetId              String                    /subscriptions/[redacted]/resourceGroups/cdbr
                          g/providers/Microsoft.Network/virtualNetworks/cdbVnet/subnets/cdbSubnet

Outputs                 : {[privateEndpointNetworkInterface, Microsoft.Azure.Commands.ResourceManager.Cmdlets.SdkModels.DeploymentVariable]}
OutputsString           :
                          Name                              Type                      Value
                          ==============================    ========================  ==========
                          privateEndpointNetworkInterface   String                    /subscriptions/[redacted]/resourc
                          eGroups/cdbrg/providers/Microsoft.Network/networkInterfaces/cdbPrivateEndpoint3.nic.039046f0-efe8-4165-8570-9557ebb8ccd3
```

After the template is deployed, the private IP addresses are reserved within the subnet. The firewall rule of the Azure Cosmos account is configured to accept connections from the private endpoint only.

Integrate the private endpoint with a Private DNS Zone

Use the following code to create a Resource Manager template named "PrivateZone_template.json." This template creates a private DNS zone for an existing Azure Cosmos SQL API account in an existing virtual network.

```
{
    "$schema": "http://schema.management.azure.com/schemas/2015-01-01/deploymentTemplate.json#",
    "contentVersion": "1.0.0.0",
    "parameters": {
        "privateZoneName": {
            "type": "string"
        },
        "VNetId": {
            "type": "string"
        }
    },
    "resources": [
        {
            "name": "[parameters('privateZoneName')]",
            "type": "Microsoft.Network/privateDnsZones",
            "apiVersion": "2018-09-01",
            "location": "global",
            "properties": {
            }
        },
        {
            "type": "Microsoft.Network/privateDnsZones/virtualNetworkLinks",
            "apiVersion": "2018-09-01",
            "name": "[concat(parameters('privateZoneName'), '/myvnetlink')]",
            "location": "global",
            "dependsOn": [
                "[resourceId('Microsoft.Network/privateDnsZones', parameters('privateZoneName'))]"
            ],
            "properties": {
                "registrationEnabled": false,
                "virtualNetwork": {
                    "id": "[parameters('VNetId')]"
                }
            }
        }
    ]
}
```

Use the following code to create a Resource Manager template named "PrivateZoneRecords_template.json."

```
{
    "$schema": "http://schema.management.azure.com/schemas/2015-01-01/deploymentTemplate.json#",
    "contentVersion": "1.0.0.0",
    "parameters": {
        "DNSRecordName": {
            "type": "string"
        },
        "IPAddress": {
            "type":"string"
        }
    },
    "resources": [
         {
            "type": "Microsoft.Network/privateDnsZones/A",
            "apiVersion": "2018-09-01",
            "name": "[parameters('DNSRecordName')]",
            "properties": {
                "ttl": 300,
                "aRecords": [
                    {
                        "ipv4Address": "[parameters('IPAddress')]"
                    }
                ]
            }
        }
    ]
}
```

Define the parameters file for the template

Create the following two parameters file for the template. Create the "PrivateZone_parameters.json." with the following code:

```
{
    "$schema": "https://schema.management.azure.com/schemas/2015-01-01/deploymentParameters.json#",
    "contentVersion": "1.0.0.0",
    "parameters": {
        "privateZoneName": {
            "value": ""
        },
        "VNetId": {
            "value": ""
        }
    }
}
```

Create the "PrivateZoneRecords_parameters.json." with the following code:

```
{
    "$schema": "https://schema.management.azure.com/schemas/2015-01-01/deploymentParameters.json#",
    "contentVersion": "1.0.0.0",
    "parameters": {
        "DNSRecordName": {
            "value": ""
        },
        "IPAddress": {
            "type":"object"
        }
    }
}
```

Deploy the template by using a PowerShell script

Create a PowerShell script by using the following code. Before you run the script, replace the subscription ID, resource group name, and other variable values with the details for your environment.

```
### This script:
### - creates a private zone
### - creates a private endpoint for an existing Cosmos DB account in an existing VNet
### - maps the private endpoint to the private zone

## Step 1: Fill in these details. Replace the variable values with the details for your environment.
$SubscriptionId = "<your Azure subscription ID>"
# Resource group where the Azure Cosmos account and virtual network resources are located
$ResourceGroupName = "myResourceGroup"
# Name of the Azure Cosmos account
$CosmosDbAccountName = "mycosmosaccount"
# API type of the Azure Cosmos account. It can be one of the following: "Sql", "MongoDB", "Cassandra",
"Gremlin", "Table"
$CosmosDbApiType = "Sql"
# Name of the existing virtual network
$VNetName = "myVnet"
# Name of the target subnet in the virtual network
$SubnetName = "mySubnet"
# Name of the private zone to create
$PrivateZoneName = "myPrivateZone.documents.azure.com"
# Name of the private endpoint to create
$PrivateEndpointName = "myPrivateEndpoint"

$cosmosDbResourceId =
"/subscriptions/$($SubscriptionId)/resourceGroups/$($ResourceGroupName)/providers/Microsoft.DocumentDB/databas
eAccounts/$($CosmosDbAccountName)"
$VNetResourceId =
"/subscriptions/$($SubscriptionId)/resourceGroups/$($ResourceGroupName)/providers/Microsoft.Network/virtualNet
works/$($VNetName)"
$SubnetResourceId = "$($VNetResourceId)/subnets/$($SubnetName)"
$PrivateZoneTemplateFilePath = "PrivateZone_template.json"
$PrivateZoneParametersFilePath = "PrivateZone_parameters.json"
$PrivateZoneRecordsTemplateFilePath = "PrivateZoneRecords_template.json"
$PrivateZoneRecordsParametersFilePath = "PrivateZoneRecords_parameters.json"
$PrivateEndpointTemplateFilePath = "PrivateEndpoint_template.json"
$PrivateEndpointParametersFilePath = "PrivateEndpoint_parameters.json"

## Step 2: Login your Azure account and select the target subscription
Login-AzAccount
Select-AzSubscription -SubscriptionId $subscriptionId

## Step 3: Make sure private endpoint network policies are disabled in the subnet
$VirtualNetwork= Get-AzVirtualNetwork -Name "$VNetName" -ResourceGroupName "$ResourceGroupName"
($virtualNetwork | Select -ExpandProperty subnets | Where-Object  {$_.Name -eq "$SubnetName"}
).PrivateEndpointNetworkPolicies = "Disabled"
$virtualNetwork | Set-AzVirtualNetwork

## Step 4: Create the private zone
New-AzResourceGroupDeployment -Name "PrivateZoneDeployment" `
    -ResourceGroupName $ResourceGroupName `
    -TemplateFile $PrivateZoneTemplateFilePath `
    -TemplateParameterFile $PrivateZoneParametersFilePath `
    -PrivateZoneName $PrivateZoneName `
    -VNetId $VNetResourceId

## Step 5: Create the private endpoint
Write-Output "Deploying private endpoint on $($resourceGroupName)"
$deploymentOutput = New-AzResourceGroupDeployment -Name "PrivateCosmosDbEndpointDeployment" `
    -ResourceGroupName $resourceGroupName `
    -TemplateFile $PrivateEndpointTemplateFilePath `
    -TemplateParameterFile $PrivateEndpointParametersFilePath `
    -SubnetId $SubnetResourceId `
    -ResourceId $CosmosDbResourceId `
    -GroupId $CosmosDbApiType `
    -PrivateEndpointName $PrivateEndpointName
```

```
    -PrivateEndpointName $PrivateEndpointName
$deploymentOutput

## Step 6: Map the private endpoint to the private zone
$networkInterface = Get-AzResource -ResourceId $deploymentOutput.Outputs.privateEndpointNetworkInterface.Value
-ApiVersion "2019-04-01"
foreach ($ipconfig in $networkInterface.properties.ipConfigurations) {
    foreach ($fqdn in $ipconfig.properties.privateLinkConnectionProperties.fqdns) {
        $recordName = $fqdn.split('.',2)[0]
        $dnsZone = $fqdn.split('.',2)[1]
        Write-Output "Deploying PrivateEndpoint DNS Record $($PrivateZoneName)/$($recordName) Template on
$($resourceGroupName)"
        New-AzResourceGroupDeployment -Name "PrivateEndpointDNSDeployment" `
            -ResourceGroupName $ResourceGroupName `
            -TemplateFile $PrivateZoneRecordsTemplateFilePath `
            -TemplateParameterFile $PrivateZoneRecordsParametersFilePath `
            -DNSRecordName "$($PrivateZoneName)/$($RecordName)" `
            -IPAddress $ipconfig.properties.privateIPAddress
    }
}
```

Configure custom DNS

You should use a private DNS zone within the subnet where you've created the private endpoint. Configure the endpoints so that each private IP address is mapped to a DNS entry. (See the `fqdns` property in the response shown earlier.)

When you're creating the private endpoint, you can integrate it with a private DNS zone in Azure. If you choose to instead use a custom DNS zone, you have to configure it to add DNS records for all private IP addresses reserved for the private endpoint.

Private Link combined with firewall rules

The following situations and outcomes are possible when you use Private Link in combination with firewall rules:

- If you don't configure any firewall rules, then by default, all traffic can access an Azure Cosmos account.
- If you configure public traffic or a service endpoint and you create private endpoints, then different types of incoming traffic are authorized by the corresponding type of firewall rule. If a private endpoint is configured in a subnet where service endpoint is also configured:
  - traffic to the database account mapped by the private endpoint is routed via private endpoint,
  - traffic to other database accounts from the subnet is routed via service endpoint.
- If you don't configure any public traffic or service endpoint and you create private endpoints, then the Azure Cosmos account is accessible only through the private endpoints. If you don't configure public traffic or a service endpoint, after all approved private endpoints are rejected or deleted, the account is open to the entire network unless PublicNetworkAccess is set to Disabled (see section below).

Blocking public network access during account creation

As described in the previous section, and unless specific firewall rules have been set, adding a private endpoint makes your Azure Cosmos account accessible through private endpoints only. This means that the Azure Cosmos account could be reached from public traffic after it is created and before a private endpoint gets added. To make sure that public network access is disabled even before the creation of private endpoints, you can set the `publicNetworkAccess` flag to `Disabled` during account creation. Note that this flag takes precedence over any IP or virtual network rule; all public and virtual network traffic is blocked when the flag is set to `Disabled`, even if the source IP or virtual network is allowed in the firewall configuration.

See this Azure Resource Manager template for an example showing how to use this flag.

Adding private endpoints to an existing Cosmos account with no downtime

By default, adding a private endpoint to an existing account results in a short downtime of approximately 5 minutes. Follow the instructions below to avoid this downtime:

1. Add IP or virtual network rules to your firewall configuration to explicitly allow your client connections.
2. Wait for 10 minutes to ensure that the configuration update is applied.
3. Configure your new private endpoint.
4. Remove the firewall rules set at step 1.

Port range when using direct mode

When you're using Private Link with an Azure Cosmos account through a direct mode connection, you need to ensure that the full range of TCP ports (0 - 65535) is open.

Update a private endpoint when you add or remove a region

Unless you're using a private DNS zone group, adding or removing regions to an Azure Cosmos account requires you to add or remove DNS entries for that account. After regions have been added or removed, you can update the subnet's private DNS zone to reflect the added or removed DNS entries and their corresponding private IP addresses.

For example, imagine that you deploy an Azure Cosmos account in three regions: "West US," "Central US," and "West Europe." When you create a private endpoint for your account, four private IPs are reserved in the subnet. There's one IP for each of the three regions, and there's one IP for the global/region-agnostic endpoint.

Later, you might add a new region (for example, "East US") to the Azure Cosmos account. After adding the new region, you need to add a corresponding DNS record to either your private DNS zone or your custom DNS.

You can use the same steps when you remove a region. After removing the region, you need to remove the corresponding DNS record from either your private DNS zone or your custom DNS.

Current limitations

The following limitations apply when you're using Private Link with an Azure Cosmos account:

- You can't have more than 200 private endpoints on a single Azure Cosmos account.
- When you're using Private Link with an Azure Cosmos account through a direct mode connection, you can use only the TCP protocol. The HTTP protocol is not currently supported.
- When you're using Azure Cosmos DB's API for MongoDB accounts, a private endpoint is supported for accounts on server version 3.6 only (that is, accounts using the endpoint in the format `*.mongo.cosmos.azure.com`). Private Link is not supported for accounts on server version 3.2 (that is, accounts using the endpoint in the format `*.documents.azure.com`). To use Private Link, you should migrate old accounts to the new version.
- When you're using an Azure Cosmos DB's API for MongoDB account that has Private Link, some tools or libraries may not work as they automatically strip out the `appName` parameter from the connection string. This parameter is required to connect to the account over a private endpoint. Some tools, like Visual Studio Code, do not remove this parameter from the connection string and are therefore compatible.
- A network administrator should be granted at least the `Microsoft.DocumentDB/databaseAccounts/PrivateEndpointConnectionsApproval/action` permission at the Azure Cosmos account scope to create automatically approved private endpoints.

Limitations to private DNS zone integration

Unless you're using a private DNS zone group, DNS records in the private DNS zone are not removed automatically when you delete a private endpoint or you remove a region from the Azure Cosmos account. You must manually remove the DNS records before:

- Adding a new private endpoint linked to this private DNS zone.
- Adding a new region to any database account that has private endpoints linked to this private DNS zone.

If you don't clean up the DNS records, unexpected data plane issues might happen. These issues include data outage to regions added after private endpoint removal or region removal.

Next steps

To learn more about Azure Cosmos DB security features, see the following articles:

- To configure a firewall for Azure Cosmos DB, see Firewall support.
- To learn how to configure a virtual network service endpoint for your Azure Cosmos account, see Configure access from virtual networks.
- To learn more about Private Link, see the Azure Private Link documentation.

Tutorial: Connect to an Azure Cosmos account using an Azure Private Endpoint

12/4/2020 • 5 minutes to read • Edit Online

Azure Private endpoint is the fundamental building block for Private Link in Azure. It enables Azure resources, like virtual machines (VMs), to communicate with Private Link resources privately.

In this tutorial, you learn how to:

- Create a virtual network and bastion host.
- Create a virtual machine.
- Create a Cosmos DB account with a private endpoint.
- Test connectivity to Cosmos DB account private endpoint.

If you don't have an Azure subscription, create a free account before you begin.

Prerequisites

- An Azure subscription

Sign in to Azure

Sign in to the Azure portal.

Create a virtual network and bastion host

In this section, you'll create a virtual network, subnet, and bastion host.

The bastion host will be used to connect securely to the virtual machine for testing the private endpoint.

1. On the upper-left side of the screen, select **Create a resource** > **Networking** > **Virtual network** or search for **Virtual network** in the search box.
2. In **Create virtual network**, enter or select this information in the **Basics** tab:

SETTING	VALUE
Project Details	
Subscription	Select your Azure subscription
Resource Group	Select **myResourceGroup**
Instance details	
Name	Enter **myVNet**
Region	Select **East US**

3. Select the **IP Addresses** tab or select the **Next: IP Addresses** button at the bottom of the page.

4. In the **IP Addresses** tab, enter this information:

SETTING	VALUE
IPv4 address space	Enter **10.1.0.0/16**

5. Under **Subnet name**, select the word **default**.
6. In **Edit subnet**, enter this information:

SETTING	VALUE
Subnet name	Enter **mySubnet**
Subnet address range	Enter **10.1.0.0/24**

7. Select **Save**.
8. Select the **Security** tab.
9. Under **BastionHost**, select **Enable**. Enter this information:

SETTING	VALUE
Bastion name	Enter **myBastionHost**
AzureBastionSubnet address space	Enter **10.1.1.0/24**
Public IP Address	Select **Create new**. For **Name**, enter **myBastionIP**. Select **OK**.

10. Select the **Review + create** tab or select the **Review + create** button.
11. Select **Create**.

Create a virtual machine

In this section, you'll create a virtual machine that will be used to test the private endpoint.

1. On the upper-left side of the portal, select **Create a resource** > **Compute** > **Virtual machine** or search for **Virtual machine** in the search box.
2. In **Create a virtual machine**, type or select the values in the **Basics** tab:

SETTING	VALUE
Project Details	
Subscription	Select your Azure subscription
Resource Group	Select **myResourceGroup**
Instance details	
Virtual machine name	Enter **myVM**

SETTING	VALUE
Region	Select **East US**
Availability Options	Select **No infrastructure redundancy required**
Image	Select **Windows Server 2019 Datacenter - Gen1**
Azure Spot instance	Select **No**
Size	Choose VM size or take default setting
Administrator account	
Username	Enter a username
Password	Enter a password
Confirm password	Reenter password

3. Select the **Networking** tab, or select **Next: Disks**, then **Next: Networking**.
4. In the Networking tab, select or enter:

SETTING	VALUE
Network interface	
Virtual network	**myVNet**
Subnet	**mySubnet**
Public IP	Select **None**.
NIC network security group	**Basic**
Public inbound ports	Select **None**.

5. Select **Review + create**.
6. Review the settings, and then select **Create**.

Create a Cosmos DB account with a private endpoint

In this section, you'll create a Cosmos DB account and configure the private endpoint.

1. In the left-hand menu, select **Create a resource** > **Databases** > **Cosmos DB Account**, or search for **Cosmos DB account** in the search box.
2. In the **Basics** tab of **Create Cosmos DB account** enter or select the following information:

SETTING	VALUE
Project Details	

SETTING	VALUE
Subscription	Select your Azure subscription.
Resource Group	Select **myResourceGroup**.
Instance details	
Account name	Enter **mycosmosdb**. If the name is unavailable, enter a unique name.
API	Select **Core (SQL)**.
Location	Select **East US**.
Capacity mode	Leave the default **Provisioned throughput**.
Apply Free Tier Discount	Leave the default **Do Not Apply**.
Geo-Redundancy	Leave the default **Disable**.
Multi-region Writes	Leave the default **Disable**.

3. Select the **Networking** tab or select the **Next: Networking** button.
4. In the **Networking** tab, enter or select the following information:

SETTING	VALUE
Network connectivity	
Connectivity method	Select **Private endpoint**.
Configure Firewall	
Allow access from the Azure portal	Leave the default **Allow**.
Allow access from my IP	Leave the default **Deny**.

5. In **Private endpoint**, select **+ Add**.
6. In **Create private endpoint** enter or select the following information:

SETTING	VALUE
Subscription	Select your Azure subscription
Resource Group	Select **myResourceGroup**
Location	Select **East US**
Name	Enter **myPrivateEndpoint**

SETTING	VALUE
Target subresource	Leave the default **Core (SQL)**
Networking	
Virtual network	Select **myVNet**
Subnet	Select **mySubnet**
Private DNS integration	
Integrate with private DNS zone	Leave the default **Yes**
Private DNS Zone	Leave the default (New) privatelink.documents.azure.com

7. Select **OK**.
8. Select **Review + create**.
9. Select **Create**.

Add a database and a container

1. Select **Got to resource** or in the left-hand menu of the Azure portal, select **All Resources** > **mycosmosdb**.
2. In the left-hand menu, select **Data Explorer**.
3. In the **Data Explorer** window, select **New Container**.
4. In **Add Container**, enter or select the following information:

SETTING	VALUE
Database ID	Leave the default of **Create new**. Enter **mydatabaseid** in the text box.
Throughput (400 - 100,000 RU/s)	Leave the default of **Manual**. Enter **400** in the text box.
Container ID	Enter **mycontainerid**
Partition key	Enter **/mykey**

5. Select **OK**.
6. In the **Settings** section of the CosmosDB account, select **Keys**.
7. Select **myResourceGroup**.
8. Select the storage account you created in the previous steps.
9. Select copy on the **PRIMARY CONNECTION STRING**.

Test connectivity to private endpoint

In this section, you'll use the virtual machine you created in the previous step to connect to the Cosmos DB account

across the private endpoint.

1. Select **Resource groups** in the left-hand navigation pane.
2. Select **myResourceGroup**.
3. Select **myVM**.
4. On the overview page for **myVM**, select **Connect** then **Bastion**.
5. Select the blue **Use Bastion** button.
6. Enter the username and password that you entered during the virtual machine creation.
7. Open Windows PowerShell on the server after you connect.
8. Enter `nslookup <storage-account-name>.documents.azure.com`. Replace **<storage-account-name>** with the name of the storage account you created in the previous steps.

   ```
   Server:  UnKnown
   Address:  168.63.129.16

   Non-authoritative answer:
   Name:    mycosmosdb8675.privatelink.documents.azure.com
   Address:  10.1.0.5
   Aliases:  mycosmosdb8675.documents.azure.com
   ```

   A private IP address of **10.1.0.5** is returned for the Cosmos DB account name. This address is in the subnet of the virtual network you created previously.
9. Install Microsoft Azure Storage Explorer on the virtual machine.
10. Select **Finish** after the **Microsoft Azure Storage Explorer** is installed. Leave the box checked to open the application.
11. In the **Connect to Azure Storage** screen, select **Cancel**.
12. In Storage Explorer, select the right mouse button on **Cosmos DB Accounts** and select **Connect to Cosmos DB**.
13. Leave the default of **SQL** under **Select API**.
14. In the box under **Connection String**, paste the connection string from the Cosmos DB account you copied in the previous steps.
15. Select **Next**.
16. Verify the settings are correct in **Connection Summary**.
17. Select **Connect**.
18. Close the connection to **myVM**.

Clean up resources

If you're not going to continue to use this application, delete the virtual network, virtual machine, and Cosmos DB account with the following steps:

1. From the left-hand menu, select **Resource groups**.
2. Select **myResourceGroup**.

3. Select **Delete resource group**.
4. Enter **myResourceGroup** in **TYPE THE RESOURCE GROUP NAME**.
5. Select **Delete**.

Next steps

In this tutorial, you created a:

- Virtual network and bastion host.
- Virtual Machine.
- Cosmos DB Account.

Learn how to create a Private Link service:

Create a Private Link service

Secure Azure Cosmos keys using Azure Key Vault

12/4/2020 • 3 minutes to read • Edit Online

APPLIES TO: SQL API Cassandra API Gremlin API Table API Azure Cosmos DB API for MongoDB

> **IMPORTANT**
> The recommended solution to access Azure Cosmos DB keys is to use a system-assigned managed identity. If your service cannot take advantage of managed identities then use the cert based solution. If both the managed identity solution and cert based solution do not meet your needs, please use the key vault solution below.

When using Azure Cosmos DB for your applications, you can access the database, collections, documents by using the endpoint and the key within the app's configuration file. However, it's not safe to put keys and URL directly in the application code because they are available in clear text format to all the users. You want to make sure that the endpoint and keys are available but through a secured mechanism. This is where Azure Key Vault can help you to securely store and manage application secrets.

The following steps are required to store and read Azure Cosmos DB access keys from Key Vault:

- Create a Key Vault
- Add Azure Cosmos DB access keys to the Key Vault
- Create an Azure web application
- Register the application & grant permissions to read the Key Vault

Create a Key Vault

1. Sign in to Azure portal.
2. Select **Create a resource** > **Security** > **Key Vault**.
3. On the **Create key vault** section provide the following information:
   - **Name:** Provide a unique name for your Key Vault.
   - **Subscription:** Choose the subscription that you will use.
   - Under **Resource Group** choose **Create new** and enter a resource group name.
   - In the Location pull-down menu, choose a location.
   - Leave other options to their defaults.
4. After providing the information above, select **Create**.

Add Azure Cosmos DB access keys to the Key Vault.

1. Navigate to the Key Vault you created in the previous step, open the **Secrets** tab.
2. Select **+Generate/Import**,
   - Select **Manual** for **Upload options**.
   - Provide a **Name** for your secret
   - Provide the connection string of your Cosmos DB account into the **Value** field. And then select **Create**.

3. After the secret is created, open it and copy the **Secret Identifier that is in the following format. You will use this identifier in the next section.

```
https://<Key_Vault_Name>.vault.azure.net/secrets/<Secret _Name>/<ID>
```

Create an Azure web application

1. Create an Azure web application or you can download the app from the GitHub repository. It is a simple MVC application.

2. Unzip the downloaded application and open the **HomeController.cs** file. Update the secret ID in the following line:

```
var secret = await keyVaultClient.GetSecretAsync("<Your Key Vault's secret identifier>")
```

3. **Save** the file, **Build** the solution.

4. Next deploy the application to Azure. Right click on project and choose **publish**. Create a new app service profile (you can name the app WebAppKeyVault1) and select **Publish**.

5. Once the application is deployed. From the Azure portal, navigate to web app that you deployed, and turn on the **Managed service identity** of this application.

If you will run the application now, you will see the following error, as you have not given any permission to this application in Key Vault.

Register the application & grant permissions to read the Key Vault

In this section, you register the application with Azure Active Directory and give permissions for the application to read the Key Vault.

1. Navigate to the Azure portal, open the **Key Vault** you created in the previous section.
2. Open **Access policies**, select **+Add New** find the web app you deployed, select permissions and select **OK**.

Now, if you run the application, you can read the secret from Key Vault.

Similarly, you can add a user to access the key Vault. You need to add yourself to the Key Vault by selecting **Access Policies** and then grant all the permissions you need to run the application from Visual studio. When this application is running from your desktop, it takes your identity.

Next steps

- To configure a firewall for Azure Cosmos DB see firewall support article.
- To configure virtual network service endpoint, see secure access by using VNet service endpoint article.

Restrict user access to data operations in Azure Cosmos DB

11/2/2020 • 2 minutes to read • Edit Online

APPLIES TO: SQL API Cassandra API Gremlin API Table API Azure Cosmos DB API for MongoDB

In Azure Cosmos DB, there are two ways to authenticate your interactions with the database service:

- using your Azure Active Directory identity when interacting with the Azure portal,
- using Azure Cosmos DB keys or resource tokens when issuing calls from APIs and SDKs.

Each authentication method gives access to different sets of operations, with some overlap:

In some scenarios, you may want to restrict some users of your organization to perform data operations (that is CRUD requests and queries) only. This is typically the case for developers who don't need to create or delete resources, or change the provisioned throughput of the containers they are working on.

You can restrict the access by applying the following steps:

1. Creating a custom Azure Active Directory role for the users whom you want to restrict access. The custom Active Directory role should have fine-grained access level to operations using Azure Cosmos DB's granular actions.
2. Disallowing the execution of non-data operations with keys. You can achieve this by restricting these operations to Azure Resource Manager calls only.

The next sections of this article show how to perform these steps.

> **NOTE**
>
> In order to execute the commands in the next sections, you need to install Azure PowerShell Module 3.0.0 or later, as well as the Azure Owner Role on the subscription that you are trying to modify.

In the PowerShell scripts in the next sections, substitute the following placeholders with values specific to your environment:

- `$MySubscriptionId` - The subscription ID that contains the Azure Cosmos account where you want to limit the

permissions. For example: `e5c8766a-eeb0-40e8-af56-0eb142ebf78e`.

- `$MyResourceGroupName` - The resource group containing the Azure Cosmos account. For example: `myresourcegroup`.
- `$MyAzureCosmosDBAccountName` - The name of your Azure Cosmos account. For example: `mycosmosdbsaccount`.
- `$MyUserName` - The login (username@domain) of the user for whom you want to limit access. For example: `cosmosdbuser@contoso.com`.

Select your Azure subscription

Azure PowerShell commands require you to login and select the subscription to execute the commands:

```
Login-AzAccount
Select-AzSubscription $MySubscriptionId
```

Create the custom Azure Active Directory role

The following script creates an Azure Active Directory role assignment with "Key Only" access for Azure Cosmos accounts. The role is based on Azure custom roles and Granular actions for Azure Cosmos DB. These roles and actions are part of the `Microsoft.DocumentDB` Azure Active Directory namespace.

1. First, create a JSON document named `AzureCosmosKeyOnlyAccess.json` with the following content:

```
{
    "Name": "Azure Cosmos DB Key Only Access Custom Role",
    "Id": "00000000-0000-0000-0000-0000000000",
    "IsCustom": true,
    "Description": "This role restricts the user to read the account keys only.",
    "Actions":
    [
        "Microsoft.DocumentDB/databaseAccounts/listKeys/action"
    ],
    "NotActions": [],
    "DataActions": [],
    "NotDataActions": [],
    "AssignableScopes":
    [
        "/subscriptions/$MySubscriptionId"
    ]
}
```

2. Run the following commands to create the Role assignment and assign it to the user:

```
New-AzRoleDefinition -InputFile "AzureCosmosKeyOnlyAccess.json"
New-AzRoleAssignment -SignInName $MyUserName -RoleDefinitionName "Azure Cosmos DB Key Only Access Custom
Role" -ResourceGroupName $MyResourceGroupName -ResourceName $MyAzureCosmosDBAccountName -ResourceType
"Microsoft.DocumentDb/databaseAccounts"
```

Disallow the execution of non-data operations

The following commands remove the ability to use keys to:

- create, modify or delete resources
- update container settings (including indexing policies, throughput etc.).

```
$cdba = Get-AzResource -ResourceType "Microsoft.DocumentDb/databaseAccounts" -ApiVersion "2015-04-08" -
ResourceGroupName $MyResourceGroupName -ResourceName $MyAzureCosmosDBAccountName
$cdba.Properties.disableKeyBasedMetadataWriteAccess="True"
$cdba | Set-AzResource -Force
```

Next steps

- Learn more about Cosmos DB's role-based access control
- Get an overview of secure access to data in Cosmos DB

Configure customer-managed keys for your Azure Cosmos account with Azure Key Vault

12/4/2020 • 8 minutes to read • Edit Online

APPLIES TO: SQL API Cassandra API Gremlin API Table API Azure Cosmos DB API for MongoDB

Data stored in your Azure Cosmos account is automatically and seamlessly encrypted with keys managed by Microsoft (**service-managed keys**). Optionally, you can choose to add a second layer of encryption with keys you manage (**customer-managed keys**).

You must store customer-managed keys in Azure Key Vault and provide a key for each Azure Cosmos account that is enabled with customer-managed keys. This key is used to encrypt all the data stored in that account.

> **NOTE**
> Currently, customer-managed keys are available only for new Azure Cosmos accounts. You should configure them during account creation.

Register the Azure Cosmos DB resource provider for your Azure subscription

1. Sign in to the Azure portal, go to your Azure subscription, and select **Resource providers** under the **Settings** tab:

2. Search for the **Microsoft.DocumentDB** resource provider. Verify if the resource provider is already marked as registered. If not, choose the resource provider and select **Register**:

Configure your Azure Key Vault instance

Using customer-managed keys with Azure Cosmos DB requires you to set two properties on the Azure Key Vault instance that you plan to use to host your encryption keys: **Soft Delete** and **Purge Protection**.

If you create a new Azure Key Vault instance, enable these properties during creation:

If you're using an existing Azure Key Vault instance, you can verify that these properties are enabled by looking at the **Properties** section on the Azure portal. If any of these properties isn't enabled, see the "Enabling soft-delete" and "Enabling Purge Protection" sections in one of the following articles:

- How to use soft-delete with PowerShell
- How to use soft-delete with Azure CLI

Add an access policy to your Azure Key Vault instance

1. From the Azure portal, go to the Azure Key Vault instance that you plan to use to host your encryption keys. Select **Access Policies** from the left menu:

2. Select **+ Add Access Policy**.

3. Under the **Key permissions** drop-down menu, select **Get**, **Unwrap Key**, and **Wrap Key** permissions:

4. Under **Select principal**, select **None selected**. Then, search for **Azure Cosmos DB** principal and select it (to make it easier to find, you can also search by principal ID: `a232010e-820c-4083-83bb-3ace5fc29d0b` for any Azure region except Azure Government regions where the principal ID is

`57506a73-e302-42a9-b869-6f12d9ec29e9`). Finally, choose **Select** at the bottom. If the **Azure Cosmos DB** principal isn't in the list, you might need to re-register the **Microsoft.DocumentDB** resource provider as described in the Register the resource provider section of this article.

5. Select **Add** to add the new access policy.

6. Select **Save** on the Key Vault instance to save all changes.

Generate a key in Azure Key Vault

1. From the Azure portal, go the Azure Key Vault instance that you plan to use to host your encryption keys. Then, select **Keys** from the left menu:

2. Select **Generate/Import**, provide a name for the new key, and select an RSA key size. A minimum of 3072 is recommended for best security. Then select **Create**:

3. After the key is created, select the newly created key and then its current version.

4. Copy the key's **Key Identifier**, except the part after the last forward slash:

Create a new Azure Cosmos account

Using the Azure portal

When you create a new Azure Cosmos DB account from the Azure portal, choose **Customer-managed key** in the **Encryption** step. In the **Key URI** field, paste the URI/key identifier of the Azure Key Vault key that you copied from the previous step:

Using Azure PowerShell

When you create a new Azure Cosmos DB account with PowerShell:

- Pass the URI of the Azure Key Vault key copied earlier under the **keyVaultKeyUri** property in **PropertyObject**.
- Use **2019-12-12** or later as the API version.

> **IMPORTANT**
> You must set the `locations` property explicitly for the account to be successfully created with customer-managed keys.

```
$resourceGroupName = "myResourceGroup"
$accountLocation = "West US 2"
$accountName = "mycosmosaccount"

$failoverLocations = @(
    @{ "locationName"="West US 2"; "failoverPriority"=0 }
)

$CosmosDBProperties = @{
    "databaseAccountOfferType"="Standard";
    "locations"=$failoverLocations;
    "keyVaultKeyUri" = "https://<my-vault>.vault.azure.net/keys/<my-key>";
}

New-AzResource -ResourceType "Microsoft.DocumentDb/databaseAccounts" `
    -ApiVersion "2019-12-12" -ResourceGroupName $resourceGroupName `
    -Location $accountLocation -Name $accountName -PropertyObject $CosmosDBProperties
```

After the account has been created, you can verify that customer-managed keys have been enabled by fetching the URI of the Azure Key Vault key:

```
Get-AzResource -ResourceGroupName $resourceGroupName -Name $accountName `
    -ResourceType "Microsoft.DocumentDb/databaseAccounts" `
    | Select-Object -ExpandProperty Properties `
    | Select-Object - expandProperty keyVaultKeyUri
```

Using an Azure Resource Manager template

When you create a new Azure Cosmos account through an Azure Resource Manager template:

- Pass the URI of the Azure Key Vault key that you copied earlier under the **keyVaultKeyUri** property in the **properties** object.
- Use **2019-12-12** or later as the API version.

> **IMPORTANT**
>
> You must set the `locations` property explicitly for the account to be successfully created with customer-managed keys.

```
{
    "$schema": "https://schema.management.azure.com/schemas/2015-01-01/deploymentTemplate.json#",
    "contentVersion": "1.0.0.0",
    "parameters": {
        "accountName": {
            "type": "string"
        },
        "location": {
            "type": "string"
        },
        "keyVaultKeyUri": {
            "type": "string"
        }
    },
    "resources":
    [
        {
            "type": "Microsoft.DocumentDB/databaseAccounts",
            "name": "[parameters('accountName')]",
            "apiVersion": "2019-12-12",
            "kind": "GlobalDocumentDB",
            "location": "[parameters('location')]",
            "properties": {
                "locations": [
                    {
                        "locationName": "[parameters('location')]",
                        "failoverPriority": 0,
                        "isZoneRedundant": false
                    }
                ],
                "databaseAccountOfferType": "Standard",
                "keyVaultKeyUri": "[parameters('keyVaultKeyUri')]"
            }
        }
    ]
}
```

Deploy the template with the following PowerShell script:

```
$resourceGroupName = "myResourceGroup"
$accountName = "mycosmosaccount"
$accountLocation = "West US 2"
$keyVaultKeyUri = "https://<my-vault>.vault.azure.net/keys/<my-key>"

New-AzResourceGroupDeployment `
    -ResourceGroupName $resourceGroupName `
    -TemplateFile "deploy.json" `
    -accountName $accountName `
    -location $accountLocation `
    -keyVaultKeyUri $keyVaultKeyUri
```

Using the Azure CLI

When you create a new Azure Cosmos account through the Azure CLI, pass the URI of the Azure Key Vault key that you copied earlier under the `--key-uri` parameter.

```
resourceGroupName='myResourceGroup'
accountName='mycosmosaccount'
keyVaultKeyUri = 'https://<my-vault>.vault.azure.net/keys/<my-key>'

az cosmosdb create \
    -n $accountName \
    -g $resourceGroupName \
    --locations regionName='West US 2' failoverPriority=0 isZoneRedundant=False \
    --key-uri $keyVaultKeyUri
```

After the account has been created, you can verify that customer-managed keys have been enabled by fetching the URI of the Azure Key Vault key:

```
az cosmosdb show \
    -n $accountName \
    -g $resourceGroupName \
    --query keyVaultKeyUri
```

Key rotation

Rotating the customer-managed key used by your Azure Cosmos account can be done in two ways.

- Create a new version of the key currently used from Azure Key Vault:

- Swap the key currently used with a totally different one by updating the key URI on your account. From the Azure portal, go to your Azure Cosmos account and select **Data Encryption** from the left menu:

Then, replace the **Key URI** with the new key you want to use and select **Save**:

Here's how to do achieve the same result in PowerShell:

```
$resourceGroupName = "myResourceGroup"
$accountName = "mycosmosaccount"
$newKeyUri = "https://<my-vault>.vault.azure.net/keys/<my-new-key>"

$account = Get-AzResource -ResourceGroupName $resourceGroupName -Name $accountName `
    -ResourceType "Microsoft.DocumentDb/databaseAccounts"

$account.Properties.keyVaultKeyUri = $newKeyUri

$account | Set-AzResource -Force
```

The previous key or key version can be disabled after 24 hours, or after the Azure Key Vault audit logs don't show activity from Azure Cosmos DB on that key or key version anymore.

Error handling

When using Customer-Managed Keys (CMK) in Azure Cosmos DB, if there are any errors, Azure Cosmos DB returns the error details along with a HTTP sub-status code in the response. You can use this sub-status code to debug the root cause of the issue. See the HTTP Status Codes for Azure Cosmos DB article to get the list of supported HTTP sub-status codes.

Frequently asked questions

Is there an additional charge to enable customer-managed keys?

No, there's no charge to enable this feature.

How do customer-managed keys impact capacity planning?

When using customer-managed keys, Request Units consumed by your database operations see an increase to reflect the additional processing required to perform encryption and decryption of your data. This may lead to slightly higher utilization of your provisioned capacity. Use the table below for guidance:

OPERATION TYPE	REQUEST UNIT INCREASE
Point-reads (fetching items by their ID)	+ 5% per operation
Any write operation	+ 6% per operation approx. + 0.06 RU per indexed property
Queries, reading change feed, or conflict feed	+ 15% per operation

What data gets encrypted with the customer-managed keys?

All the data stored in your Azure Cosmos account is encrypted with the customer-managed keys, except for the following metadata:

- The names of your Azure Cosmos DB accounts, databases, and containers
- The names of your stored procedures
- The property paths declared in your indexing policies
- The values of your containers' partition keys

Are customer-managed keys supported for existing Azure Cosmos accounts?

This feature is currently available only for new accounts.

Is there a plan to support finer granularity than account-level keys?

Not currently, but container-level keys are being considered.

How can I tell if customer-managed keys are enabled on my Azure Cosmos account?

From the Azure portal, go to your Azure Cosmos account and watch for the **Data Encryption** entry in the left menu; if this entry exists, customer-managed keys are enabled on your account:

You can also programmatically fetch the details of your Azure Cosmos account and look for the presence of the `keyVaultKeyUri` property. See above for ways to do that in PowerShell and using the Azure CLI.

How do customer-managed keys affect a backup?

Azure Cosmos DB takes regular and automatic backups of the data stored in your account. This operation backs up the encrypted data. To use the restored backup, the encryption key that you used at the time of the backup is required. This means that no revocation was made and the version of the key that was used at the time of the backup will still be enabled.

How do I revoke an encryption key?

Key revocation is done by disabling the latest version of the key:

Alternatively, to revoke all keys from an Azure Key Vault instance, you can delete the access policy granted to the Azure Cosmos DB principal:

What operations are available after a customer-managed key is revoked?

The only operation possible when the encryption key has been revoked is account deletion.

Next steps

- Learn more about data encryption in Azure Cosmos DB.
- Get an overview of secure access to data in Cosmos DB.

Online backup and on-demand data restore in Azure Cosmos DB

11/2/2020 • 11 minutes to read • Edit Online

APPLIES TO: SQL API Cassandra API Gremlin API Table API Azure Cosmos DB API for MongoDB

Azure Cosmos DB automatically takes backups of your data at regular intervals. The automatic backups are taken without affecting the performance or availability of the database operations. All the backups are stored separately in a storage service, and those backups are globally replicated for resiliency against regional disasters. The automatic backups are helpful in scenarios when you accidentally delete or update your Azure Cosmos account, database, or container and later require the data recovery.

Automatic and online backups

With Azure Cosmos DB, not only your data, but also the backups of your data are highly redundant and resilient to regional disasters. The following steps show how Azure Cosmos DB performs data backup:

- Azure Cosmos DB automatically takes a full backup of your database every 4 hours and at any point of time, only the latest two backups are stored by default. If the default intervals aren't sufficient for your workloads, you can change the backup interval and the retention period from the Azure portal. You can change the backup configuration during or after the Azure Cosmos account is created. If the container or database is deleted, Azure Cosmos DB retains the existing snapshots of a given container or database for 30 days.
- Azure Cosmos DB stores these backups in Azure Blob storage whereas the actual data resides locally within Azure Cosmos DB.
- To guarantee low latency, the snapshot of your backup is stored in Azure Blob storage in the same region as the current write region (or **one** of the write regions, in case you have a multi-region write configuration). For resiliency against regional disaster, each snapshot of the backup data in Azure Blob storage is again replicated to another region through geo-redundant storage (GRS). The region to which the backup is replicated is based on your source region and the regional pair associated with the source region. To learn more, see the list of geo-redundant pairs of Azure regions article. You cannot access this backup directly. Azure Cosmos DB team will restore your backup when you request through a support request.

  The following image shows how an Azure Cosmos container with all the three primary physical partitions in West US is backed up in a remote Azure Blob Storage account in West US and then replicated to East US:

- The backups are taken without affecting the performance or availability of your application. Azure Cosmos DB performs data backup in the background without consuming any additional provisioned throughput (RUs) or affecting the performance and availability of your database.

Modify the backup interval and retention period

Azure Cosmos DB automatically takes a full backup of your data for every 4 hours and at any point of time, the latest two backups are stored. This configuration is the default option and it's offered without any additional cost. You can change the default backup interval and retention period during the Azure Cosmos account creation or after the account is created. The backup configuration is set at the Azure Cosmos account level and you need to configure it on each account. After you configure the backup options for an account, it's applied to all the containers within that account. Currently you can change them backup options from Azure portal only.

If you have accidentally deleted or corrupted your data, **before you create a support request to restore the data, make sure to increase the backup retention for your account to at least seven days. It's best to increase your retention within 8 hours of this event.** This way, the Azure Cosmos DB team has enough time to restore your account.

Use the following steps to change the default backup options for an existing Azure Cosmos account:

1. Sign into the Azure portal.

2. Navigate to your Azure Cosmos account and open the **Backup & Restore** pane. Update the backup interval and the backup retention period as required.

   - **Backup Interval** - It's the interval at which Azure Cosmos DB attempts to take a backup of your data. Backup takes a non-zero amount of time and in some case it could potentially fail due to downstream dependencies. Azure Cosmos DB tries its best to take a backup at the configured interval, however, it doesn't guarantee that the backup completes within that time interval. You can configure this value in hours or minutes. Backup Interval cannot be less than 1 hour and greater than 24 hours. When you change this interval, the new interval takes into effect starting from the time when the last backup was taken.

   - **Backup Retention** - It represents the period where each backup is retained. You can configure it in hours or days. The minimum retention period can't be less than two times the backup interval (in

hours) and it can't be greater than 720 hours.

- **Copies of data retained** - By default, two backup copies of your data are offered at free of charge. There is an additional charge if you need more than two copies. See the Consumed Storage section in the Pricing page to know the exact price for additional copies.

If you configure backup options during the account creation, you can configure the **Backup policy**, which is either **Periodic** or **Continuous**. The periodic policy allows you to configure the Backup interval and Backup retention. The continuous policy is currently available by sign-up only. The Azure Cosmos DB team will assess your workload and approve your request.

Request data restore from a backup

If you accidentally delete your database or a container, you can file a support ticket or call the Azure support to restore the data from automatic online backups. Azure support is available for selected plans only such as **Standard**, **Developer**, and plans higher than those. Azure support is not available with **Basic** plan. To learn

about different support plans, see the Azure support plans page.

To restore a specific snapshot of the backup, Azure Cosmos DB requires that the data is available for the duration of the backup cycle for that snapshot. You should have the following details before requesting a restore:

- Have your subscription ID ready.
- Based on how your data was accidentally deleted or modified, you should prepare to have additional information. It is advised that you have the information available ahead to minimize the back-and-forth that can be detrimental in some time sensitive cases.
- If the entire Azure Cosmos DB account is deleted, you need to provide the name of the deleted account. If you create another account with the same name as the deleted account, share that with the support team because it helps to determine the right account to choose. It's recommended to file different support tickets for each deleted account because it minimizes the confusion for the state of restore.
- If one or more databases are deleted, you should provide the Azure Cosmos account, as well as the Azure Cosmos database names and specify if a new database with the same name exists.
- If one or more containers are deleted, you should provide the Azure Cosmos account name, database names, and the container names. And specify if a container with the same name exists.
- If you have accidentally deleted or corrupted your data, you should contact Azure support within 8 hours so that the Azure Cosmos DB team can help you restore the data from the backups. **Before you create a support request to restore the data, make sure to increase the backup retention for your account to at least seven days. It's best to increase your retention within 8 hours of this event.** This way the Azure Cosmos DB support team will have enough time to restore your account.

In addition to Azure Cosmos account name, database names, container names, you should specify the point in time to which the data can be restored to. It is important to be as precise as possible to help us determine the best available backups at that time. **It is also important to specify the time in UTC.**

The following screenshot illustrates how to create a support request for a container(collection/graph/table) to restore data by using Azure portal. Provide additional details such as type of data, purpose of the restore, time when the data was deleted to help us prioritize the request.

Considerations for restoring the data from a backup

You may accidentally delete or modify your data in one of the following scenarios:

- Delete the entire Azure Cosmos account.
- Delete one or more Azure Cosmos databases.
- Delete one or more Azure Cosmos containers.
- Delete or modify the Azure Cosmos items (for example, documents) within a container. This specific case is typically referred to as data corruption.
- A shared offer database or containers within a shared offer database are deleted or corrupted.

Azure Cosmos DB can restore data in all the above scenarios. When restoring, a new Azure Cosmos account is created to hold the restored data. The name of the new account, if it's not specified, will have the format `<Azure_Cosmos_account_original_name>-restored1`. The last digit is incremented when multiple restores are attempted. You can't restore data to a pre-created Azure Cosmos account.

When you accidentally delete an Azure Cosmos account, we can restore the data into a new account with the same name, provided that the account name is not in use. So, we recommend that you don't re-create the account after deleting it. Because it not only prevents the restored data to use the same name, but also makes discovering the right account to restore from difficult.

When you accidentally delete an Azure Cosmos database, we can restore the whole database or a subset of the containers within that database. It is also possible to select specific containers across databases and restore them to a new Azure Cosmos account.

When you accidentally delete or modify one or more items within a container (the data corruption case), you need to specify the time to restore to. Time is important if there is data corruption. Because the container is live, the backup is still running, so if you wait beyond the retention period (the default is eight hours) the backups would be overwritten. In order to prevent the backup from being overwritten, increase the backup retention for your account to at least seven days. It's best to increase your retention within 8 hours from the data corruption.

If you have accidentally deleted or corrupted your data, you should contact Azure support within 8 hours so that the Azure Cosmos DB team can help you restore the data from the backups. This way the Azure Cosmos DB support team will have enough time to restore your account.

> **NOTE**
>
> After you restore the data, not all the source capabilities or settings are carried over to the restored account. The following settings are not carried over to the new account:
>
> - VNET access control lists
> - Stored procedures, triggers and user-defined functions
> - Multi-region settings

If you provision throughput at the database level, the backup and restore process in this case happen at the entire database level, and not at the individual containers level. In such cases, you can't select a subset of containers to restore.

Options to manage your own backups

With Azure Cosmos DB SQL API accounts, you can also maintain your own backups by using one of the following approaches:

- Use Azure Data Factory to move data periodically to a storage of your choice.
- Use Azure Cosmos DB change feed to read data periodically for full backups or for incremental changes, and store it in your own storage.

Post-restore actions

The primary goal of the data restore is to recover the data that you have accidentally deleted or modified. So, we recommend that you first inspect the content of the recovered data to ensure it contains what you are expecting. If everything looks good, you can migrate the data back to the primary account. Although it is possible to use the restored account as your new active account, it's not a recommended option if you have production workloads.

After you restore the data, you get a notification about the name of the new account (it's typically in the format `<original-name>-restored1`) and the time when the account was restored to. The restored account will have the same provisioned throughput, indexing policies and it is in same region as the original account. A user who is the subscription admin or a co-admin can see the restored account.

Migrate data to the original account

The following are different ways to migrate data back to the original account:

- Use the Azure Cosmos DB data migration tool.
- Use the Azure Data Factory.
- Use the change feed in Azure Cosmos DB.
- You can write your own custom code.

It is advised that you delete the container or database immediately after migrating the data. If you don't delete the restored databases or containers, they will incur cost for request units, storage, and egress.

Next steps

Next you can learn about how to restore data from an Azure Cosmos account or learn how to migrate data to an Azure Cosmos account

- To make a restore request, contact Azure Support, file a ticket from the Azure portal
- Use Cosmos DB change feed to move data to Azure Cosmos DB.
- Use Azure Data Factory to move data to Azure Cosmos DB.

How to request quota increase for Azure Cosmos DB resources

11/2/2020 • 2 minutes to read • Edit Online

APPLIES TO: SQL API Cassandra API Gremlin API Table API Azure Cosmos DB API for MongoDB

The resources in Azure Cosmos DB have default quotas/limits. However, there may be a case where your workload needs more quota than the default value. In such case, you must reach out to the Azure Cosmos DB team to request a quota increase. This article explains how to request a quota increase for Azure Cosmos DB resources. You will also learn how to enable a subscription to access a region.

Create a new support request

To request a quota increase, you must create a new support request with your workload details. The Azure Cosmos DB team will then evaluate your request and approve it. Use the following steps to create a new support request from the Azure portal:

1. Sign into the Azure portal.
2. From the left-hand menu, select **Help + support** and then select **New support request**.
3. In the **Basics** tab fill the following details:
   - For **Issue type**, select **Service and subscription limits (quotas)**
   - For **Subscription**, select the subscription for which you want to increase the quota.
   - For **Quota type**, select **Cosmos DB**

4. In the **Details** tab, enter the details corresponding to your quota request. The Information provided on this tab will be used to further assess your issue and help the support engineer troubleshoot the problem.
5. Fill the following details in this form:

- **Description**: Provide a short description of your request such as your workload, why the default values aren't sufficient. Based on the resource type for which you want quota increase, it's mandatory for you to provide the following details within the **Description** field:

  Regions requests If your request corresponds to adding a region to the allowed list, make sure to provide the following values:

  - Region name
  - Subscription ID

  Throughput limit requests If your request corresponds to increasing the quota for the throughput, make sure to provide the following values:

  - Database name
  - Subscription ID
  - New throughput limit

  Database account limit requests If your request corresponds to increasing the quota for the number of database accounts in a subscription, make sure to provide the following values:

  - Subscription ID
  - New database account limit

- **File upload**: Upload the diagnostic files or any other files that you think are relevant to the support request. To learn more on the file upload guidance, see the Azure support article.

- **Severity**: Choose one of the available severity levels based on the business impact.

- **Preferred contact method**: You can either choose to be contacted over **Email** or by **Phone**.

6. Fill out the remaining details such as your availability, support language, contact information, email, and phone number on the form.

7. Select **Next: Review + Create**. Validate the information provided and select **Create** to create a support request.

Within 24 hours, the Azure Cosmos DB support team will evaluate your request and get back to you.

Next steps

- See the Azure Cosmos DB default service quotas

Monitor Azure Cosmos DB

12/4/2020 • 10 minutes to read • Edit Online

APPLIES TO: SQL API Cassandra API Gremlin API Table API Azure Cosmos DB API for MongoDB

When you have critical applications and business processes relying on Azure resources, you want to monitor those resources for their availability, performance, and operation. This article describes the monitoring data generated by Azure Cosmos databases and how you can use the features of Azure Monitor to analyze and alert on this data.

You can monitor your data with client-side and server-side metrics. When using server-side metrics, you can monitor the data stored in Azure Cosmos DB with the following options:

- **Monitor from Azure Cosmos DB portal:** You can monitor with the metrics available within the **Metrics** tab of the Azure Cosmos account. The metrics on this tab include throughput, storage, availability, latency, consistency, and system level metrics. By default, these metrics have a retention period of 7 days. To learn more, see the Monitoring data collected from Azure Cosmos DB section of this article.
- **Monitor with metrics in Azure monitor:** You can monitor the metrics of your Azure Cosmos account and create dashboards from the Azure Monitor. Azure Monitor collects the Azure Cosmos DB metrics by default, you don't have configure anything explicitly. These metrics are collected with one-minute granularity, the granularity may vary based on the metric you choose. By default, these metrics have a retention period of 30 days. Most of the metrics that are available from the previous options are also available in these metrics. The dimension values for the metrics such as container name are case-insensitive. So you need to use case-insensitive comparison when doing string comparisons on these dimension values. To learn more, see the Analyze metric data section of this article.
- **Monitor with diagnostic logs in Azure Monitor:** You can monitor the logs of your Azure Cosmos account and create dashboards from the Azure Monitor. Telemetry such as events and traces that occur at a second granularity are stored as logs. For example, if the throughput of a container is changes, the properties of a Cosmos account are changed these events are captures within the logs. You can analyze these logs by running queries on the gathered data. To learn more, see the Analyze log data section of this article.
- **Monitor programmatically with SDKs:** You can monitor your Azure Cosmos account programmatically by using the .NET, Java, Python, Node.js SDKs, and the headers in REST API. To learn more, see the Monitoring Azure Cosmos DB programmatically section of this article.

The following image shows different options available to monitor Azure Cosmos DB account through Azure portal:

When using Azure Cosmos DB, at the client-side you can collect the details for request charge, activity ID, exception/stack trace information, HTTP status/sub-status code, diagnostic string to debug any issue that might occur. This information is also required if you need to reach out to the Azure Cosmos DB support team.

Monitor overview

The **Overview** page in the Azure portal for each Azure Cosmos DB account includes a brief view of the resource usage, such as total requests, requests that resulted in a specific HTTP status code, and hourly billing. This information is helpful, however only a small amount of the monitoring data is available from this pane. Some of this data is collected automatically and is available for analysis as soon as you create the resource. You can enable additional types of data collection with some configuration.

What is Azure Monitor?

Azure Cosmos DB creates monitoring data using Azure Monitor which is a full stack monitoring service in Azure that provides a complete set of features to monitor your Azure resources in addition to resources in other clouds and on-premises.

If you're not already familiar with monitoring Azure services, start with the article Monitoring Azure resources with Azure Monitor which describes the following concepts:

- What is Azure Monitor?
- Costs associated with monitoring
- Monitoring data collected in Azure
- Configuring data collection
- Standard tools in Azure for analyzing and alerting on monitoring data

The following sections build on this article by describing the specific data gathered from Azure Cosmos DB and providing examples for configuring data collection and analyzing this data with Azure tools.

Azure Monitor for Azure Cosmos DB

Azure Monitor for Azure Cosmos DB is based on the workbooks feature of Azure Monitor and uses the same monitoring data collected for Azure Cosmos DB described in the sections below. Use Azure Monitor for a view of the overall performance, failures, capacity, and operational health of all your Azure Cosmos DB resources in

a unified interactive experience, and leverage the other features of Azure Monitor for detailed analysis and alerting. To learn more, see the Explore Azure Monitor for Azure Cosmos DB article.

> **NOTE**
>
> When creating containers, make sure you don't create two containers with the same name but different casing. That's because some parts of the Azure platform are not case-sensitive, and this can result in confusion/collision of telemetry and actions on containers with such names.

Monitoring data

Azure Cosmos DB collects the same kinds of monitoring data as other Azure resources which are described in Monitoring data from Azure resources. See Azure Cosmos DB monitoring data reference for a detailed reference of the logs and metrics created by Azure Cosmos DB.

The **Overview** page in the Azure portal for each Azure Cosmos database includes a brief view of the database usage including its request and hourly billing usage. This is useful information but only a small amount of the monitoring data available. Some of this data is collected automatically and available for analysis as soon as you create the database while you can enable additional data collection with some configuration.

Collection and routing

Platform metrics and the Activity log are collected and stored automatically, but can be routed to other locations by using a diagnostic setting.

Resource Logs are not collected and stored until you create a diagnostic setting and route them to one or more locations.

See Create diagnostic setting to collect platform logs and metrics in Azure for the detailed process for creating a diagnostic setting using the Azure portal and some diagnostic query examples. When you create a diagnostic setting, you specify which categories of logs to collect.

The metrics and logs you can collect are discussed in the following sections.

Analyzing metrics

Azure Cosmos DB provides a custom experience for working with metrics. You can analyze metrics for Azure

Cosmos DB with metrics from other Azure services using Metrics explorer by opening **Metrics** from the **Azure Monitor** menu. See Getting started with Azure Metrics Explorer for details on using this tool. You can also checkout how to monitor server-side latency, request unit usage, and normalized request unit usage for your Azure Cosmos DB resources.

For a list of the platform metrics collected for Azure Cosmos DB, see Monitoring Azure Cosmos DB data reference metrics article.

All metrics for Azure Cosmos DB are in the namespace **Cosmos DB standard metrics**. You can use the following dimensions with these metrics when adding a filter to a chart:

- CollectionName
- DatabaseName
- OperationType
- Region
- StatusCode

For reference, you can see a list of all resource metrics supported in Azure Monitor.

View operation level metrics for Azure Cosmos DB

1. Sign in to the Azure portal.

2. Select **Monitor** from the left-hand navigation bar, and select **Metrics**.

3. From the **Metrics** pane > **Select a resource** > choose the required **subscription**, and **resource group**. For the **Resource type**, select **Azure Cosmos DB accounts**, choose one of your existing Azure Cosmos accounts, and select **Apply**.

4. Next you can select a metric from the list of available metrics. You can select metrics specific to request units, storage, latency, availability, Cassandra, and others. To learn in detail about all the available metrics in this list, see the Metrics by category article. In this example, let's select **Request units** and **Avg** as the aggregation value.

   In addition to these details, you can also select the **Time range** and **Time granularity** of the metrics. At max, you can view metrics for the past 30 days. After you apply the filter, a chart is displayed based on your filter. You can see the average number of request units consumed per minute for the selected period.

Add filters to metrics

You can also filter metrics and the chart displayed by a specific **CollectionName**, **DatabaseName**, **OperationType**, **Region**, and **StatusCode**. To filter the metrics, select **Add filter** and choose the required property such as **OperationType** and select a value such as **Query**. The graph then displays the request units consumed for the query operation for the selected period. The operations executed via Stored procedure are not logged so they are not available under the OperationType metric.

You can group metrics by using the **Apply splitting** option. For example, you can group the request units per operation type and view the graph for all the operations at once as shown in the following image:

Analyzing logs

Data in Azure Monitor Logs is stored in tables which each table having its own set of unique properties.

All resource logs in Azure Monitor have the same fields followed by service-specific fields. The common schema is outlined in Azure Monitor resource log schema. For a list of the types of resource logs collected for Azure Cosmos DB, see Monitoring Azure Cosmos DB data reference.

The Activity log is a platform login Azure that provides insight into subscription-level events. You can view it independently or route it to Azure Monitor Logs, where you can do much more complex queries using Log Analytics.

Azure Cosmos DB stores data in the following tables.

TABLE	DESCRIPTION
AzureDiagnostics	Common table used by multiple services to store Resource logs. Resource logs from Azure Cosmos DB can be identified with `MICROSOFT.DOCUMENTDB`.
AzureActivity	Common table that stores all records from the Activity log.

Sample Kusto queries

> **IMPORTANT**
>
> When you select **Logs** from the Azure Cosmos DB menu, Log Analytics is opened with the query scope set to the current Azure Cosmos DB account. This means that log queries will only include data from that resource. If you want to run a query that includes data from other accounts or data from other Azure services, select **Logs** from the **Azure Monitor** menu. See Log query scope and time range in Azure Monitor Log Analytics for details.

Here are some queries that you can enter into the **Log search** search bar to help you monitor your Azure Cosmos resources. These queries work with the new language.

- To query for all of the diagnostic logs from Azure Cosmos DB for a specified time period:

```
AzureDiagnostics
| where ResourceProvider=="Microsoft.DocumentDb" and Category=="DataPlaneRequests"
```

- To query for all operations, grouped by resource:

```
AzureActivity
| where ResourceProvider=="Microsoft.DocumentDb" and Category=="DataPlaneRequests"
| summarize count() by Resource
```

- To query for all user activity, grouped by resource:

```
AzureActivity
| where Caller == "test@company.com" and ResourceProvider=="Microsoft.DocumentDb" and
Category=="DataPlaneRequests"
| summarize count() by Resource
```

Alerts

Azure Monitor alerts proactively notify you when important conditions are found in your monitoring data. They allow you to identify and address issues in your system before your customers notice them. You can set alerts on metrics, logs, and the activity log. Different types of alerts have benefits and drawbacks

For example, the following table lists few alert rules for your resources. You can find a detailed list of alert rules from the Azure portal. To learn more, see how to configure alerts article.

ALERT TYPE	CONDITION	DESCRIPTION
Rate limiting on request units (metric alert)	Dimension name: StatusCode, Operator: Equals, Dimension values: 429	Alerts if the container or a database has exceeded the provisioned throughput limit.
Region failed over	Operator: Greater than, Aggregation type: Count, Threshold value: 1	When a single region is failed over. This alert is helpful if you didn't enable automatic failover.
Rotate keys(activity log alert)	Event level: Informational , Status: started	Alerts when the account keys are rotated. You can update your application with the new keys.

Monitor Azure Cosmos DB programmatically

The account level metrics available in the portal, such as account storage usage and total requests, are not available via the SQL APIs. However, you can retrieve usage data at the collection level by using the SQL APIs. To retrieve collection level data, do the following:

- To use the REST API, perform a GET on the collection. The quota and usage information for the collection is returned in the x-ms-resource-quota and x-ms-resource-usage headers in the response.
- To use the .NET SDK, use the DocumentClient.ReadDocumentCollectionAsync method, which returns a ResourceResponse that contains a number of usage properties such as **CollectionSizeUsage**, **DatabaseUsage**, **DocumentUsage**, and more.

To access additional metrics, use the Azure Monitor SDK. Available metric definitions can be retrieved by calling:

```
https://management.azure.com/subscriptions/{SubscriptionId}/resourceGroups/{ResourceGroup}/providers/Micro
soft.DocumentDb/databaseAccounts/{DocumentDBAccountName}/providers/microsoft.insights/metricDefinitions?
api-version=2018-01-01
```

To retrieve individual metrics use the following format:

```
https://management.azure.com/subscriptions/{SubscriptionId}/resourceGroups/{ResourceGroup}/providers/Micro
soft.DocumentDb/databaseAccounts/{DocumentDBAccountName}/providers/microsoft.insights/metrics?timespan=
{StartTime}/{EndTime}&interval={AggregationInterval}&metricnames={MetricName}&aggregation=
{AggregationType}&`$filter={Filter}&api-version=2018-01-01
```

To learn more, see the Azure monitoring REST API article.

Next steps

- See Azure Cosmos DB monitoring data reference for a reference of the logs and metrics created by Azure Cosmos DB.
- See Monitoring Azure resources with Azure Monitor for details on monitoring Azure resources.

Explore Azure Monitor for Azure Cosmos DB

11/2/2020 • 4 minutes to read • Edit Online

Azure Monitor for Azure Cosmos DB provides a view of the overall performance, failures, capacity, and operational health of all your Azure Cosmos DB resources in a unified interactive experience. This article will help you understand the benefits of this new monitoring experience, and how you can modify and adapt the experience to fit the unique needs of your organization.

Introduction

Before diving into the experience, you should understand how it presents and visualizes information.

It delivers:

- **At scale perspective** of your Azure Cosmos DB resources across all your subscriptions in a single location, with the ability to selectively scope to only those subscriptions and resources you are interested in evaluating.
- **Drill down analysis** of a particular Azure CosmosDB resource to help diagnose issues or perform detailed analysis by category - utilization, failures, capacity, and operations. Selecting any one of those options provides an in-depth view of the relevant Azure Cosmos DB metrics.
- **Customizable** - This experience is built on top of Azure Monitor workbook templates allowing you to change what metrics are displayed, modify or set thresholds that align with your limits, and then save into a custom workbook. Charts in the workbooks can then be pinned to Azure dashboards.

This feature does not require you to enable or configure anything, these Azure Cosmos DB metrics are collected by default.

> **NOTE**
> There is no charge to access this feature and you will only be charged for the Azure Monitor essential features you configure or enable, as described on the Azure Monitor pricing details page.

View utilization and performance metrics for Azure Cosmos DB

To view the utilization and performance of your storage accounts across all of your subscriptions, perform the following steps.

1. Sign in to the Azure portal.
2. Search for **Monitor** and select **Monitor**.

3. Select **Cosmos DB**.

Overview

On **Overview**, the table displays interactive Azure Cosmos DB metrics. You can filter the results based on the options you select from the following drop-down lists:

- **Subscriptions** - only subscriptions that have an Azure Cosmos DB resource are listed.
- **Cosmos DB** - You can select all, a subset, or single Azure Cosmos DB resource.
- **Time Range** - by default, displays the last 4 hours of information based on the corresponding selections made.

The counter tile under the drop-down lists rolls-up the total number of Azure Cosmos DB resources are in the selected subscriptions. There is conditional color-coding or heatmaps for columns in the workbook that report transaction metrics. The deepest color has the highest value and a lighter color is based on the lowest values.

Selecting a drop-down arrow next to one of the Azure Cosmos DB resources will reveal a breakdown of the performance metrics at the individual database container level:

Selecting the Azure Cosmos DB resource name highlighted in blue will take you to the default **Overview** for the associated Azure Cosmos DB account.

Failures

Select **Failures** at the top of the page and the **Failures** portion of the workbook template opens. It shows you total requests with the distribution of responses that make up those requests:

CODE	DESCRIPTION
`200 OK`	One of the following REST operations were successful: - GET on a resource. - PUT on a resource. - POST on a resource. - POST on a stored procedure resource to execute the stored procedure.
`201 Created`	A POST operation to create a resource is successful.
`404 Not Found`	The operation is attempting to act on a resource that no longer exists. For example, the resource may have already been deleted.

For a full list of status codes, consult the Azure Cosmos DB HTTP status code article.

Capacity

Select **Capacity** at the top of the page and the **Capacity** portion of the workbook template opens. It shows you how many documents you have, your document growth over time, data usage, and the total amount of available storage that you have left. This can be used to help identify potential storage and data utilization issues.

As with the overview workbook, selecting the drop-down next to an Azure Cosmos DB resource in the **Subscription** column will reveal a breakdown by the individual containers that make up the database.

Operations

Select **Operations** at the top of the page and the **Operations** portion of the workbook template opens. It gives you the ability to see your requests broken down by the type of requests made.

So in the example below you see that `eastus-billingint` is predominantly receiving read requests, but with a small number of upsert and create requests. Whereas `westeurope-billingint` is read-only from a request perspective, at least over the past four hours that the workbook is currently scoped to via its time range parameter.

Pin, export, and expand

You can pin any one of the metric sections to an Azure Dashboard by selecting the pushpin icon at the top right of the section.

To export your data into the Excel format, select the down arrow icon to the left of the pushpin icon.

To expand or collapse all drop-down views in the workbook, select the expand icon to the left of the export icon:

Customize Azure Monitor for Azure Cosmos DB

Since this experience is built on top of Azure Monitor workbook templates, you have the ability to **Customize** > **Edit** and **Save** a copy of your modified version into a custom workbook.

Workbooks are saved within a resource group, either in the **My Reports** section that's private to you or in the **Shared Reports** section that's accessible to everyone with access to the resource group. After you save the custom workbook, you need to go to the workbook gallery to launch it.

Troubleshooting

For troubleshooting guidance, refer to the dedicated workbook-based insights troubleshooting article.

Next steps

- Configure metric alerts and service health notifications to set up automated alerting to aid in detecting issues.
- Learn the scenarios workbooks are designed to support, how to author new and customize existing reports, and more by reviewing Create interactive reports with Azure Monitor workbooks.

Monitor and debug with metrics in Azure Cosmos DB

12/4/2020 • 5 minutes to read • Edit Online

APPLIES TO: SQL API Cassandra API Gremlin API Table API Azure Cosmos DB API for MongoDB

Azure Cosmos DB provides metrics for throughput, storage, consistency, availability, and latency. The Azure portal provides an aggregated view of these metrics. You can also view Azure Cosmos DB metrics from Azure Monitor API. The dimension values for the metrics such as container name are case-insensitive. So you need to use case-insensitive comparison when doing string comparisons on these dimension values. To learn about how to view metrics from Azure monitor, see the Get metrics from Azure Monitor article.

This article walks through common use cases and how Azure Cosmos DB metrics can be used to analyze and debug these issues. Metrics are collected every five minutes and are kept for seven days.

View metrics from Azure portal

1. Sign into Azure portal
2. Open the **Metrics** pane. By default, the metrics pane shows the storage, index, request units metrics for all the databases in your Azure Cosmos account. You can filter these metrics per database, container, or a region. You can also filter the metrics at a specific time granularity. More details on the throughput, storage, availability, latency, and consistency metrics are provided on separate tabs.

The following metrics are available from the **Metrics** pane:

- **Throughput metrics** - This metric shows the number of requests consumed or failed (429 response code) because the throughput or storage capacity provisioned for the container has exceeded.

- **Storage metrics** - This metric shows the size of data and index usage.
- **Availability metrics** - This metric shows the percentage of successful requests over the total requests per hour. The success rate is defined by the Azure Cosmos DB SLAs.
- **Latency metrics** - This metric shows the read and write latency observed by Azure Cosmos DB in the region where your account is operating. You can visualize latency across regions for a geo-replicated account. This metric doesn't represent the end-to-end request latency.
- **Consistency metrics** - This metric shows how eventual is the consistency for the consistency model you choose. For multi-region accounts, this metric also shows the replication latency between the regions you have selected.
- **System metrics** - This metric shows how many metadata requests are served by the primary partition. It also helps to identify the throttled requests.

The following sections explain common scenarios where you can use Azure Cosmos DB metrics.

Understand how many requests are succeeding or causing errors

To get started, head to the Azure portal and navigate to the **Metrics** blade. In the blade, find the **Number of requests exceeded capacity per 1-minute chart. This chart shows a minute by minute total requests segmented by the status code. For more information about HTTP status codes, see HTTP status codes for Azure Cosmos DB.

The most common error status code is 429 (rate limiting/throttling). This error means that requests to Azure Cosmos DB are more than the provisioned throughput. The most common solution to this problem is to scale up the RUs for the given collection.

Determine the throughput distribution across partitions

Having a good cardinality of your partition keys is essential for any scalable application. To determine the throughput distribution of any partitioned container broken down by partitions, navigate to the **Metrics blade** in the Azure portal. In the **Throughput** tab, the storage breakdown is shown in the **Max consumed RU/second by each physical partition** chart. The following graphic illustrates an example of a poor distribution of data as shown by the skewed partition on the far left.

An uneven throughput distribution may cause *hot* partitions, which can result in throttled requests and may require repartitioning. For more information about partitioning in Azure Cosmos DB, see Partition and scale in Azure Cosmos DB.

Determine the storage distribution across partitions

Having a good cardinality of your partition is essential for any scalable application. To determine the storage distribution of any partitioned container broken down by partitions, head to the Metrics blade in the Azure portal. In the Storage tab, the storage breakdown is shown in the Data + Index storage consumed by top partition keys chart. The following graphic illustrates a poor distribution of data storage as shown by the skewed partition on the far left.

You can root cause which partition key is skewing the distribution by clicking on the partition in the chart.

After identifying which partition key is causing the skew in distribution, you may have to repartition your

container with a more distributed partition key. For more information about partitioning in Azure Cosmos DB, see Partition and scale in Azure Cosmos DB.

Compare data size against index size

In Azure Cosmos DB, the total consumed storage is the combination of both the Data size and Index size. Typically, the index size is a fraction of the data size. In the Metrics blade in the Azure portal, the Storage tab showcases the breakdown of storage consumption based on data and index.

```
// Measure the document size usage (which includes the index size)
ResourceResponse<DocumentCollection> collectionInfo = await
client.ReadDocumentCollectionAsync(UriFactory.CreateDocumentCollectionUri("db", "coll"));
 Console.WriteLine("Document size quota: {0}, usage: {1}", collectionInfo.DocumentQuota,
collectionInfo.DocumentUsage);
```

If you would like to conserve index space, you can adjust the indexing policy.

Debug why queries are running slow

In the SQL API SDKs, Azure Cosmos DB provides query execution statistics.

```
IDocumentQuery<dynamic> query = client.CreateDocumentQuery(
 UriFactory.CreateDocumentCollectionUri(DatabaseName, CollectionName),
 "SELECT * FROM c WHERE c.city = 'Seattle'",
 new FeedOptions
 {
 PopulateQueryMetrics = true,
 MaxItemCount = -1,
 MaxDegreeOfParallelism = -1,
 EnableCrossPartitionQuery = true
 }).AsDocumentQuery();
FeedResponse<dynamic> result = await query.ExecuteNextAsync();

// Returns metrics by partition key range Id
IReadOnlyDictionary<string, QueryMetrics> metrics = result.QueryMetrics;
```

QueryMetrics provides details on how long each component of the query took to execution. The most common root cause for long running queries is scans, meaning the query was unable to leverage the indexes. This problem can be resolved with a better filter condition.

Next steps

You've now learned how to monitor and debug issues using the metrics provided in the Azure portal. You may want to learn more about improving database performance by reading the following articles:

- To learn about how to view metrics from Azure monitor, see the Get metrics from Azure Monitor article.
- Performance and scale testing with Azure Cosmos DB
- Performance tips for Azure Cosmos DB

How to monitor the server-side latency for operations in an Azure Cosmos DB container or account

11/2/2020 • 2 minutes to read • Edit Online

APPLIES TO: SQL API Cassandra API Gremlin API Table API Azure Cosmos DB API for MongoDB

Azure Monitor for Azure Cosmos DB provides a metrics view to monitor your account and create dashboards. The Azure Cosmos DB metrics are collected by default, this feature does not require you to enable or configure anything explicitly. The server-side latency metric is used to view the server-side latency of an operation. Azure Cosmos DB provides SLA of less than 10 ms for point read/write operations with direct connectivity. For point read and write operations, the SLAs are calculated as detailed in the SLA document.

You can monitor server-side latency if you see unusually high latency for point operation such as:

- A GET or a SET operation with partition key and ID in direct connectivity mode
- A read or write operation or
- A query

You can look up the diagnostic log to see the size of the data returned. If you see a sustained high latency for query operations, you should look up the diagnostic log for higher throughput or RU/s used. Server side latency shows the amount of time spent on the backend infrastructure before the data was returned to the client. It is important to look at this metric to rule out any backend latency issues.

View the server-side latency metric

1. Sign in to the Azure portal.

2. Select **Monitor** from the left-hand navigation bar and select **Metrics**.

3. From the **Metrics** pane > **Select a resource** > choose the required **subscription**, and **resource group**. For the **Resource type**, select **Azure Cosmos DB accounts**, choose one of your existing Azure Cosmos accounts, and select **Apply**.

4. Next select the **Server Side Latency** metric from the list of available metrics. To learn in detail about all the available metrics in this list, see the Metrics by category article. In this example, let's select **Server Side Latency** and **Avg** as the aggregation value. In addition to these details, you can also select the **Time range** and **Time granularity** of the metrics. At max, you can view metrics for the past 30 days. After you apply the filter, a chart is displayed based on your filter. You can see the server-side latency per minute for the selected period.

Filters for server-side latency

You can also filter metrics and get the charts displayed by a specific **CollectionName**, **ConnectionMode**, **DatabaseName**, **OperationType**, **Region**, and **PublicAPIType**.

To filter the metrics, select **Add filter** and choose the required property such as **PublicAPIType** and select the value **sql**. Add another filter for **OperationType**. The graph then displays the server-side latency for different operations during the selected period. The operations executed via Stored procedure are not logged so they are not available under the OperationType metric.

The **Server Side Latency** metrics for each operation are displayed as shown in the following image:

You can also group the metrics by using the **Apply splitting** option.

Next steps

- Monitor Azure Cosmos DB data by using diagnostic settings in Azure.
- Audit Azure Cosmos DB control plane operations

How to monitor throughput or request unit usage of an operation in Azure Cosmos DB

11/2/2020 • 2 minutes to read • Edit Online

APPLIES TO: SQL API Cassandra API Gremlin API Table API Azure Cosmos DB API for MongoDB

Azure Monitor for Azure Cosmos DB provides a metrics view to monitor your account and create dashboards. The Azure Cosmos DB metrics are collected by default, this feature does not require you to enable or configure anything explicitly. The **Total Request Units** metric is used to get the request units usage for different types of operations. Later you can analyze which operations used most of the throughput. By default, the throughput data is aggregated at one-minute interval. However, you can change the aggregation unit by changing the time granularity option.

There are two ways to analyze the request unit usage data:

- Within the given time interval which operations are taking more request units.
- Which operations in general dominate your workload by consuming more request units. This analysis allows you to focus on operations such as insert, upsert and look at their indexing. You can find out if you are over/under indexing specific fields and modify the indexing policy to include or exclude the paths.

If you notice certain queries are taking more request units, you can take actions such as:

- Reconsider if you are requesting the right amount of data.
- Modify the query to use index with filter clause.
- Perform less expensive UDF function calls.
- Define partition keys to minimize the fan out of query into different partitions.
- You can also use the query metrics returned in the call response, the diagnostic log details and refer to query performance tuning article to learn more about the query execution.
- You can start from sum and then look at avg utilization using the right dimension.

View the total request unit usage metric

1. Sign in to the Azure portal.

2. Select **Monitor** from the left-hand navigation bar and select **Metrics**.

3. From the **Metrics** pane > **Select a resource** > choose the required **subscription**, and **resource group**. For the **Resource type**, select **Azure Cosmos DB accounts**, choose one of your existing Azure Cosmos accounts, and select **Apply**.

4. Next select the **Total Request Units** metric from the list of available metrics. To learn in detail about all the available metrics in this list, see the Metrics by category article. In this example, let's select **Total Request Units** and **Avg** as the aggregation value. In addition to these details, you can also select the **Time range** and **Time granularity** of the metrics. At max, you can view metrics for the past 30 days. After you apply the filter, a chart is displayed based on your filter. You can see the average number of request units consumed per minute for the selected period.

Filters for request unit usage

You can also filter metrics and get the charts displayed by a specific **CollectionName**, **DatabaseName**, **OperationType**, **Region**, **Status**, and **StatusCode**. The **Add filter** and **Apply splitting** options allows you to filter the request unit usage and group the metrics.

To get the request unit usage of each operation either by total(sum) or average, select **Apply splitting** and choose **Operation type** and the filter value as shown in the following image:

If you want to see the request unit usage by collection, select **Apply splitting** and choose the collection name as a filter. You will see a chat like the following with a choice of collections within the dashboard. You can then select a specific collection name to view more details:

Next steps

- Monitor Azure Cosmos DB data by using diagnostic settings in Azure.
- Audit Azure Cosmos DB control plane operations

How to monitor normalized RU/s for an Azure Cosmos container or an account

11/2/2020 • 4 minutes to read • Edit Online

APPLIES TO: SQL API Cassandra API Gremlin API Table API Azure Cosmos DB API for MongoDB

Azure Monitor for Azure Cosmos DB provides a metrics view to monitor your account and create dashboards. The Azure Cosmos DB metrics are collected by default, this feature does not require you to enable or configure anything explicitly.

The **Normalized RU Consumption** metric is used to see how well saturated the partition key ranges are with respect to the traffic. Azure Cosmos DB distributes the throughput equally across all the partition key ranges. This metric provides a per second view of the maximum throughput utilization for partition key range. Use this metric to calculate the RU/s usage across partition key range for given container. By using this metric, if you see high percentage of request units utilization across all partition key ranges in Azure monitor, you should increase the throughput to meet the needs of your workload. Example - Normalized utilization is defined as the max of the RU/s utilization across all partition key ranges. For example, suppose your max throughput is 20,000 RU/s and you have two partition key ranges, P_1 and P_2, each capable of scaling to 10,000 RU/s. In a given second, if P_1 has used 6000 RUs, and P_2 8000 RUs, the normalized utilization is MAX(6000 RU / 10,000 RU, 8000 RU / 10,000 RU) = 0.8.

What to expect and do when normalized RU/s is higher

When the normalized RU/s consumption reaches 100% for given partition key range, and if a client still makes requests in that time window of 1 second to that specific partition key range - it receives a rate limited error. The client should respect the suggested wait time and retry the request. The SDK makes it easy to handle this situation by retrying preconfigured times by waiting appropriately. It is not necessary that you see the RU rate limiting error just because the normalized RU has reached 100%. That's because the normalized RU is a single value that represents the max usage over all partition key ranges, one partition key range may be busy but the other partition key ranges can serve the requests without issues. For example, a single operation such as a stored procedure that consumes all the RU/s on a partition key range will lead to a short spike in the normalized RU/s consumption. In such cases, there will not be any immediate rate limiting errors if the request rate is low or requests are made to other partitions on different partition key ranges.

The Azure Monitor metrics help you to find the operations per status code for SQL API by using the **Total Requests** metric. Later you can filter on these requests by the 429 status code and split them by **Operation Type**.

To find the requests which are rate limited, the recommended way is to get this information through diagnostic logs.

If there is continuous peak of 100% normalized RU/s consumption or close to 100% across multiple partition key ranges it's recommended to increase the throughput. You can find out which operations are heavy and their peak usage by utilizing the Azure monitor metrics and Azure monitor diagnostic logs.

In summary, the **Normalized RU Consumption** metric is used to see which partition key range is more warm in terms of usage. So it gives you the skew of throughput towards a partition key range. You can later follow up to see the **PartitionKeyRUConsumption** log in Azure Monitor logs to get information about which logical partition keys are hot in terms of usage. This will point to change in either the partition key choice, or the change

in application logic. To resolve the rate limiting, distribute the load of data say across multiple partitions or just increase in the throughput as it is really required.

View the normalized request unit consumption metric

1. Sign in to the Azure portal.

2. Select **Monitor** from the left-hand navigation bar, and select **Metrics**.

3. From the **Metrics** pane > **Select a resource** > choose the required **subscription**, and **resource group**. For the **Resource type**, select **Azure Cosmos DB accounts**, choose one of your existing Azure Cosmos accounts, and select **Apply**.

4. Next you can select a metric from the list of available metrics. You can select metrics specific to request units, storage, latency, availability, Cassandra, and others. To learn in detail about all the available metrics in this list, see the Metrics by category article. In this example, let's select **Normalized RU Consumption** metric and **Max** as the aggregation value.

   In addition to these details, you can also select the **Time range** and **Time granularity** of the metrics. At max, you can view metrics for the past 30 days. After you apply the filter, a chart is displayed based on your filter.

Filters for normalized request unit consumption

You can also filter metrics and the chart displayed by a specific **CollectionName**, **DatabaseName**, **PartitionKeyRangeID**, and **Region**. To filter the metrics, select **Add filter** and choose the required property such as **CollectionName** and corresponding value you are interested in. The graph then displays the Normalized RU Consumption units consumed for the container for the selected period.

You can group metrics by using the **Apply splitting** option.

The normalized request unit consumption metric for each container are displayed as shown in the following image:

Next steps

- Monitor Azure Cosmos DB data by using diagnostic settings in Azure.
- Audit Azure Cosmos DB control plane operations

Monitor Azure Cosmos DB data by using diagnostic settings in Azure

11/2/2020 • 7 minutes to read • Edit Online

APPLIES TO: SQL API Cassandra API Gremlin API Table API Azure Cosmos DB API for MongoDB

Diagnostic settings in Azure are used to collect resource logs. Azure resource Logs are emitted by a resource and provide rich, frequent data about the operation of that resource. These logs are captured per request and they are also referred as "data plane logs". Some examples of the data plane operations include delete, insert, and readFeed. The content of these logs varies by resource type.

Platform metrics and the Activity logs are collected automatically, whereas you must create a diagnostic setting to collect resource logs or forward them outside of Azure Monitor. You can turn on diagnostic setting for Azure Cosmos accounts by using the following steps:

1. Sign into the Azure portal.
2. Navigate to your Azure Cosmos account. Open the **Diagnostic settings** pane, and then select **Add diagnostic setting** option.
3. In the **Diagnostic settings** pane, fill the form with the following details:
   - **Name**: Enter a name for the logs to create.
   - You can store the logs to **Archive to a storage account**, **Stream to an event hub** or **Send to Log Analytics**
4. When you create a diagnostic setting, you specify which category of logs to collect. The categories of logs supported by Azure Cosmos DB are listed below along with sample log collected by them:

- **DataPlaneRequests**: Select this option to log back-end requests to all APIs, which include SQL, Graph, MongoDB, Cassandra, and Table API accounts in Azure Cosmos DB. Key properties to note are: `Requestcharge`, `statusCode`, `clientIPaddress`, `partitionID`, `resourceTokenPermissionId`, and `resourceTokenPermissionMode`.

  ```
  { "time": "2019-04-23T23:12:52.3814846Z", "resourceId":
  "/SUBSCRIPTIONS/<your_subscription_ID>/RESOURCEGROUPS/<your_resource_group>/PROVIDERS/MICROSOFT.DOCUME
  NTDB/DATABASEACCOUNTS/<your_database_account>", "category": "DataPlaneRequests", "operationName":
  "ReadFeed", "properties": {"activityId": "66a0c647-af38-4b8d-a92a-c48a805d6460","requestResourceType":
  "Database","requestResourceId": "","collectionRid": "","statusCode": "200","duration":
  "0","userAgent": "Microsoft.Azure.Documents.Common/2.2.0.0","clientIpAddress":
  "10.0.0.24","requestCharge": "1.000000","requestLength": "0","responseLength": "372",
  "resourceTokenPermissionId": "perm-prescriber-app","resourceTokenPermissionMode": "all",
  "resourceTokenUserRid": "","region": "East US","partitionId": "062abe3e-de63-4aa5-b9de-
  4a77119c59f8","keyType": "PrimaryReadOnlyMasterKey","databaseName": "","collectionName": ""}}
  ```

- **MongoRequests**: Select this option to log user-initiated requests from the front end to serve requests to Azure Cosmos DB's API for MongoDB. This log type is not available for other API accounts. Key properties to note are: `Requestcharge`, `opCode`. When you enable MongoRequests in diagnostics logs, make sure to turn off the DataPlaneRequests. You would see one log for every request made on the API.

```
{ "time": "2019-04-10T15:10:46.7820998Z", "resourceId":
"/SUBSCRIPTIONS/<your_subscription_ID>/RESOURCEGROUPS/<your_resource_group>/PROVIDERS/MICROSOFT.DOCUME
NTDB/DATABASEACCOUNTS/<your_database_account>", "category": "MongoRequests", "operationName": "ping",
"properties": {"activityId": "823cae64-0000-0000-0000-000000000000","opCode":
"MongoOpCode_OP_QUERY","errorCode": "0","duration": "0","requestCharge": "0.000000","databaseName":
"admin","collectionName": "$cmd","retryCount": "0"}}
```

- **CassandraRequests**: Select this option to log user-initiated requests from the front end to serve requests to Azure Cosmos DB's API for Cassandra. This log type is not available for other API accounts. The key properties to note are `operationName`, `requestCharge`, `piiCommandText`. When you enable CassandraRequests in diagnostics logs, make sure to turn off the DataPlaneRequests. You would see one log for every request made on the API.

```
{ "time": "2020-03-30T23:55:10.9579593Z", "resourceId":
"/SUBSCRIPTIONS/<your_subscription_ID>/RESOURCEGROUPS/<your_resource_group>/PROVIDERS/MICROSOFT.DOCUME
NTDB/DATABASEACCOUNTS/<your_database_account>", "category": "CassandraRequests", "operationName":
"QuerySelect", "properties": {"activityId": "6b33771c-baec-408a-b305-3127c17465b6","opCode": "
<empty>","errorCode": "-1","duration": "0.311900","requestCharge": "1.589237","databaseName":
"system","collectionName": "local","retryCount": "<empty>","authorizationTokenType":
"PrimaryMasterKey","address": "104.42.195.92","piiCommandText": "{"request":"SELECT key from
system.local"}","userAgent": """}}
```

- **QueryRuntimeStatistics**: Select this option to log the query text that was executed. This log type is available for SQL API accounts only.

```
{ "time": "2019-04-14T19:08:11.6353239Z", "resourceId":
"/SUBSCRIPTIONS/<your_subscription_ID>/RESOURCEGROUPS/<your_resource_group>/PROVIDERS/MICROSOFT.DOCUME
NTDB/DATABASEACCOUNTS/<your_database_account>", "category": "QueryRuntimeStatistics", "properties":
{"activityId": "278b0661-7452-4df3-b992-8aa0864142cf","databasename": "Tasks","collectionname":
"Items","partitionkeyrangeid": "0","querytext": "{"query":"SELECT *\nFROM c\nWHERE (c.p1__10 !=
true)","parameters":[]}"}}
```

- **PartitionKeyStatistics**: Select this option to log the statistics of the partition keys. This is currently represented with the storage size (KB) of the partition keys. See the Troubleshooting issues by using Azure Diagnostic queries section of this article. For example queries that use "PartitionKeyStatistics". The log is emitted against the first three partition keys that occupy most data storage. This log contains data such as subscription ID, region name, database name, collection name, partition key, and storage size in KB.

```
{ "time": "2019-10-11T02:33:24.2018744Z", "resourceId":
"/SUBSCRIPTIONS/<your_subscription_ID>/RESOURCEGROUPS/<your_resource_group>/PROVIDERS/MICROSOFT.DOCUME
NTDB/DATABASEACCOUNTS/<your_database_account>", "category": "PartitionKeyStatistics", "properties":
{"subscriptionId": "<your_subscription_ID>","regionName": "West US 2","databaseName":
"KustoQueryResults","collectionname": "CapacityMetrics","partitionkey": "
["CapacityMetricsPartition.136"]","sizeKb": "2048270"}}
```

- **PartitionKeyRUConsumption**: This log reports the aggregated per-second RU/s consumption of partition keys. Currently, Azure Cosmos DB reports partition keys for SQL API accounts only and for point read/write and stored procedure operations. other APIs and operation types are not supported. For other APIs, the partition key column in the diagnostic log table will be empty. This log contains data such as subscription ID, region name, database name, collection name, partition key, operation type, and request charge. See the Troubleshooting issues by using Azure Diagnostic queries section of this article. For example queries that use "PartitionKeyRUConsumption".

- **ControlPlaneRequests**: This log contains details on control plane operations like creating an account, adding or removing a region, updating account replication settings etc. This log type is available for all API types that include SQL (Core), MongoDB, Gremlin, Cassandra, Table API.

- **Requests**: Select this option to collect metric data from Azure Cosmos DB to the destinations in the diagnostic setting. This is the same data collected automatically in Azure Metrics. Collect metric data with resource logs to analyze both kinds of data together and to send metric data outside of Azure Monitor.

For detailed information about how to create a diagnostic setting by using the Azure portal, CLI, or PowerShell, see Create diagnostic setting to collect platform logs and metrics in Azure article.

Troubleshoot issues with diagnostics queries

1. How to query for the operations that are taking longer than 3 milliseconds to run:

```
AzureDiagnostics
| where toint(duration_s) > 3 and ResourceProvider=="MICROSOFT.DOCUMENTDB" and
Category=="DataPlaneRequests"
| summarize count() by clientIpAddress_s, TimeGenerated
```

2. How to query for the user agent that is running the operations:

```
AzureDiagnostics
| where ResourceProvider=="MICROSOFT.DOCUMENTDB" and Category=="DataPlaneRequests"
| summarize count() by OperationName, userAgent_s
```

3. How to query for the long running operations:

```
AzureDiagnostics
| where ResourceProvider=="MICROSOFT.DOCUMENTDB" and Category=="DataPlaneRequests"
| project TimeGenerated , duration_s
| summarize count() by bin(TimeGenerated, 5s)
| render timechart
```

4. How to get partition key statistics to evaluate skew across top 3 partitions for a database account:

```
AzureDiagnostics
| where ResourceProvider=="MICROSOFT.DOCUMENTDB" and Category=="PartitionKeyStatistics"
| project SubscriptionId, regionName_s, databaseName_s, collectionName_s, partitionKey_s, sizeKb_d,
ResourceId
```

5. How to get the request charges for expensive queries?

```
AzureDiagnostics
| where ResourceProvider=="MICROSOFT.DOCUMENTDB" and Category=="DataPlaneRequests" and
todouble(requestCharge_s) > 10.0
| project activityId_g, requestCharge_s
| join kind= inner (
AzureDiagnostics
| where ResourceProvider =="MICROSOFT.DOCUMENTDB" and Category == "QueryRuntimeStatistics"
| project activityId_g, querytext_s
) on $left.activityId_g == $right.activityId_g
| order by requestCharge_s desc
| limit 100
```

6. How to find which operations are taking most of RU/s?

```
AzureDiagnostics
| where ResourceProvider=="MICROSOFT.DOCUMENTDB" and Category=="DataPlaneRequests"
| where TimeGenerated >= ago(2h)
| summarize max(responseLength_s), max(requestLength_s), max(requestCharge_s), count = count() by
OperationName, requestResourceType_s, userAgent_s, collectionRid_s, bin(TimeGenerated, 1h)
```

7. How to get all queries that are consuming more then 100 RU/s joined with data from **DataPlaneRequests** and **QueryRunTimeStatistics**.

```
AzureDiagnostics
| where ResourceProvider=="MICROSOFT.DOCUMENTDB" and Category=="DataPlaneRequests" and
todouble(requestCharge_s) > 100.0
| project activityId_g, requestCharge_s
| join kind= inner (
        AzureDiagnostics
        | where ResourceProvider =="MICROSOFT.DOCUMENTDB" and Category == "QueryRuntimeStatistics"
        | project activityId_g, querytext_s
) on $left.activityId_g == $right.activityId_g
| order by requestCharge_s desc
| limit 100
```

8. How to get the request charges and the execution duration of a query?

```
AzureDiagnostics
| where TimeGenerated >= ago(24hr)
| where Category == "QueryRuntimeStatistics"
| join (
AzureDiagnostics
| where TimeGenerated >= ago(24hr)
| where Category == "DataPlaneRequests"
) on $left.activityId_g == $right.activityId_g
| project databasename_s, collectionname_s, OperationName1 , querytext_s,requestCharge_s1,
duration_s1, bin(TimeGenerated, 1min)
```

9. How to get the distribution for different operations?

```
AzureDiagnostics
| where ResourceProvider=="MICROSOFT.DOCUMENTDB" and Category=="DataPlaneRequests"
| where TimeGenerated >= ago(2h)
| summarize count = count()  by OperationName, requestResourceType_s, bin(TimeGenerated, 1h)
```

10. What is the maximum throughput that a partition has consumed?

```
AzureDiagnostics
| where ResourceProvider=="MICROSOFT.DOCUMENTDB" and Category=="DataPlaneRequests"
| where TimeGenerated >= ago(2h)
| summarize max(requestCharge_s) by bin(TimeGenerated, 1h), partitionId_g
```

11. How to get the information about the partition keys RU/s consumption per second?

```
AzureDiagnostics
| where ResourceProvider == "MICROSOFT.DOCUMENTDB" and Category == "PartitionKeyRUConsumption"
| summarize total = sum(todouble(requestCharge_s)) by databaseName_s, collectionName_s,
partitionKey_s, TimeGenerated
| order by TimeGenerated asc
```

12. How to get the request charge for a specific partition key

```
AzureDiagnostics
| where ResourceProvider == "MICROSOFT.DOCUMENTDB" and Category == "PartitionKeyRUConsumption"
| where parse_json(partitionKey_s)[0] == "2"
```

13. How to get the top partition keys with most RU/s consumed in a specific period?

```
AzureDiagnostics
| where ResourceProvider == "MICROSOFT.DOCUMENTDB" and Category == "PartitionKeyRUConsumption"
| where TimeGenerated >= datetime("11/26/2019, 11:20:00.000 PM") and TimeGenerated <=
datetime("11/26/2019, 11:30:00.000 PM")
| summarize total = sum(todouble(requestCharge_s)) by databaseName_s, collectionName_s, partitionKey_s
| order by total desc
```

14. How to get the logs for the partition keys whose storage size is greater than 8 GB?

```
AzureDiagnostics
| where ResourceProvider=="MICROSOFT.DOCUMENTDB" and Category=="PartitionKeyStatistics"
| where todouble(sizeKb_d) > 800000
```

15. How to get P99 or P50 replication latencies for operations, request charge or the length of the response?

```
AzureDiagnostics
| where ResourceProvider=="MICROSOFT.DOCUMENTDB" and Category=="DataPlaneRequests"
| where TimeGenerated >= ago(2d)
| summarize
percentile(todouble(responseLength_s), 50), percentile(todouble(responseLength_s), 99),
max(responseLength_s),
percentile(todouble(requestCharge_s), 50), percentile(todouble(requestCharge_s), 99),
max(requestCharge_s),
percentile(todouble(duration_s), 50), percentile(todouble(duration_s), 99), max(duration_s),
count()
by OperationName, requestResourceType_s, userAgent_s, collectionRid_s, bin(TimeGenerated, 1h)
```

16. How to get ControlPlane logs?

    Remember to switch on the flag as described in the Disable key-based metadata write access article, and execute the operations by using Azure PowerShell, the Azure CLI, or Azure Resource Manager.

```
AzureDiagnostics
| where Category =="ControlPlaneRequests"
| summarize by OperationName
```

Next steps

- Azure Monitor for Azure Cosmos DB
- Monitor and debug with metrics in Azure Cosmos DB

How to audit Azure Cosmos DB control plane operations

12/4/2020 • 7 minutes to read • Edit Online

APPLIES TO: SQL API Cassandra API Gremlin API Table API Azure Cosmos DB API for MongoDB

Control Plane in Azure Cosmos DB is a RESTful service that enables you to perform a diverse set of operations on the Azure Cosmos account. It exposes a public resource model (for example: database, account) and various operations to the end users to perform actions on the resource model. The control plane operations include changes to the Azure Cosmos account or container. For example, operations such as create an Azure Cosmos account, add a region, update throughput, region failover, add a VNet etc. are some of the control plane operations. This article explains how to audit the control plane operations in Azure Cosmos DB. You can run the control plane operations on Azure Cosmos accounts by using Azure CLI, PowerShell or Azure portal, whereas for containers, use Azure CLI or PowerShell.

The following are some example scenarios where auditing control plane operations is helpful:

- You want to get an alert when the firewall rules for your Azure Cosmos account are modified. The alert is required to find unauthorized modifications to rules that govern the network security of your Azure Cosmos account and take quick action.
- You want to get an alert if a new region is added or removed from your Azure Cosmos account. Adding or removing regions has implications on billing and data sovereignty requirements. This alert will help you detect an accidental addition or removal of region on your account.
- You want to get more details from the diagnostic logs on what has changed. For example, a VNet was changed.

Disable key based metadata write access

Before you audit the control plane operations in Azure Cosmos DB, disable the key-based metadata write access on your account. When key based metadata write access is disabled, clients connecting to the Azure Cosmos account through account keys are prevented from accessing the account. You can disable write access by setting the `disableKeyBasedMetadataWriteAccess` property to true. After you set this property, changes to any resource can happen from a user with the proper Azure role and credentials. To learn more on how to set this property, see the Preventing changes from SDKs article.

After the `disableKeyBasedMetadataWriteAccess` is turned on, if the SDK based clients run create or update operations, an error *"Operation 'POST' on resource 'ContainerNameorDatabaseName' is not allowed through Azure Cosmos DB endpoint* is returned. You have to turn on access to such operations for your account, or perform the create/update operations through Azure Resource Manager, Azure CLI or Azure PowerShell. To switch back, set the disableKeyBasedMetadataWriteAccess to **false** by using Azure CLI as described in the Preventing changes from Cosmos SDK article. Make sure to change the value of `disableKeyBasedMetadataWriteAccess` to false instead of true.

Consider the following points when turning off the metadata write access:

- Evaluate and ensure that your applications do not make metadata calls that change the above resources (For example, create collection, update throughput, ...) by using the SDK or account keys.
- Currently, the Azure portal uses account keys for metadata operations and hence these operations will be

blocked. Alternatively, use the Azure CLI, SDKs, or Resource Manager template deployments to perform such operations.

Enable diagnostic logs for control plane operations

You can enable diagnostic logs for control plane operations by using the Azure portal. After enabling, the diagnostic logs will record the operation as a pair of start and complete events with relevant details. For example, the *RegionFailoverStart* and *RegionFailoverComplete* will complete the region failover event.

Use the following steps to enable logging on control plane operations:

1. Sign into Azure portal and navigate to your Azure Cosmos account.
2. Open the **Diagnostic settings** pane, provide a **Name** for the logs to create.
3. Select **ControlPlaneRequests** for log type and select the **Send to Log Analytics** option.

You can also store the logs in a storage account or stream to an event hub. This article shows how to send logs to log analytics and then query them. After you enable, it takes a few minutes for the diagnostic logs to take effect. All the control plane operations performed after that point can be tracked. The following screenshot shows how to enable control plane logs:

View the control plane operations

After you turn on logging, use the following steps to track down operations for a specific account:

1. Sign into Azure portal.
2. Open the **Monitor** tab from the left-hand navigation and then select the **Logs** pane. It opens a UI where you can easily run queries with that specific account in scope. Run the following query to view control plane logs:

```
AzureDiagnostics
| where ResourceProvider=="MICROSOFT.DOCUMENTDB" and Category=="ControlPlaneRequests"
| where TimeGenerated >= ago(1h)
```

The following screenshots capture logs when a consistency level is changed for an Azure Cosmos account:

The following screenshots capture logs when the keyspace or a table of a Cassandra account are created and when the throughput is updated. The control plane logs for create and update operations on the database and the container are logged separately as shown in the following screenshot:

Identify the identity associated to a specific operation

If you want to debug further, you can identify a specific operation in the **Activity log** by using the Activity ID or by the timestamp of the operation. Timestamp is used for some Resource Manager clients where the activity ID is not explicitly passed. The Activity log gives details about the identity with which the operation was initiated. The following screenshot shows how to use the activity ID and find the operations associated with it in the Activity log:

Control plane operations for Azure Cosmos account

The following are the control plane operations available at the account level. Most of the operations are tracked at account level. These operations are available as metrics in Azure monitor:

- Region added
- Region removed
- Account deleted
- Region failed over
- Account created
- Virtual network deleted
- Account network settings updated
- Account replication settings updated
- Account keys updated
- Account backup settings updated
- Account diagnostic settings updated

Control plane operations for database or containers

The following are the control plane operations available at the database and container level. These operations are available as metrics in Azure monitor:

- SQL Database Created
- SQL Database Updated
- SQL Database Throughput Updated
- SQL Database Deleted
- SQL Container Created
- SQL Container Updated
- SQL Container Throughput Updated
- SQL Container Deleted
- Cassandra Keyspace Created
- Cassandra Keyspace Updated
- Cassandra Keyspace Throughput Updated
- Cassandra Keyspace Deleted
- Cassandra Table Created

- Cassandra Table Updated
- Cassandra Table Throughput Updated
- Cassandra Table Deleted
- Gremlin Database Created
- Gremlin Database Updated
- Gremlin Database Throughput Updated
- Gremlin Database Deleted
- Gremlin Graph Created
- Gremlin Graph Updated
- Gremlin Graph Throughput Updated
- Gremlin Graph Deleted
- Mongo Database Created
- Mongo Database Updated
- Mongo Database Throughput Updated
- Mongo Database Deleted
- Mongo Collection Created
- Mongo Collection Updated
- Mongo Collection Throughput Updated
- Mongo Collection Deleted
- AzureTable Table Created
- AzureTable Table Updated
- AzureTable Table Throughput Updated
- AzureTable Table Deleted

Diagnostic log operations

The following are the operation names in diagnostic logs for different operations:

- RegionAddStart, RegionAddComplete
- RegionRemoveStart, RegionRemoveComplete
- AccountDeleteStart, AccountDeleteComplete
- RegionFailoverStart, RegionFailoverComplete
- AccountCreateStart, AccountCreateComplete
- AccountUpdateStart, AccountUpdateComplete
- VirtualNetworkDeleteStart, VirtualNetworkDeleteComplete
- DiagnosticLogUpdateStart, DiagnosticLogUpdateComplete

For API-specific operations, the operation is named with the following format:

- ApiKind + ApiKindResourceType + OperationType
- ApiKind + ApiKindResourceType + "Throughput" + operationType

Example

- CassandraKeyspacesCreate
- CassandraKeyspacesUpdate
- CassandraKeyspacesThroughputUpdate
- SqlContainersUpdate

The *ResourceDetails* property contains the entire resource body as a request payload and it contains all the properties requested to update

Diagnostic log queries for control plane operations

The following are some examples to get diagnostic logs for control plane operations:

```
AzureDiagnostics
| where Category startswith "ControlPlane"
| where OperationName contains "Update"
| project httpstatusCode_s, statusCode_s, OperationName, resourceDetails_s, activityId_g
```

```
AzureDiagnostics
| where Category =="ControlPlaneRequests"
| where TimeGenerated >= todatetime('2020-05-14T17:37:09.563Z')
| project TimeGenerated, OperationName, apiKind_s, apiKindResourceType_s, operationType_s, resourceDetails_s
```

```
AzureDiagnostics
| where Category =="ControlPlaneRequests"
| where  OperationName startswith "SqlContainersUpdate"
```

```
AzureDiagnostics
| where Category =="ControlPlaneRequests"
| where  OperationName startswith "SqlContainersThroughputUpdate"
```

Query to get the activityId and the caller who initiated the container delete operation:

```
(AzureDiagnostics
| where Category == "ControlPlaneRequests"
| where OperationName == "SqlContainersDelete"
| where TimeGenerated >= todatetime('9/3/2020, 5:30:29.300 PM')
| summarize by activityId_g )
| join (
AzureActivity
| parse HTTPRequest with * "clientRequestId\": \"" activityId_g "\"" *
| summarize by Caller, HTTPRequest, activityId_g)
on activityId_g
| project Caller, activityId_g
```

Query to get index or ttl updates. You can then compare the output of this query with an earlier update to see the change in index or ttl.

```
AzureDiagnostics
| where Category =="ControlPlaneRequests"
| where  OperationName == "SqlContainersUpdate"
| project resourceDetails_s
```

output:

```
{id:skewed,indexingPolicy:{automatic:true,indexingMode:consistent,includedPaths:[{path:/*,indexes:
[]}],excludedPaths:[{path:/_etag/?}],compositeIndexes:[],spatialIndexes:[]},partitionKey:{paths:
[/pk],kind:Hash},defaultTtl:1000000,uniqueKeyPolicy:{uniqueKeys:[]},conflictResolutionPolicy:
{mode:LastWriterWins,conflictResolutionPath:/_ts,conflictResolutionProcedure:}
```

Next steps

- Explore Azure Monitor for Azure Cosmos DB
- Monitor and debug with metrics in Azure Cosmos DB

Create alerts for Azure Cosmos DB using Azure Monitor

12/4/2020 • 5 minutes to read • Edit Online

APPLIES TO: SQL API Cassandra API Gremlin API Table API Azure Cosmos DB API for MongoDB

Alerts are used to set up recurring tests to monitor the availability and responsiveness of your Azure Cosmos DB resources. Alerts can send you a notification in the form of an email, or execute an Azure Function when one of your metrics reaches the threshold or if a specific event is logged in the activity log.

You can receive an alert based on the metrics, or the activity log events on your Azure Cosmos account:

- **Metrics** - The alert triggers when the value of a specified metric crosses a threshold you assign. For example, when the total request units consumed exceed 1000 RU/s. This alert is triggered both when the condition is first met and then afterwards when that condition is no longer being met. See the monitoring data reference article for different metrics available in Azure Cosmos DB.
- **Activity log events** – This alert triggers when a certain event occurs. For example, when the keys of your Azure Cosmos account are accessed or refreshed.

You can set up alerts from the Azure Cosmos DB pane or the Azure Monitor service in the Azure portal. Both the interfaces offer the same options. This article shows you how to set up alerts for Azure Cosmos DB using Azure Monitor.

Create an alert rule

This section shows how to create an alert when you receive an HTTP status code 429, which is received when the requests are rate limited. For examples, you may want to receive an alert when there are 100 or more rate limited requests. This article shows you how to configure an alert for such scenario by using the HTTP status code. You can use the similar steps to configure other types of alerts as well, you just need to choose a different condition based on your requirement.

1. Sign into the Azure portal.
2. Select **Monitor** from the left-hand navigation bar and select **Alerts**.
3. Select the New alert rule button to open the Create alert rule pane.
4. Fill out the **Scope** section:
   - Open the **Select resource** pane and configure the following:
   - Choose your **subscription** name.
   - Select **Azure Cosmos DB accounts** for the **resource type**.
   - The **location** of your Azure Cosmos account.
   - After filling in the details, a list of Azure Cosmos accounts in the selected scope is displayed. Choose the one for which you want to configure alerts and select **Done**.
5. Fill out the **Condition** section:
   - Open the **Select condition** pane to open the **Configure signal logic** page and configure the

following:

- Select a signal. The **signal type** can be a **Metric** or an **Activity Log**. Choose **Metrics** for this scenario. Because you want to get an alert when there are rate limiting issues on the total request units metric.
- Select **All** for the **Monitor service**
- Choose a **Signal name**. To get an alert for HTTP status codes, choose the **Total Request Units** signal.
- In the next tab, you can define the logic for triggering an alert and use the chart to view trends of your Azure Cosmos account. The **Total Request Units** metric supports dimensions. These dimensions allow you to filter on the metric. If you don't select any dimension, this value is ignored.
- Choose **StatusCode** as the **Dimension name**. Select **Add custom value** and set the status code to 429.
- In the **Alert logic**, set the **Threshold** to **Static**. The static threshold uses a user-defined threshold value to evaluate the rule, whereas the dynamic thresholds use inbuilt machine learning algorithms to continuously learn the metric behavior pattern and calculate the thresholds automatically.
- Set the **operator** to **Greater than**, the **Aggregation type** to **Total**, and the **Threshold value** to **100**. With this logic, if your client sees more than 100 requests that have a 429 status code, the alert is triggered. You can also configure the aggregation type, aggregation granularity, and the frequency of evaluation based on your requirement.
- After filling the form, select **Done**. The following screenshot shows the details of the alert logic:

6. Fill out the **Action group** section:

- On the **Create rule** pane, select an existing action group, or create a new action group. An action group enables you to define the action to be taken when an alert condition occurs. For this example, create a new action group to receive an email notification when the alert is triggered. Open the **Add action group** pane and fill out the following details:
- **Action group name** - The action group name must be unique within a resource group.
- **Short name** - The action group's Short name, this value is included in email and SMS notifications

to identify which action group was the source of the notification.

- Choose the subscription and the resource group in which this action group will be created.
- Provide a name for your action and select **Email/SMS message/Push/Voice** as the **Action Type**. The following screenshot shows the details of the action type:

7. Fill out the **Alert rule details** section:

   - Define a name for the rule, provide an optional description, the severity level of the alert, choose whether to enable the rule upon rule creation, and then select **Create rule alert** to create the metric rule alert.

After creating the alert, it will be active within 10 minutes.

Common alerting scenarios

The following are some scenarios where you can use alerts:

- When the keys of an Azure Cosmos account are updated.
- When the data or index usage of a container, database, or a region exceeds a certain number of bytes.
- When the normalized RU/s consumption is greater than certain percentage. The normalized RU consumption metric gives the maximum throughput utilization within a replica set. To learn, see the How to monitor normalized RU/s article.
- When a region is added, removed, or if it goes offline.
- When a database or a container is created, deleted, or updated.
- When the throughput of your database or the container is changed.

Next steps

- How to monitor normalized RU/s metric in Azure Cosmos container.
- How to monitor throughput or request unit usage of an operation in Azure Cosmos DB.

Azure Cosmos DB monitoring data reference

11/2/2020 • 6 minutes to read • Edit Online

APPLIES TO: SQL API Cassandra API Gremlin API Table API Azure Cosmos DB API for MongoDB

This article provides a reference of log and metric data collected to analyze the performance and availability of Azure Cosmos DB. See the Monitor Azure Cosmos DB article for how to collect and analyze monitoring data for Azure Cosmos DB.

Resource logs

The following table lists the properties of resource logs in Azure Cosmos DB. The resource logs are collected into Azure Monitor Logs or Azure Storage. In Azure Monitor, logs are collected in the **AzureDiagnostics** table under the resource provider** name of `MICROSOFT.DOCUMENTDB`.

AZURE STORAGE FIELD OR PROPERTY	AZURE MONITOR LOGS PROPERTY	DESCRIPTION
time	**TimeGenerated**	The date and time (UTC) when the operation occurred.
resourceId	**Resource**	The Azure Cosmos DB account for which logs are enabled.
category	**Category**	For Azure Cosmos DB, **DataPlaneRequests**, **MongoRequests**, **QueryRuntimeStatistics**, **PartitionKeyStatistics**, **PartitionKeyRUConsumption**, **ControlPlaneRequests** are the available log types.
operationName	**OperationName**	Name of the operation. The operation name can be `Create`, `Update`, `Read`, `ReadFeed`, `Delete`, `Replace`, `Execute`, `SqlQuery`, `Query`, `JSQuery`, `Head`, `HeadFeed`, or `Upsert`.
properties	n/a	The contents of this field are described in the rows that follow.
activityId	**activityId_g**	The unique GUID for the logged operation.
userAgent	**userAgent_s**	A string that specifies the client user agent from which, the request was sent. The format of user agent is `{user agent name}/{version}`.

AZURE STORAGE FIELD OR PROPERTY	AZURE MONITOR LOGS PROPERTY	DESCRIPTION
requestResourceType	requestResourceType_s	The type of the resource accessed. This value can be database, container, document, attachment, user, permission, stored procedure, trigger, user-defined function, or an offer.
statusCode	statusCode_s	The response status of the operation.
requestResourceId	ResourceId	The resourceId that pertains to the request. Depending on the operation performed, this value may point to `databaseRid`, `collectionRid`, or `documentRid`.
clientIpAddress	clientIpAddress_s	The client's IP address.
requestCharge	requestCharge_s	The number of RU/s that are used by the operation
collectionRid	collectionId_s	The unique ID for the collection.
duration	duration_d	The duration of the operation, in milliseconds.
requestLength	requestLength_s	The length of the request, in bytes.
responseLength	responseLength_s	The length of the response, in bytes.
resourceTokenPermissionId	resourceTokenPermissionId_s	This property indicates the resource token permission Id that you have specified. To learn more about permissions, see the Secure access to your data article.
resourceTokenPermissionMode	resourceTokenPermissionMode_s	This property indicates the permission mode that you have set when creating the resource token. The permission mode can have values such as "all" or "read". To learn more about permissions, see the Secure access to your data article.
resourceTokenUserRid	resourceTokenUserRid_s	This value is non-empty when resource tokens are used for authentication. The value points to the resource ID of the user.
responseLength	responseLength_s	The length of the response, in bytes.

For a list of all Azure Monitor log categories and links to associated schemas, see Azure Monitor Logs categories and schemas.

Metrics

The following tables list the platform metrics collected for Azure CosmOS DB. All metrics are stored in the

namespace **Cosmos DB standard metrics**.

For a list of all Azure Monitor support metrics (including Azure Cosmos DB), see Azure Monitor supported metrics.

Request metrics

METRIC (METRIC DISPLAY NAME)	UNIT (AGGREGATION TYPE)	DESCRIPTION	DIMENSIONS	TIME GRANULARITIES	LEGACY METRIC MAPPING	USAGE
TotalRequests (Total Requests)	Count (Count)	Number of requests made	DatabaseName, CollectionName, Region, StatusCode	All	TotalRequests, Http 2xx, Http 3xx, Http 400, Http 401, Internal Server error, Service Unavailable, Throttled Requests, Average Requests per Second	Used to monitor requests per status code, container at a minute granularity. To get average requests per second, use Count aggregation at minute and divide by 60.
MetadataRequests (Metadata Requests)	Count (Count)	Count of metadata requests. Azure Cosmos DB maintains system metadata container for each account, that allows you to enumerate collections, databases, etc., and their configurations, free of charge.	DatabaseName, CollectionName, Region, StatusCode	All		Used to monitor throttles due to metadata requests.
MongoRequests (Mongo Requests)	Count (Count)	Number of Mongo Requests Made	DatabaseName, CollectionName, Region, CommandName, ErrorCode	All	Mongo Query Request Rate, Mongo Update Request Rate, Mongo Delete Request Rate, Mongo Insert Request Rate, Mongo Count Request Rate	Used to monitor Mongo request errors, usages per command type.

Request Unit metrics

METRIC (METRIC DISPLAY NAME)	UNIT (AGGREGATION TYPE)	DESCRIPTION	DIMENSIONS	TIME GRANULARITIES	LEGACY METRIC MAPPING	USAGE
MongoRequestCharge (Mongo Request Charge)	Count (Total)	Mongo Request Units Consumed	DatabaseName, CollectionName, Region, CommandName, ErrorCode	All	Mongo Query Request Charge, Mongo Update Request Charge, Mongo Delete Request Charge, Mongo Insert Request Charge, Mongo Count Request Charge	Used to monitor Mongo resource RUs in a minute.
TotalRequestUnits (Total Request Units)	Count (Total)	Request Units consumed	DatabaseName, CollectionName, Region, StatusCode	All	TotalRequestUnits	Used to monitor Total RU usage at a minute granularity. To get average RU consumed per second, use Total aggregation at minute and divide by 60.
ProvisionedThroughput (Provisioned Throughput)	Count (Maximum)	Provisioned throughput at container granularity	DatabaseName, ContainerName	5M		Used to monitor provisioned throughput per container.

Storage metrics

METRIC (METRIC DISPLAY NAME)	UNIT (AGGREGATION TYPE)	DESCRIPTION	DIMENSIONS	TIME GRANULARITIES	LEGACY METRIC MAPPING	USAGE
AvailableStorage (Available Storage)	Bytes (Total)	Total available storage reported at 5-minutes granularity per region	DatabaseName, CollectionName, Region	5M	Available Storage	Used to monitor available storage capacity (applicable only for fixed storage collections) Minimum granularity should be 5 minutes.

METRIC (METRIC DISPLAY NAME)	UNIT (AGGREGATION TYPE)	DESCRIPTION	DIMENSIONS	TIME GRANULARITIES	LEGACY METRIC MAPPING	USAGE
DataUsage (Data Usage)	Bytes (Total)	Total data usage reported at 5-minutes granularity per region	DatabaseName, CollectionName, Region	5M	Data size	Used to monitor total data usage at container and region, minimum granularity should be 5 minutes.
IndexUsage (Index Usage)	Bytes (Total)	Total Index usage reported at 5-minutes granularity per region	DatabaseName, CollectionName, Region	5M	Index Size	Used to monitor total data usage at container and region, minimum granularity should be 5 minutes.
DocumentQuota (Document Quota)	Bytes (Total)	Total storage quota reported at 5-minutes granularity per region.	DatabaseName, CollectionName, Region	5M	Storage Capacity	Used to monitor total quota at container and region, minimum granularity should be 5 minutes.
DocumentCount (Document Count)	Count (Total)	Total document count reported at 5-minutes granularity per region	DatabaseName, CollectionName, Region	5M	Document Count	Used to monitor document count at container and region, minimum granularity should be 5 minutes.

Latency metrics

METRIC (METRIC DISPLAY NAME)	UNIT (AGGREGATION TYPE)	DESCRIPTION	DIMENSIONS	TIME GRANULARITIES	USAGE
ReplicationLatency (Replication Latency)	MilliSeconds (Minimum, Maximum, Average)	P99 Replication Latency across source and target regions for geo-enabled account	SourceRegion, TargetRegion	All	Used to monitor P99 replication latency between any two regions for a geo-replicated account.

METRIC (METRIC DISPLAY NAME)	UNIT (AGGREGATION TYPE)	DESCRIPTION	DIMENSIONS	TIME GRANULARITIES	USAGE
Server Side Latency	MilliSeconds (Average)	Time taken by the server to process the request.	CollectionName, ConnectionMode, DatabaseName, OperationType, PublicAPIType, Region	All	Used to monitor the request latency on the Azure Cosmos DB server.

Availability metrics

METRIC (METRIC DISPLAY NAME)	UNIT (AGGREGATION TYPE)	DESCRIPTION	TIME GRANULARITIES	LEGACY METRIC MAPPING	USAGE
ServiceAvailability (Service Availability)	Percent (Minimum, Maximum)	Account requests availability at one hour granularity	1H	Service Availability	Represents the percent of total passed requests. A request is considered to be failed due to system error if the status code is 410, 500 or 503 Used to monitor availability of the account at hour granularity.

Cassandra API metrics

METRIC (METRIC DISPLAY NAME)	UNIT (AGGREGATION TYPE)	DESCRIPTION	DIMENSIONS	TIME GRANULARITIES	USAGE
CassandraRequests (Cassandra Requests)	Count (Count)	Number of Cassandra API requests made	DatabaseName, CollectionName, ErrorCode, Region, OperationType, ResourceType	All	Used to monitor Cassandra requests at a minute granularity. To get average requests per second, use Count aggregation at minute and divide by 60.
CassandraRequestCharges (Cassandra Request Charges)	Count (Sum, Min, Max, Avg)	Request units consumed by the Cassandra API	DatabaseName, CollectionName, Region, OperationType, ResourceType	All	Used to monitor RUs used per minute by a Cassandra API account.
CassandraConnectionClosures (Cassandra Connection Closures)	Count (Count)	Number of Cassandra Connections closed	ClosureReason, Region	All	Used to monitor the connectivity between clients and the Azure Cosmos DB Cassandra API.

See Also

- See Monitoring Azure Cosmos DB for a description of monitoring Azure Cosmos DB.
- See Monitoring Azure resources with Azure Monitor for details on monitoring Azure resources.

Scenario: Exception handling and error logging for logic apps

11/2/2020 • 8 minutes to read • Edit Online

This scenario describes how you can extend a logic app to better support exception handling. We've used a real-life use case to answer the question: "Does Azure Logic Apps support exception and error handling?"

> **NOTE**
> The current Azure Logic Apps schema provides a standard template for action responses. This template includes both internal validation and error responses returned from an API app.

Scenario and use case overview

Here's the story as the use case for this scenario:

A well-known healthcare organization engaged us to develop an Azure solution that would create a patient portal by using Microsoft Dynamics CRM Online. They needed to send appointment records between the Dynamics CRM Online patient portal and Salesforce. We were asked to use the HL7 FHIR standard for all patient records.

The project had two major requirements:

- A method to log records sent from the Dynamics CRM Online portal
- A way to view any errors that occurred within the workflow

> **TIP**
> For a high-level video about this project, see Integration User Group.

How we solved the problem

We chose Azure Cosmos DB As a repository for the log and error records (Cosmos DB refers to records as documents). Because Azure Logic Apps has a standard template for all responses, we would not have to create a custom schema. We could create an API app to **Insert** and **Query** for both error and log records. We could also define a schema for each within the API app.

Another requirement was to purge records after a certain date. Cosmos DB has a property called Time to Live (TTL), which allowed us to set a **Time to Live** value for each record or collection. This capability eliminated the need to manually delete records in Cosmos DB.

> **IMPORTANT**
> To complete this tutorial, you need to create a Cosmos DB database and two collections (Logging and Errors).

Create the logic app

The first step is to create the logic app and open the app in Logic App Designer. In this example, we are using parent-child logic apps. Let's assume that we have already created the parent and are going to create one child logic

app.

Because we are going to log the record coming out of Dynamics CRM Online, let's start at the top. We must use a **Request** trigger because the parent logic app triggers this child.

Logic app trigger

We are using a **Request** trigger as shown in the following example:

```
"triggers": {
        "request": {
          "type": "request",
          "kind": "http",
          "inputs": {
            "schema": {
              "properties": {
                "CRMid": {
                  "type": "string"
                },
                "recordType": {
                  "type": "string"
                },
                "salesforceID": {
                  "type": "string"
                },
                "update": {
                  "type": "boolean"
                }
              },
              "required": [
                "CRMid",
                "recordType",
                "salesforceID",
                "update"
              ],
              "type": "object"
            }
          }
        }
      },
```

Steps

We must log the source (request) of the patient record from the Dynamics CRM Online portal.

1. We must get a new appointment record from Dynamics CRM Online.

   The trigger coming from CRM provides us with the **CRM PatentId**, **record type**, **New or Updated Record** (new or update Boolean value), and **SalesforceId**. The **SalesforceId** can be null because it's only used for an update. We get the CRM record by using the CRM **PatientID** and the **Record Type**.

2. Next, we need to add our Azure Cosmos DB SQL API app **InsertLogEntry** operation as shown here in Logic App Designer.

   Insert log entry

Insert error entry

CreateErrorRecord

Enter a valid integer

STATUSCODE

```
actions('Create_Appointment')['outputs']['statusCode']
```

MESSAGE

```
actions('Create_Appointment')['outputs']['body']['message']
```

SOURCE

```
@{concat(triggerBody()['description'],
' START: ', triggerBody()['start'],
' END: ', triggerBody()['end'],
' COMMENT: ', triggerBody()['comment'] ) }
```

ACTION

Create_Appointment

ERRORS

RESOLVED

0

NOTES

ISERROR

true

PATIENTID

```
@{replace(triggerBody()['participant'][0]['actor']['display'],' ', '')}
```

SEVERITY

4

...

Check for create record failure

Condition

CONDITION

```
@equals(actions('Creat[redacted]ecord')
['status'], 'Failed')
```

If yes — Add an action

CreateErrorRecord

If no, do nothing — Add an action

Logic app source code

> **NOTE**
>
> The following examples are samples only. Because this tutorial is based on an implementation now in production, the value of a **Source Node** might not display properties that are related to scheduling an appointment.>

Logging

The following logic app code sample shows how to handle logging.

Log entry

Here is the logic app source code for inserting a log entry.

```
"InsertLogEntry": {
        "metadata": {
        "apiDefinitionUrl": "https://.../swagger/docs/v1",
        "swaggerSource": "website"
        },
        "type": "Http",
        "inputs": {
        "body": {
            "date": "@{outputs('Gets_NewPatientRecord')['headers']['Date']}",
            "operation": "New Patient",
            "patientId": "@{triggerBody()['CRMid']}",
            "providerId": "@{triggerBody()['providerID']}",
            "source": "@{outputs('Gets_NewPatientRecord')['headers']}"
        },
        "method": "post",
        "uri": "https://.../api/Log"
        },
        "runAfter":    {
            "Gets_NewPatientecord": ["Succeeded"]
        }
}
```

Log request

Here is the log request message posted to the API app.

```
    {
    "uri": "https://.../api/Log",
    "method": "post",
    "body": {
        "date": "Fri, 10 Jun 2016 22:31:56 GMT",
        "operation": "New Patient",
        "patientId": "6b115f6d-a7ee-e511-80f5-3863bb2eb2d0",
        "providerId": "",
        "source": "{/"Pragma/":/"no-cache/",/"x-ms-request-id/":/"e750c9a9-bd48-44c4-bbba-
1688b6f8a132/",/"OData-Version/":/"4.0/",/"Cache-Control/":/"no-cache/",/"Date/":/"Fri, 10 Jun 2016 22:31:56
GMT/",/"Set-
Cookie/":/"ARRAffinity=785f4334b5e64d2db0b84edcc1b84f1bf37319679aefce206b51510e56fd9770;Path=/;Domain=127.0.0.1
/",/"Server/":/"Microsoft-IIS/8.0,Microsoft-HTTPAPI/2.0/",/"X-AspNet-Version/":/"4.0.30319/",/"X-Powered-
By/":/"ASP.NET/",/"Content-Length/":/"1935/",/"Content-Type/":/"application/json; odata.metadata=minimal;
odata.streaming=true/",/"Expires/":/"-1/"}"
        }
    }
```

Log response

Here is the log response message from the API app.

```
{
    "statusCode": 200,
    "headers": {
        "Pragma": "no-cache",
        "Cache-Control": "no-cache",
        "Date": "Fri, 10 Jun 2016 22:32:17 GMT",
        "Server": "Microsoft-IIS/8.0",
        "X-AspNet-Version": "4.0.30319",
        "X-Powered-By": "ASP.NET",
        "Content-Length": "964",
        "Content-Type": "application/json; charset=utf-8",
        "Expires": "-1"
    },
    "body": {
        "ttl": 2592000,
        "id": "6b115f6d-a7ee-e511-80f5-3863bb2eb2d0_1465597937",
        "_rid": "XngRAOT6IQEHAAAAAAAAAA==",
        "_self": "dbs/XngRAA==/colls/XngRAOT6IQE=/docs/XngRAOT6IQEHAAAAAAAAAA==/",
        "_ts": 1465597936,
        "_etag": "/"0400fc2f-0000-0000-0000-575b3ff00000/"",
        "patientID": "6b115f6d-a7ee-e511-80f5-3863bb2eb2d0",
        "timestamp": "2016-06-10T22:31:56Z",
        "source": "{/"Pragma/":/"no-cache/",/"x-ms-request-id/":/"e750c9a9-bd48-44c4-bbba-
1688b6f8a132/",/"OData-Version/":/"4.0/",/"Cache-Control/":/"no-cache/",/"Date/":/"Fri, 10 Jun 2016 22:31:56
GMT/",/"Set-
Cookie/":/"ARRAffinity=785f4334b5e64d2db0b84edcc1b84f1bf37319679aefce206b51510e56fd9770;Path=/;Domain=127.0.0.1
/",/"Server/":/"Microsoft-IIS/8.0,Microsoft-HTTPAPI/2.0/",/"X-AspNet-Version/":/"4.0.30319/",/"X-Powered-
By/":/"ASP.NET/",/"Content-Length/":/"1935/",/"Content-Type/":/"application/json; odata.metadata=minimal;
odata.streaming=true/",/"Expires/":/"-1/"}",
        "operation": "New Patient",
        "salesforceId": "",
        "expired": false
    }
}
```


Now let's look at the error handling steps.

Error handling

The following logic app code sample shows how you can implement error handling.

Create error record

Here is the logic app source code for creating an error record.

```
"actions": {
    "CreateErrorRecord": {
        "metadata": {
        "apiDefinitionUrl": "https://.../swagger/docs/v1",
        "swaggerSource": "website"
        },
        "type": "Http",
        "inputs": {
        "body": {
            "action": "New_Patient",
            "isError": true,
            "crmId": "@{triggerBody()['CRMid']}",
            "patientID": "@{triggerBody()['CRMid']}",
            "message": "@{body('Create_NewPatientRecord')['message']}",
            "providerId": "@{triggerBody()['providerId']}",
            "severity": 4,
            "source": "@{actions('Create_NewPatientRecord')['inputs']['body']}",
            "statusCode": "@{int(outputs('Create_NewPatientRecord')['statusCode'])}",
            "salesforceId": "",
            "update": false
        },
        "method": "post",
        "uri": "https://.../api/CrMtoSfError"
        },
        "runAfter":
        {
            "Create_NewPatientRecord": ["Failed" ]
        }
    }
}
```

Insert error into Cosmos DB--request


```
{
    "uri": "https://.../api/CrMtoSfError",
    "method": "post",
    "body": {
        "action": "New_Patient",
        "isError": true,
        "crmId": "6b115f6d-a7ee-e511-80f5-3863bb2eb2d0",
        "patientId": "6b115f6d-a7ee-e511-80f5-3863bb2eb2d0",
        "message": "Salesforce failed to complete task: Message: duplicate value found: Account_ID_MED__c
duplicates value on record with id: 001U000001c83gK",
        "providerId": "",
        "severity": 4,
        "salesforceId": "",
        "update": false,
        "source": "
{/"Account_Class_vod__c/":/"PRAC/",/"Account_Status_MED__c/":/"I/",/"CRM_HUB_ID__c/":/"6b115f6d-a7ee-e511-80f5-
3863bb2eb2d0/",/"Credentials_vod__c/",/"DTC_ID_MED__c/":/"/",/"Fax/":/"/",/"FirstName/":/"A/",/"Gender_vod__c/"
:/"/",/"IMS_ID__c/":/"/",/"LastName/":/"BAILEY/",/"MasterID_mp__c/":/"/",/"C_ID_MED__c/":/"851588/",/"Middle_vo
d__c/":/"/",/"NPI_vod__c/":/"/",/"PDRP_MED__c/":false,/"PersonDoNotCall/":false,/"PersonEmail/":/"/",/"PersonHa
sOptedOutOfEmail/":false,/"PersonHasOptedOutOfFax/":false,/"PersonMobilePhone/":/"/",/"Phone/":/"/",/"Practicin
g_Specialty__c/":/"FM - FAMILY
MEDICINE/",/"Primary_City__c/":/"/",/"Primary_State__c/":/"/",/"Primary_Street_Line2__c/":/"/",/"Primary_Street
__c/":/"/",/"Primary_Zip__c/":/"/",/"RecordTypeId/":/"012U0000000JaPWIA0/",/"Request_Date__c/":/"2016-06-
10T22:31:55.9647467Z/",/"ONY_ID__c/":/"/",/"Specialty_1_vod__c/":/"/",/"Suffix_vod__c/":/"/",/"Website/":/"/"}"
,
        "statusCode": "400"
    }
}
```


Insert error into Cosmos DB--response

```
{
    "statusCode": 200,
    "headers": {
        "Pragma": "no-cache",
        "Cache-Control": "no-cache",
        "Date": "Fri, 10 Jun 2016 22:31:57 GMT",
        "Server": "Microsoft-IIS/8.0",
        "X-AspNet-Version": "4.0.30319",
        "X-Powered-By": "ASP.NET",
        "Content-Length": "1561",
        "Content-Type": "application/json; charset=utf-8",
        "Expires": "-1"
    },
    "body": {
        "id": "6b115f6d-a7ee-e511-80f5-3863bb2eb2d0-1465597917",
        "_rid": "sQx2APhVzAA8AAAAAAAAAA==",
        "_self": "dbs/sQx2AA==/colls/sQx2APhVzAA=/docs/sQx2APhVzAA8AAAAAAAAAA==/",
        "_ts": 1465597912,
        "_etag": "/"0c00eaac-0000-0000-0000-575b3fdc0000/"",
        "prescriberId": "6b115f6d-a7ee-e511-80f5-3863bb2eb2d0",
        "timestamp": "2016-06-10T22:31:57.3651027Z",
        "action": "New_Patient",
        "salesforceId": "",
        "update": false,
        "body": "CRM failed to complete task: Message: duplicate value found: CRM_HUB_ID__c duplicates value on
record with id: 001U000001c83gK",
        "source": "
{/"Account_Class_vod__c/":/"PRAC/",/"Account_Status_MED__c/":/"I/",/"CRM_HUB_ID__c/":/"6b115f6d-a7ee-e511-80f5-
3863bb2eb2d0/",/"Credentials_vod__c/":/"DO - Degree level is
DO/",/"DTC_ID_MED__c/":/"/",/"Fax/":/"/",/"FirstName/":/"A/",/"Gender_vod__c/":/"/",/"IMS_ID__c/":/"/",/"LastNa
me/":/"BAILEY/",/"MterID_mp__c/":/"/",/"Medicis_ID_MED__c/":/"851588/",/"Middle_vod__c/":/"/",/"NPI_vod__c/":/"
/",/"PDRP_MED__c/":false,/"PersonDoNotCall/":false,/"PersonEmail/":/"/",/"PersonHasOptedOutOfEmail/":false,/"Pe
rsonHasOptedOutOfFax/":false,/"PersonMobilePhone/":/"/",/"Phone/":/"/",/"Practicing_Specialty__c/":/"FM -
FAMILY
MEDICINE/",/"Primary_City__c/":/"/",/"Primary_State__c/":/"/",/"Primary_Street_Line2__c/":/"/",/"Primary_Street
__c/":/"/",/"Primary_Zip__c/":/"/",/"RecordTypeId/":/"012U0000000JaPWIA0/",/"Request_Date__c/":/"2016-06-
10T22:31:55.9647467Z/",/"XXXXXXX/":/"/",/"Specialty_1_vod__c/":/"/",/"Suffix_vod__c/":/"/",/"Website/":/"/"}",
        "code": 400,
        "errors": null,
        "isError": true,
        "severity": 4,
        "notes": null,
        "resolved": 0
        }
}
```


Salesforce error response

```
{
    "statusCode": 400,
    "headers": {
        "Pragma": "no-cache",
        "x-ms-request-id": "3e8e4884-288e-4633-972c-8271b2cc912c",
        "X-Content-Type-Options": "nosniff",
        "Cache-Control": "no-cache",
        "Date": "Fri, 10 Jun 2016 22:31:56 GMT",
        "Set-Cookie":
"ARRAffinity=785f4334b5e64d2db0b84edcc1b84f1bf37319679aefce206b51510e56fd9770;Path=/;Domain=127.0.0.1",
        "Server": "Microsoft-IIS/8.0,Microsoft-HTTPAPI/2.0",
        "X-AspNet-Version": "4.0.30319",
        "X-Powered-By": "ASP.NET",
        "Content-Length": "205",
        "Content-Type": "application/json; charset=utf-8",
        "Expires": "-1"
    },
    "body": {
        "status": 400,
        "message": "Salesforce failed to complete task: Message: duplicate value found: Account_ID_MED__c
duplicates value on record with id: 001U000001c83gK",
        "source": "Salesforce.Common",
        "errors": []
    }
}
```

Return the response back to parent logic app

After you get the response, you can pass the response back to the parent logic app.

Return success response to parent logic app

```
"SuccessResponse": {
    "runAfter":
        {
            "UpdateNew_CRMPatientResponse": ["Succeeded"]
        },
    "inputs": {
        "body": {
            "status": "Success"
    },
    "headers": {
    "    Content-type": "application/json",
        "x-ms-date": "@utcnow()"
    },
    "statusCode": 200
    },
    "type": "Response"
}
```

Return error response to parent logic app

```
"ErrorResponse": {
    "runAfter":
        {
            "Create_NewPatientRecord": ["Failed"]
        },
    "inputs": {
        "body": {
            "status": "BadRequest"
        },
        "headers": {
            "Content-type": "application/json",
            "x-ms-date": "@utcnow()"
        },
        "statusCode": 400
    },
    "type": "Response"
}
```

Cosmos DB repository and portal

Our solution added capabilities with Azure Cosmos DB.

Error management portal

To view the errors, you can create an MVC web app to display the error records from Cosmos DB. The **List**, **Details**, **Edit**, and **Delete** operations are included in the current version.

> **NOTE**
>
> Edit operation: Cosmos DB replaces the entire document. The records shown in the **List** and **Detail** views are samples only. They are not actual patient appointment records.

Here are examples of our MVC app details created with the previously described approach.

Error management list

PrescriberId	Action	TimeStamp	Body	Resolved	
ce1820b0-ee2c-e611-80e7-5065f38a5ba1	Create_SFaddress	6/8/2016 4:16:50 PM	Salesforce failed to complete task: Message: duplicate value found: CRM_Hub_Id__c duplicates value on record with id: a01m0000005H3hu	No	Details
ce1820b0-ee2c-e611-80e7-5065f38a5ba1	Create_SFaddress	6/8/2016 4:16:58 PM	Salesforce failed to complete task: Message: duplicate value found: CRM_Hub_Id__c duplicates value on record with id: a01m0000005H3hu	No	Details
4433c660-ac2d-e611-80e7-c4346bdc0271	Create_SFcallPlan	6/8/2016 7:23:00 PM	Salesforce failed to complete task: Message: duplicate value found: External_ID_MED__c duplicates value on record with id: a2pm0000000TFWg	No	Details
12c3f133-b02d-e611-80e7-c4346bdc0271	Update Decile	6/8/2016 7:36:48 PM	Salesforce failed to complete task: Message: Unable to create/update fields: Account_MED__c. Please check the security settings of this field and verify that it is read/write for your profile or permission set.	No	Details
8ea10937-fe2e-e611-80e9-5065f38a3ba1	Update Prescriber	6/10/2016 11:32:30 AM	Salesforce failed to complete task: Message: Provided external ID field does not exist or is not accessible: 001m000000X8ImoAAF123	No	Details
8ea10937-fe2e-e611-80e9-5065f38a3ba1	Update_ATL	6/10/2016 11:32:32 AM	Salesforce failed to complete task: Message: Account: id value of incorrect type: 001m000000X8ImoAAF123	No	Details

Error management detail view

CRM to SF Error Managment

View

Error Response

TimeStamp	6/8/2016 4:16:50 PM
Code	400
Body	Salesforce failed to complete task: Message: duplicate value found: CRM_Hub_Id__c duplicates value on record with id: a01m0000005H3hu
Source	{"Account_vod__c":"001m000000X74NAAAZ","Address_Type_MED__c":"Mailing","Address_line_2_vod__c":"test str 2","CRM_Hub_Id__c":"ce1820b0-ee2c-e611-80e7-5065f38a5ba1","City_vod__c":"edison","Country_vod__c":"United States","Fax_vod__c":"","License_Expiration_Date_vod__c":"2016-06-30","License_State_MED__c":"NJ","License_Status_vod__c":"Valid_vod","License_vod__c":"342198","Name":"test str 1","Phone_vod__c":"","Primary_vod__c":"false","SLX_Address_Line_3__c":"","State_vod__c":"NJ","Zip_vod__c":"435465"}
Action	Create_SFaddress
Notes	
IsError	☑
PrescriberId	ce1820b0-ee2c-e611-80e7-5065f38a5ba1

Back to List

© 2016 - VNBConsulting, Inc

Log management portal

To view the logs, we also created an MVC web app. Here are examples of our MVC app details created with the previously described approach.

Sample log detail view

PeterJamesChalmers_1466191912
Document

Save Discard Delete Properties


```
{
  "id": "PeterJamesChalmers_1466191912",
  "patientId": "PeterJamesChalmers",
  "timestamp": "2016-06-17T19:31:51.8360138Z",
  "source": "Discussion on the results of your recent MRI  START:  2013-12-10T09:00:00Z END: 2013-12-10T11:00:00Z  COMMENT:  Further expand on the results of the MRI and determine the next actions that may be appropriate.",
  "operation": "General Discussion",
  "Provider": "DrAdamCareful",
  "ttl": 2592000,
  "notDeleted": true
}
```


Properties
PeterJamesChalmers_1466191912

_RID
Uw4fAJrEEwEqAAAAAAAAAA==

_TS
Fri, 17 Jun 2016 19:31:50 GMT

_SELF
dbs/Uw4fAA==/colls/Uw4fAJrEEwE=/doc

_ETAG
"0000dc00-0000-0000-0000-5764502900

_ATTACHMENTS
attachments/

API app details

Logic Apps exception management API

Our open-source Azure Logic Apps exception management API app provides functionality as described here - there are two controllers:

- **ErrorController** inserts an error record (document) in an Azure Cosmos DB collection.
- **LogController** Inserts a log record (document) in an Azure Cosmos DB collection.

> **TIP**
>
> Both controllers use `async Task<dynamic>` operations, allowing operations to resolve at runtime, so we can create the Azure Cosmos DB schema in the body of the operation.

Every document in Azure Cosmos DB must have a unique ID. We are using `PatientId` and adding a timestamp that is converted to a Unix timestamp value (double). We truncate the value to remove the fractional value.

You can view the source code of our error controller API from GitHub.

We call the API from a logic app by using the following syntax:

```
"actions": {
       "CreateErrorRecord": {
         "metadata": {
           "apiDefinitionUrl": "https://.../swagger/docs/v1",
           "swaggerSource": "website"
         },
         "type": "Http",
         "inputs": {
           "body": {
             "action": "New_Patient",
             "isError": true,
             "crmId": "@{triggerBody()['CRMid']}",
             "prescriberId": "@{triggerBody()['CRMid']}",
             "message": "@{body('Create_NewPatientRecord')['message']}",
             "salesforceId": "@{triggerBody()['salesforceID']}",
             "severity": 4,
             "source": "@{actions('Create_NewPatientRecord')['inputs']['body']}",
             "statusCode": "@{int(outputs('Create_NewPatientRecord')['statusCode'])}",
             "update": false
           },
           "method": "post",
           "uri": "https://.../api/CrMtoSfError"
         },
         "runAfter": {
             "Create_NewPatientRecord": ["Failed"]
           }
       }
}
```

The expression in the preceding code sample checks for the *Create_NewPatientRecord* status of **Failed**.

Summary

- You can easily implement logging and error handling in a logic app.
- You can use Azure Cosmos DB as the repository for log and error records (documents).
- You can use MVC to create a portal to display log and error records.

Source code

The source code for the Logic Apps exception management API application is available in this GitHub repository.

Next steps

- View more logic app examples and scenarios
- Monitor logic apps
- Automate logic app deployment

Options to migrate your on-premises or cloud data to Azure Cosmos DB

11/2/2020 • 7 minutes to read • Edit Online

APPLIES TO: SQL API Cassandra API Gremlin API Table API Azure Cosmos DB API for MongoDB

You can load data from various data sources to Azure Cosmos DB. Since Azure Cosmos DB supports multiple APIs, the targets can be any of the existing APIs. The following are some scenarios where you migrate data to Azure Cosmos DB:

- Move data from one Azure Cosmos container to another container in the same database or a different databases.
- Moving data between dedicated containers to shared database containers.
- Move data from an Azure Cosmos account located in region1 to another Azure Cosmos account in the same or a different region.
- Move data from a source such as Azure blob storage, a JSON file, Oracle database, Couchbase, DynamoDB to Azure Cosmos DB.

In order to support migration paths from the various sources to the different Azure Cosmos DB APIs, there are multiple solutions that provide specialized handling for each migration path. This document lists the available solutions and describes their advantages and limitations.

Factors affecting the choice of migration tool

The following factors determine the choice of the migration tool:

- **Online vs offline migration**: Many migration tools provide a path to do a one-time migration only. This means that the applications accessing the database might experience a period of downtime. Some migration solutions provide a way to do a live migration where there is a replication pipeline set up between the source and the target.

- **Data source**: The existing data can be in various data sources like Oracle DB2, Datastax Cassanda, Azure SQL Database, PostgreSQL, etc. The data can also be in an existing Azure Cosmos DB account and the intent of migration can be to change the data model or repartition the data in a container with a different partition key.

- **Azure Cosmos DB API**: For the SQL API in Azure Cosmos DB, there are a variety of tools developed by the Azure Cosmos DB team which aid in the different migration scenarios. All of the other APIs have their own specialized set of tools developed and maintained by the community. Since Azure Cosmos DB supports these APIs at a wire protocol level, these tools should work as-is while migrating data into Azure Cosmos DB too. However, they might require custom handling for throttles as this concept is specific to Azure Cosmos DB.

- **Size of data**: Most migration tools work very well for smaller datasets. When the data set exceeds a few hundred gigabytes, the choices of migration tools are limited.

- **Expected migration duration**: Migrations can be configured to take place at a slow, incremental pace that consumes less throughput or can consume the entire throughput provisioned on the target Azure Cosmos DB container and complete the migration in less time.

Azure Cosmos DB SQL API

MIGRATION TYPE	SOLUTION	SUPPORTED SOURCES	SUPPORTED TARGETS	CONSIDERATIONS
Offline	Data Migration Tool	•JSON/CSV Files •Azure Cosmos DB SQL API •MongoDB •SQL Server •Table Storage •AWS DynamoDB •Azure Blob Storage	•Azure Cosmos DB SQL API •Azure Cosmos DB Tables API •JSON Files	• Easy to set up and supports multiple sources. • Not suitable for large datasets.
Offline	Azure Data Factory	•JSON/CSV Files •Azure Cosmos DB SQL API •Azure Cosmos DB API for MongoDB •MongoDB •SQL Server •Table Storage •Azure Blob Storage See the Azure Data Factory article for other supported sources.	•Azure Cosmos DB SQL API •Azure Cosmos DB API for MongoDB •JSON Files See the Azure Data Factory article for other supported targets.	• Easy to set up and supports multiple sources. • Makes use of the Azure Cosmos DB bulk executor library. • Suitable for large datasets. • Lack of checkpointing - It means that if an issue occurs during the course of migration, you need to restart the whole migration process. • Lack of a dead letter queue - It means that a few erroneous files can stop the entire migration process.
Offline	Azure Cosmos DB Spark connector	Azure Cosmos DB SQL API. You can use other sources with additional connectors from the Spark ecosystem.	Azure Cosmos DB SQL API. You can use other targets with additional connectors from the Spark ecosystem.	• Makes use of the Azure Cosmos DB bulk executor library. • Suitable for large datasets. • Needs a custom Spark setup. • Spark is sensitive to schema inconsistencies and this can be a problem during migration.
Offline	Custom tool with Cosmos DB bulk executor library	The source depends on your custom code	Azure Cosmos DB SQL API	• Provides checkpointing, dead-lettering capabilities which increases migration resiliency. • Suitable for very large datasets (10 TB+). • Requires custom setup of this tool running as an App Service.

MIGRATION TYPE	SOLUTION	SUPPORTED SOURCES	SUPPORTED TARGETS	CONSIDERATIONS
Online	Cosmos DB Functions + ChangeFeed API	Azure Cosmos DB SQL API	Azure Cosmos DB SQL API	• Easy to set up. • Works only if the source is an Azure Cosmos DB container. • Not suitable for large datasets. • Does not capture deletes from the source container.
Online	Custom Migration Service using ChangeFeed	Azure Cosmos DB SQL API	Azure Cosmos DB SQL API	• Provides progress tracking. • Works only if the source is an Azure Cosmos DB container. • Works for larger datasets as well. • Requires the user to set up an App Service to host the Change feed processor. • Does not capture deletes from the source container.
Online	Striim	•Oracle •Apache Cassandra See the Striim website for other supported sources.	•Azure Cosmos DB SQL API • Azure Cosmos DB Cassandra API See the Striim website for other supported targets.	• Works with a large variety of sources like Oracle, DB2, SQL Server. • Easy to build ETL pipelines and provides a dashboard for monitoring. • Supports larger datasets. • Since this is a third-party tool, it needs to be purchased from the marketplace and installed in the user's environment.

Azure Cosmos DB Mongo API

MIGRATION TYPE	SOLUTION	SUPPORTED SOURCES	SUPPORTED TARGETS	CONSIDERATIONS
Online	Azure Database Migration Service	MongoDB	Azure Cosmos DB API for MongoDB	• Makes use of the Azure Cosmos DB bulk executor library. • Suitable for large datasets and takes care of replicating live changes. • Works only with other MongoDB sources.

MIGRATION TYPE	SOLUTION	SUPPORTED SOURCES	SUPPORTED TARGETS	CONSIDERATIONS
Offline	Azure Database Migration Service	MongoDB	Azure Cosmos DB API for MongoDB	• Makes use of the Azure Cosmos DB bulk executor library. • Suitable for large datasets and takes care of replicating live changes. • Works only with other MongoDB sources.
Offline	Azure Data Factory	•JSON/CSV Files •Azure Cosmos DB SQL API •Azure Cosmos DB API for MongoDB •MongoDB •SQL Server •Table Storage •Azure Blob Storage See the Azure Data Factory article for other supported sources.	•Azure Cosmos DB SQL API •Azure Cosmos DB API for MongoDB • JSON files See the Azure Data Factory article for other supported targets.	• Easy to set up and supports multiple sources. • Makes use of the Azure Cosmos DB bulk executor library. • Suitable for large datasets. • Lack of checkpointing means that any issue during the course of migration would require a restart of the whole migration process. • Lack of a dead letter queue would mean that a few erroneous files could stop the entire migration process. • Needs custom code to increase read throughput for certain data sources.
Offline	Existing Mongo Tools (mongodump, mongorestore, Studio3T)	MongoDB	Azure Cosmos DB API for MongoDB	• Easy to set up and integration. • Needs custom handling for throttles.

Azure Cosmos DB Cassandra API

MIGRATION TYPE	SOLUTION	SUPPORTED SOURCES	SUPPORTED TARGETS	CONSIDERATIONS
Offline	cqlsh COPY command	CSV Files	Azure Cosmos DB Cassandra API	• Easy to set up. • Not suitable for large datasets. • Works only when the source is a Cassandra table.

MIGRATION TYPE	SOLUTION	SUPPORTED SOURCES	SUPPORTED TARGETS	CONSIDERATIONS
Offline	Copy table with Spark	•Apache Cassandra •Azure Cosmos DB Cassandra API	Azure Cosmos DB Cassandra API	• Can make use of Spark capabilities to parallelize transformation and ingestion. • Needs configuration with a custom retry policy to handle throttles.
Online	Striim (from Oracle DB/Apache Cassandra)	•Oracle •Apache Cassandra See the Striim website for other supported sources.	•Azure Cosmos DB SQL API •Azure Cosmos DB Cassandra API See the Striim website for other supported targets.	• Works with a large variety of sources like Oracle, DB2, SQL Server. • Easy to build ETL pipelines and provides a dashboard for monitoring. • Supports larger datasets. • Since this is a third-party tool, it needs to be purchased from the marketplace and installed in the user's environment.
Online	Blitzz (from Oracle DB/Apache Cassandra)	•Oracle •Apache Cassandra See the Blitzz website for other supported sources.	Azure Cosmos DB Cassandra API. See the Blitzz website for other supported targets.	• Supports larger datasets. • Since this is a third-party tool, it needs to be purchased from the marketplace and installed in the user's environment.

Other APIs

For APIs other than the SQL API, Mongo API and the Cassandra API, there are various tools supported by each of the API's existing ecosystems.

Table API

- Data Migration Tool
- AzCopy

Gremlin API

- Graph bulk executor library
- Gremlin Spark

Next steps

- Learn more by trying out the sample applications consuming the bulk executor library in .NET and Java.
- The bulk executor library is integrated into the Cosmos DB Spark connector, to learn more, see Azure Cosmos DB Spark connector article.

- Contact the Azure Cosmos DB product team by opening a support ticket under the "General Advisory" problem type and "Large (TB+) migrations" problem subtype for additional help with large scale migrations.

Migrate hundreds of terabytes of data into Azure Cosmos DB

12/4/2020 • 8 minutes to read • Edit Online

APPLIES TO: SQL API Cassandra API Gremlin API Table API Azure Cosmos DB API for MongoDB

Azure Cosmos DB can store terabytes of data. You can perform a large-scale data migration to move your production workload to Azure Cosmos DB. This article describes the challenges involved in moving large-scale data to Azure Cosmos DB and introduces you to the tool that helps with the challenges and migrates data to Azure Cosmos DB. In this case study, the customer used the Cosmos DB SQL API.

Before you migrate the entire workload to Azure Cosmos DB, you can migrate a subset of data to validate some of the aspects like partition key choice, query performance, and data modeling. After you validate the proof of concept, you can move the entire workload to Azure Cosmos DB.

Tools for data migration

Azure Cosmos DB migration strategies currently differ based on the API choice and the size of the data. To migrate smaller datasets – for validating data modeling, query performance, partition key choice etc. – you can choose the Data Migration Tool or Azure Data Factory's Azure Cosmos DB connector. If you are familiar with Spark, you can also choose to use the Azure Cosmos DB Spark connector to migrate data.

Challenges for large-scale migrations

The existing tools for migrating data to Azure Cosmos DB have some limitations that become especially apparent at large scales:

- **Limited scale out capabilities**: In order to migrate terabytes of data into Azure Cosmos DB as quickly as possible, and to effectively consume the entire provisioned throughput, the migration clients should have the ability to scale out indefinitely.
- **Lack of progress tracking and check-pointing**: It is important to track the migration progress and have check-pointing while migrating large data sets. Otherwise, any error that occurs during the migration will stop the migration, and you have to start the process from scratch. It would be not productive to restart the whole migration process when 99% of it has already completed.
- **Lack of dead letter queue**: Within large data sets, in some cases there could be issues with parts of the source data. Additionally, there might be transient issues with the client or the network. Either of these cases should not cause the entire migration to fail. Even though most migration tools have robust retry capabilities that guard against intermittent issues, it is not always enough. For example, if less than 0.01% of the source data documents are greater than 2 MB in size, it will cause the document write to fail in Azure Cosmos DB. Ideally, it is useful for the migration tool to persist these 'failed' documents to another dead letter queue, which can be processed post migration.

Many of these limitations are being fixed for tools like Azure Data factory, Azure Data Migration services.

Custom tool with bulk executor library

The challenges described in the above section, can be solved by using a custom tool that can be easily scaled out across multiple instances and it is resilient to transient failures. Additionally, the custom tool can pause and resume

migration at various checkpoints. Azure Cosmos DB already provides the bulk executor library that incorporates some of these features. For example, the bulk executor library already has the functionality to handle transient errors and can scale out threads in a single node to consume about 500 K RUs per node. The bulk executor library also partitions the source dataset into micro-batches that are operated independently as a form of checkpointing.

The custom tool uses the bulk executor library and supports scaling out across multiple clients and to track errors during the ingestion process. To use this tool, the source data should be partitioned into distinct files in Azure Data Lake Storage (ADLS) so that different migration workers can pick up each file and ingest them into Azure Cosmos DB. The custom tool makes use of a separate collection, which stores metadata about the migration progress for each individual source file in ADLS and tracks any errors associated with them.

The following image describes the migration process using this custom tool. The tool is running on a set of virtual machines, and each virtual machine queries the tracking collection in Azure Cosmos DB to acquire a lease on one of the source data partitions. Once this is done, the source data partition is read by the tool and ingested into Azure Cosmos DB by using the bulk executor library. Next, the tracking collection is updated to record the progress of data ingestion and any errors encountered. After a data partition is processed, the tool attempts to query for the next available source partition. It continues to process the next source partition until all the data is migrated. The source code for the tool is available here.

The tracking collection contains documents as shown in the following example. You will see such documents one for each partition in the source data. Each document contains the metadata for the source data partition such as its location, migration status, and errors (if any):

```
{
  "owner": "25812@bulkimporttest07",
  "jsonStoreEntityImportResponse": {
    "numberOfDocumentsReceived": 446688,
    "isError": false,
    "totalRequestUnitsConsumed": 3950252.2800000003,
    "errorInfo": [],
    "totalTimeTakenInSeconds": 188,
    "numberOfDocumentsImported": 446688
  },
  "storeType": "AZURE_BLOB",
  "name": "sourceDataPartition",
  "location": "sourceDataPartitionLocation",
  "id": "sourceDataPartitionId",
  "isInProgress": false,
  "operation": "unpartitioned-writes",
  "createDate": {
    "seconds": 1561667225,
    "nanos": 146000000
  },
  "completeDate": {
    "seconds": 1561667515,
    "nanos": 180000000
  },
  "isComplete": true
}
```


Prerequisites for data migration

Before the data migration starts, there are a few prerequisites to consider:

Estimate the data size:

The source data size may not exactly map to the data size in Azure Cosmos DB. A few sample documents from the source can be inserted to check their data size in Azure Cosmos DB. Depending on the sample document size, the total data size in Azure Cosmos DB post-migration, can be estimated.

For example, if each document after migration in Azure Cosmos DB is around 1 KB and if there are around 60 billion documents in the source dataset, it would mean that the estimated size in Azure Cosmos DB would be close to 60 TB.

Pre-create containers with enough RUs:

Although Azure Cosmos DB scales out storage automatically, it is not advisable to start from the smallest container size. Smaller containers have lower throughput availability, which means that the migration would take much longer to complete. Instead, it is useful to create the containers with the final data size (as estimated in the previous step) and make sure that the migration workload is fully consuming the provisioned throughput.

In the previous step. since the data size was estimated to be around 60 TB, a container of at least 2.4 M RUs is required to accommodate the entire dataset.

Estimate the migration speed:

Assuming that the migration workload can consume the entire provisioned throughput, the provisioned throughout would provide an estimation of the migration speed. Continuing the previous example, 5 RUs are required for writing a 1-KB document to Azure Cosmos DB SQL API account. 2.4 million RUs would allow a transfer of 480,000 documents per second (or 480 MB/s). This means that the complete migration of 60 TB will take 125,000 seconds or about 34 hours.

In case you want the migration to be completed within a day, you should increase the provisioned throughput to 5 million RUs.

Turn off the indexing:

Since the migration should be completed as soon as possible, it is advisable to minimize time and RUs spent on creating indexes for each of the documents ingested. Azure Cosmos DB automatically indexes all properties, it is worthwhile to minimize indexing to a selected few terms or turn it off completely for the course of migration. You can turn off the container's indexing policy by changing the indexingMode to none as shown below:

```
{
      "indexingMode": "none"
}
```

After the migration is complete, you can update the indexing.

Migration process

After the prerequisites are completed, you can migrate data with the following steps:

1. First import the data from source to Azure Blob Storage. To increase the speed of migration, it is helpful to parallelize across distinct source partitions. Before starting the migration, the source data set should be partitioned into files with size around 200 MB size.

2. The bulk executor library can scale up, to consume 500,000 RUs in a single client VM. Since the available throughput is 5 million RUs, 10 Ubuntu 16.04 VMs (Standard_D32_v3) should be provisioned in the same region where your Azure Cosmos database is located. You should prepare these VMs with the migration tool and its settings file.

3. Run the queue step on one of the client virtual machines. This step creates the tracking collection, which scans the ADLS container and creates a progress-tracking document for each of the source data set's partition files.

4. Next, run the import step on all the client VMs. Each of the clients can take ownership on a source partition and ingest its data into Azure Cosmos DB. Once it's completed and its status is updated in the tracking collection, the clients can then query for the next available source partition in the tracking collection.

5. This process continues until the entire set of source partitions were ingested. Once all the source partitions are processed, the tool should be rerun on the error-correction mode on the same tracking collection. This step is required to identify the source partitions that should to be re-processed due to errors.

6. Some of these errors could be due to incorrect documents in the source data. These should be identified and fixed. Next, you should rerun the import step on the failed partitions to reingest them.

Once the migration is completed, you can validate that the document count in Azure Cosmos DB is same as the document count in the source database. In this example, the total size in Azure Cosmos DB turned out to 65 terabytes. Post migration, indexing can be selectively turned on and the RUs can be lowered to the level required by the workload's operations.

Next steps

- Learn more by trying out the sample applications consuming the bulk executor library in .NET and Java.
- The bulk executor library is integrated into the Cosmos DB Spark connector, to learn more, see Azure Cosmos DB Spark connector article.
- Contact the Azure Cosmos DB product team by opening a support ticket under the "General Advisory" problem type and "Large (TB+) migrations" problem subtype for additional help with large scale migrations.

Azure Cosmos DB query cheat sheets

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: SQL API

The **Azure Cosmos DB query cheat sheets** help you quickly write queries for your data by displaying common database queries, operations, functions, and operators in easy-to-print PDF reference sheets. The cheat sheets include reference information for the SQL, MongoDB, Table, and Gremlin APIs.

Choose from a letter-sized or A3-sized download.

Letter-sized cheat sheets

Download the Azure Cosmos DB letter-sized query cheat sheets if you're going to print to letter-sized paper (8.5" x 11").

Oversized cheat sheets

Download the Azure Cosmos DB A3-sized query cheat sheets if you're going to print using a plotter or large-scale printer on A3-sized paper (11.7" x 16.5").

Next steps

For more help writing queries, see the following articles:

- For SQL API queries, see Query using the SQL API, SQL queries for Azure Cosmos DB, and SQL syntax reference
- For MongoDB queries, see Query using Azure Cosmos DB's API for MongoDB and Azure Cosmos DB's API for MongoDB feature support and syntax
- For Gremlin API queries, see Query using the Gremlin API and Azure Cosmos DB Gremlin graph support
- For Table API queries, see Query using the Table API

Frequently asked questions about different APIs in Azure Cosmos DB

12/4/2020 • 13 minutes to read • Edit Online

APPLIES TO: SQL API Cassandra API Gremlin API Table API Azure Cosmos DB API for MongoDB

What are the typical use cases for Azure Cosmos DB?

Azure Cosmos DB is a good choice for new web, mobile, gaming, and IoT applications where automatic scale, predictable performance, fast order of millisecond response times, and the ability to query over schema-free data is important. Azure Cosmos DB lends itself to rapid development and supporting the continuous iteration of application data models. Applications that manage user-generated content and data are common use cases for Azure Cosmos DB.

How does Azure Cosmos DB offer predictable performance?

A request unit (RU) is the measure of throughput in Azure Cosmos DB. A 1RU throughput corresponds to the throughput of the GET of a 1-KB document. Every operation in Azure Cosmos DB, including reads, writes, SQL queries, and stored procedure executions, has a deterministic RU value that's based on the throughput required to complete the operation. Instead of thinking about CPU, IO, and memory and how they each affect your application throughput, you can think in terms of a single RU measure.

You can configure each Azure Cosmos container with provisioned throughput in terms of RUs of throughput per second. For applications of any scale, you can benchmark individual requests to measure their RU values, and provision a container to handle the total of request units across all requests. You can also scale up or scale down your container's throughput as the needs of your application evolve. For more information about request units and for help with determining your container needs, try the throughput calculator.

How does Azure Cosmos DB support various data models such as key/value, columnar, document, and graph?

Key/value (table), columnar, document, and graph data models are all natively supported because of the ARS (atoms, records, and sequences) design that Azure Cosmos DB is built on. Atoms, records, and sequences can be easily mapped and projected to various data models. The APIs for a subset of models are available right now (SQL, MongoDB, Table, and Gremlin) and others specific to additional data models will be available in the future.

Azure Cosmos DB has a schema agnostic indexing engine capable of automatically indexing all the data it ingests without requiring any schema or secondary indexes from the developer. The engine relies on a set of logical index layouts (inverted, columnar, tree) which decouple the storage layout from the index and query processing subsystems. Cosmos DB also has the ability to support a set of wire protocols and APIs in an extensible manner and translate them efficiently to the core data model (1) and the logical index layouts (2) making it uniquely capable of supporting more than one data model natively.

Can I use multiple APIs to access my data?

Azure Cosmos DB is Microsoft's globally distributed, multi-model database service. Where multi-model means Azure Cosmos DB supports multiple APIs and multiple data models, different APIs use different data formats for storage and wire protocol. For example, SQL uses JSON, MongoDB uses BSON, Table uses EDM, Cassandra uses CQL, Gremlin uses JSON format. As a result, we recommend using the same API for all access to the data in a given account.

Each API operates independently, except the Gremlin and SQL API, which are interoperable.

Is Azure Cosmos DB HIPAA compliant?

Yes, Azure Cosmos DB is HIPAA-compliant. HIPAA establishes requirements for the use, disclosure, and safeguarding of individually identifiable health information. For more information, see the Microsoft Trust Center.

What are the storage limits of Azure Cosmos DB?

There's no limit to the total amount of data that a container can store in Azure Cosmos DB.

What are the throughput limits of Azure Cosmos DB?

There's no limit to the total amount of throughput that a container can support in Azure Cosmos DB. The key idea is to distribute your workload roughly evenly among a sufficiently large number of partition keys.

Are Direct and Gateway connectivity modes encrypted?

Yes both modes are always fully encrypted.

How much does Azure Cosmos DB cost?

For details, refer to the Azure Cosmos DB pricing details page. Azure Cosmos DB usage charges are determined by the number of provisioned containers, the number of hours the containers were online, and the provisioned throughput for each container.

Is a free account available?

Yes, you can sign up for a time-limited account at no charge, with no commitment. To sign up, visit Try Azure Cosmos DB for free or read more in the Try Azure Cosmos DB FAQ.

If you're new to Azure, you can sign up for an Azure free account, which gives you 30 days and a credit to try all the Azure services. If you have a Visual Studio subscription, you're also eligible for free Azure credits to use on any Azure service.

You can also use the Azure Cosmos DB Emulator to develop and test your application locally for free, without creating an Azure subscription. When you're satisfied with how your application is working in the Azure Cosmos DB Emulator, you can switch to using an Azure Cosmos DB account in the cloud.

How can I get additional help with Azure Cosmos DB?

To ask a technical question, you can post to one of these two question and answer forums:

- Microsoft Q&A question page
- Stack Overflow. Stack Overflow is best for programming questions. Make sure your question is on-topic and provide as many details as possible, making the question clear and answerable.

To request new features, create a new request on User voice.

To fix an issue with your account, file a support request in the Azure portal.

Try Azure Cosmos DB subscriptions

You can now enjoy a time-limited Azure Cosmos DB experience without a subscription, free of charge and commitments. To sign up for a Try Azure Cosmos DB subscription, go to Try Azure Cosmos DB for free and use any personal Microsoft account (MSA). This subscription is separate from the Azure Free Trial, and can be used along with an Azure Free Trial or an Azure paid subscription.

Try Azure Cosmos DB subscriptions appear in the Azure portal next other subscriptions associated with your user ID.

The following conditions apply to Try Azure Cosmos DB subscriptions:

- Account access can be granted to personal Microsoft accounts (MSA). Avoid using Azure Active Directory (Azure AD) accounts or accounts belonging to corporate Azure AD Tenants, they might have limitations in place that could block access granting.
- One throughput provisioned container per subscription for SQL, Gremlin API, and Table accounts.

- Up to three throughput provisioned collections per subscription for MongoDB accounts.
- One throughput provisioned database per subscription. Throughput provisioned databases can contain any number of containers inside.
- 10-GB storage capacity.
- Global replication is available in the following Azure regions: Central US, North Europe, and Southeast Asia
- Maximum throughput of 5 K RU/s when provisioned at the container level.
- Maximum throughput of 20 K RU/s when provisioned at the database level.
- Subscriptions expire after 30 days, and can be extended to a maximum of 31 days total. After expiration, the information contained is deleted.
- Azure support tickets can't be created for Try Azure Cosmos DB accounts; however, support is provided for subscribers with existing support plans.

Set up Azure Cosmos DB

How do I sign up for Azure Cosmos DB?

Azure Cosmos DB is available in the Azure portal. First, sign up for an Azure subscription. After you've signed up, you can add an Azure Cosmos DB account to your Azure subscription.

What is a primary key?

A primary key is a security token to access all resources for an account. Individuals with the key have read and write access to all resources in the database account. Use caution when you distribute primary keys. The primary primary key and secondary primary key are available on the **Keys** blade of the Azure portal. For more information about keys, see View, copy, and regenerate access keys.

What are the regions that PreferredLocations can be set to?

The PreferredLocations value can be set to any of the Azure regions in which Cosmos DB is available. For a list of available regions, see Azure regions.

Is there anything I should be aware of when distributing data across the world via the Azure datacenters?

Azure Cosmos DB is present across all Azure regions, as specified on the Azure regions page. Because it's the core service, every new datacenter has an Azure Cosmos DB presence.

When you set a region, remember that Azure Cosmos DB respects sovereign and government clouds. That is, if you create an account in a sovereign region, you can't replicate out of that sovereign region. Similarly, you can't enable replication into other sovereign locations from an outside account.

Is it possible to switch from container level throughput provisioning to database level throughput provisioning? Or vice versa

Container and database level throughput provisioning are separate offerings and switching between either of these require migrating data from source to destination. Which means you need to create a new database or a new container and then migrate data by using bulk executor library or Azure Data Factory.

Does Azure CosmosDB support time series analysis?

Yes Azure CosmosDB supports time series analysis, here is a sample for time series pattern. This sample shows how to use change feed to build aggregated views over time series data. You can extend this approach by using spark streaming or another stream data processor.

What are the Azure Cosmos DB service quotas and throughput limits

See the Azure Cosmos DB service quotas and throughout limits per container and database articles for more information.

Frequently asked questions about SQL API

How do I start developing against the SQL API?

First you must sign up for an Azure subscription. Once you sign up for an Azure subscription, you can add a SQL API container to your Azure subscription. For instructions on adding an Azure Cosmos DB account, see Create an Azure Cosmos database account.

SDKs are available for .NET, Python, Node.js, JavaScript, and Java. Developers can also use the RESTful HTTP APIs to interact with Azure Cosmos DB resources from various platforms and languages.

Can I access some ready-made samples to get a head start?

Samples for the SQL API .NET, Java, Node.js, and Python SDKs are available on GitHub.

Does the SQL API database support schema-free data?

Yes, the SQL API allows applications to store arbitrary JSON documents without schema definitions or hints. Data is immediately available for query through the Azure Cosmos DB SQL query interface.

Does the SQL API support ACID transactions?

Yes, the SQL API supports cross-document transactions expressed as JavaScript-stored procedures and triggers. Transactions are scoped to a single partition within each container and executed with ACID semantics as "all or nothing," isolated from other concurrently executing code and user requests. If exceptions are thrown through the server-side execution of JavaScript application code, the entire transaction is rolled back.

What is a container?

A container is a group of documents and their associated JavaScript application logic. A container is a billable entity, where the cost is determined by the throughput and used storage. Containers can span one or more partitions or servers and can scale to handle practically unlimited volumes of storage or throughput.

- For SQL API, the resource is called a container.
- For Cosmos DB's API for MongoDB accounts, a container maps to a Collection.
- For Cassandra and Table API accounts, a container maps to a Table.
- For Gremlin API accounts, a container maps to a Graph.

Containers are also the billing entities for Azure Cosmos DB. Each container is billed hourly, based on the provisioned throughput and used storage space. For more information, see Azure Cosmos DB Pricing.

How do I create a database?

You can create databases by using the Azure portal, as described in Add a container, one of the Azure Cosmos DB SDKs, or the REST APIs.

How do I set up users and permissions?

You can create users and permissions by using one of the Cosmos DB API SDKs or the REST APIs.

Does the SQL API support SQL?

The SQL query language supported by SQL API accounts is an enhanced subset of the query functionality that's supported by SQL Server. The Azure Cosmos DB SQL query language provides rich hierarchical and relational operators and extensibility via JavaScript-based, user-defined functions (UDFs). JSON grammar allows for modeling JSON documents as trees with labeled nodes, which are used by both the Azure Cosmos DB automatic indexing techniques and the SQL query dialect of Azure Cosmos DB. For information about using SQL grammar, see the SQL Query article.

Does the SQL API support SQL aggregation functions?

The SQL API supports low-latency aggregation at any scale via aggregate functions `COUNT`, `MIN`, `MAX`, `AVG`, and `SUM` via the SQL grammar. For more information, see Aggregate functions.

How does the SQL API provide concurrency?

The SQL API supports optimistic concurrency control (OCC) through HTTP entity tags, or ETags. Every SQL API

resource has an ETag, and the ETag is set on the server every time a document is updated. The ETag header and the current value are included in all response messages. ETags can be used with the If-Match header to allow the server to decide whether a resource should be updated. The If-Match value is the ETag value to be checked against. If the ETag value matches the server ETag value, the resource is updated. If the ETag is no longer current, the server rejects the operation with an "HTTP 412 Precondition failure" response code. The client then refetches the resource to acquire the current ETag value for the resource. In addition, ETags can be used with the If-None-Match header to determine whether a refetch of a resource is needed.

To use optimistic concurrency in .NET, use the AccessCondition class. For a .NET sample, see Program.cs in the DocumentManagement sample on GitHub.

How do I perform transactions in the SQL API?

The SQL API supports language-integrated transactions via JavaScript-stored procedures and triggers. All database operations inside scripts are executed under snapshot isolation. If it's a single-partition container, the execution is scoped to the container. If the container is partitioned, the execution is scoped to documents with the same partition-key value within the container. A snapshot of the document versions (ETags) is taken at the start of the transaction and committed only if the script succeeds. If the JavaScript throws an error, the transaction is rolled back. For more information, see Server-side JavaScript programming for Azure Cosmos DB.

How can I bulk-insert documents into Cosmos DB?

You can bulk-insert documents into Azure Cosmos DB in one of the following ways:

- The bulk executor tool, as described in Using bulk executor .NET library and Using bulk executor Java library
- The data migration tool, as described in Database migration tool for Azure Cosmos DB.
- Stored procedures, as described in Server-side JavaScript programming for Azure Cosmos DB.

Does the SQL API support resource link caching?

Yes, because Azure Cosmos DB is a RESTful service, resource links are immutable and can be cached. SQL API clients can specify an "If-None-Match" header for reads against any resource-like document or container and then update their local copies after the server version has changed.

Is a local instance of SQL API available?

Yes. The Azure Cosmos DB Emulator provides a high-fidelity emulation of the Cosmos DB service. It supports functionality that's identical to Azure Cosmos DB, including support for creating and querying JSON documents, provisioning and scaling collections, and executing stored procedures and triggers. You can develop and test applications by using the Azure Cosmos DB Emulator, and deploy them to Azure at a global scale by making a single configuration change to the connection endpoint for Azure Cosmos DB.

Why are long floating-point values in a document rounded when viewed from data explorer in the portal.

This is limitation of JavaScript. JavaScript uses double-precision floating-point format numbers as specified in IEEE 754 and it can safely hold numbers between $-(2^{53} - 1)$ and $2^{53}-1$ (i.e., 9007199254740991) only.

Where are permissions allowed in the object hierarchy?

Creating permissions by using ResourceTokens is allowed at the container level and its descendants (such as documents, attachments). This implies that trying to create a permission at the database or an account level isn't currently allowed.

Next steps

To learn about frequently asked questions in other APIs, see:

- Frequently asked questions about Azure Cosmos DB's API for MongoDB
- Frequently asked questions about Gremlin API in Azure Cosmos DB
- Frequently asked questions about Cassandra API in Azure Cosmos DB

- Frequently asked questions about Table API in Azure Cosmos DB

Azure Cosmos DB whitepapers

12/4/2020 • 2 minutes to read • Edit Online

APPLIES TO: SQL API Cassandra API Gremlin API Table API Azure Cosmos DB API for MongoDB

Whitepapers allow you to explore Azure Cosmos DB concepts at a deeper level. This article provides you with a list of available whitepapers for Azure Cosmos DB.

WHITEPAPER	DESCRIPTION
Schema-Agnostic Indexing with Azure Cosmos DB	This paper describes Azure Cosmos DB's indexing subsystem. This paper includes Azure Cosmos DB capabilities such as document representation, query language, document indexing approach, core index support, and early production experiences.
Azure Cosmos DB and personal data	This paper provides guidance for Azure Cosmos DB customers managing a cloud-based database, an on-premises database, or both, and who need to ensure that the personal data in their database systems is handled and protected in accordance with current rules.

Azure Cosmos DB NoSQL migration and application development partners

11/2/2020 • 2 minutes to read • Edit Online

APPLIES TO: SQL API Cassandra API Gremlin API Table API Azure Cosmos DB API for MongoDB

From NoSQL migration to application development, you can choose from a variety of experienced systems integrator partners and tools to support your Azure Cosmos DB solutions. This article lists the partners who have solutions or services that use Azure Cosmos DB. This list changes over time, Microsoft is not responsible to any changes or updates made to the solutions of these partners.

Systems Integrator and tooling partners

PARTNER	CAPABILITIES & EXPERIENCE	SUPPORTED COUNTRIES/REGIONS
Striim	Continuous, real-time data movement, Data migration	USA
10thMagnitude	IoT, Retail (inventory), Operational Analytics (Spark), Serverless architecture, App development	USA
Altoros Development LLC	IoT, Personalization Retail (inventory), Serverless architectures NoSQL migration, App development	USA
Avanade	IoT, Retail (inventory), Serverless Architecture, App development	Austria, Germany, Switzerland, Italy, Norway, Spain, UK, Canada
Accenture	IoT, Retail (inventory), Serverless Architecture, App development	Global
Capax Global LLC	IoT, Personalization, Retail (inventory), Operational Analytics (Spark), Serverless architecture, App development	USA
Capgemini	Retail (inventory), IoT, Operational Analytics (Spark), App development	USA, France, UK, Netherlands, Finland
Cognizant	IoT, Personalization, Retail (inventory), Operational Analytics (Spark), App development	USA, Canada, UK, Denmark, Netherlands, Switzerland, Australia, Japan
Infosys	App development	USA
Lagash Systems	IoT, Serverless architecture, App development	Argentina, Chile, Colombia, Mexico
Lambda3 Informatics	Real-time personalization, Retail inventory, App development	Brazil

PARTNER	CAPABILITIES & EXPERIENCE	SUPPORTED COUNTRIES/REGIONS
Neal Analytics	Personalization, Retail (inventory), Operational Analytics (Spark), App development	USA
Pragmatic Works Software Inc	NoSQL migration	USA
Ricoh Digital Services	IoT, Real-time personalization, Retail inventory, NoSQL migration	UK
SNP Technologies	NoSQL migration	USA
Solidsoft Reply	NoSQL migration	Croatia, Sweden, Denmark, Ireland, Bulgaria, Slovenia, Cyprus, Malta, Lithuania, the Czech Republic, Iceland, and Switzerland and Liechtenstein
Spanish Point Technologies	NoSQL migration	Ireland
Syone	NoSQL migration	Portugal
Tallan	App development	USA
TCS	App development	USA, UK, France, Malaysia, Denmark, Norway, Sweden
VTeamLabs	Personalization, Retail (inventory), IoT, Gaming, Operational Analytics (Spark), Serverless architecture, NoSQL Migration, App development	USA
White Duck GmbH	New app development, App Backend, Storage for document-based data	Germany
Xpand IT	New app development	Portugal, UK
Hanu	IoT, App development	USA
Incycle Software	NoSQL migration, Serverless architecture, App development	USA
Orion	Personalization, Retail (inventory), Operational Analytics (Spark), IoT, App development	USA, Canada

Next steps

To learn more about some of Microsoft's other partners, see the Microsoft Partner site.

Retiring the S1, S2, and S3 performance levels

11/2/2020 • 5 minutes to read • Edit Online

APPLIES TO: SQL API Cassandra API Gremlin API Table API Azure Cosmos DB API for MongoDB

> **IMPORTANT**
> The S1, S2, and S3 performance levels discussed in this article are being retired and are no longer available for new Azure Cosmos DB accounts.

This article provides an overview of S1, S2, and S3 performance levels, and discusses how the collections that use these performance levels can be migrated to single partitioned collections. After reading this article, you'll be able to answer the following questions:

- Why are the S1, S2, and S3 performance levels are being retired?
- How do single partition collections and partitioned collections compare to the S1, S2, S3 performance levels?
- What do I need to do to ensure uninterrupted access to my data?
- How will my collection change after the migration?
- How will my billing change after I'm migrated to single partition collections?
- What if I need more than 20 GB of storage?
- Can I change between the S1, S2, and S3 performance levels before the planned migration?
- How do I migrate from the S1, S2, S3 performance levels to single partition collections on my own?
- How am I impacted if I'm an EA customer?

Why are the S1, S2, and S3 performance levels being retired?

The S1, S2, and S3 performance levels do not offer the flexibility that the standard Azure Cosmos DB offer provides. With the S1, S2, S3 performance levels, both the throughput and storage capacity were pre-set and did not offer elasticity. Azure Cosmos DB now offers the ability to customize your throughput and storage, offering you much more flexibility in your ability to scale as your needs change.

How do single partition collections and partitioned collections compare to the S1, S2, S3 performance levels?

The following table compares the throughput and storage options available in single partition collections, partitioned collections, and S1, S2, S3 performance levels. Here is an example for US East 2 region:

QUOTA NAME	PARTITIONED COLLECTION	SINGLE PARTITION COLLECTION	S1	S2	S3
Maximum throughput	Unlimited	10K RU/s	250 RU/s	1 K RU/s	2.5 K RU/s
Minimum throughput	2.5 K RU/s	400 RU/s	250 RU/s	1 K RU/s	2.5 K RU/s

QUOTA NAME	PARTITIONED COLLECTION	SINGLE PARTITION COLLECTION	S1	S2	S3
Maximum storage	Unlimited	20 GB	20 GB	20 GB	20 GB
Price (monthly)	Throughput: $6 / 100 RU/s Storage: $0.25/GB	Throughput: $6 / 100 RU/s Storage: $0.25/GB	$25 USD	$50 USD	$100 USD

Are you an EA customer? If so, see How am I impacted if I'm an EA customer?

What do I need to do to ensure uninterrupted access to my data?

If you have an S1, S2, or S3 collection, you should migrate the collection to a single partition collection programmatically by using the .NET SDK.

How will my collection change after the migration?

If you have an S1 collection, you can migrate them to a single partition collection with 400 RU/s throughput. 400 RU/s is the lowest throughput available with single partition collections. However, the cost for 400 RU/s in a single partition collection is approximately the same as you were paying with your S1 collection and 250 RU/s – so you are not paying for the extra 150 RU/s available to you.

If you have an S2 collection, you can migrate them to a single partition collection with 1 K RU/s. You will see no change to your throughput level.

If you have an S3 collection, you can migrate them to a single partition collection with 2.5 K RU/s. You will see no change to your throughput level.

In each of these cases, after you migrate the collection, you will be able to customize your throughput level, or scale it up and down as needed to provide low-latency access to your users.

How will my billing change after I migrated to the single partition collections?

Assuming you have 10 S1 collections, 1 GB of storage for each, in the US East region, and you migrate these 10 S1 collections to 10 single partition collections at 400 RU/sec (the minimum level). Your bill will look as follows if you keep the 10 single partition collections for a full month:

		Month 1 (31 days = 744 hours)	Month 2 (31 days = 744 hours)
Throughput	Usage Volume	744 hours x 10 S1 collections = 7,440 hours S1	744 hours x 10 x 4 = 29,760 hours of 100 RU/sec
	Price	$0.034 per S1 hour	$0.008 per 100 RU/sec
	Monthly Cost	$0.034 x 7,440 hours = $252.96	$0.008 x 29,760 hours = $238.08
Storage	Usage Volume	included	1 GB x 10 collections = 10 GB
	Price		$0.25 per GB/month
	Monthly Cost		$0.25 x 10 GB = $2.5
Total Monthly Cost		$252.96	$240.58

What if I need more than 20 GB of storage?

Whether you have a collection with S1, S2, or S3 performance level, or have a single partition collection, all of which have 20 GB of storage available, you can use the Azure Cosmos DB Data Migration tool to migrate your data to a partitioned collection with virtually unlimited storage. For information about the benefits of a partitioned collection, see Partitioning and scaling in Azure Cosmos DB.

Can I change between the S1, S2, and S3 performance levels before the planned migration?

Only existing accounts with S1, S2, and S3 performance can be changed and alter performance level tiers programmatically by using the .NET SDK. If you change from S1, S3, or S3 to a single partition collection, you cannot return to the S1, S2, or S3 performance levels.

How do I migrate from the S1, S2, S3 performance levels to single partition collections on my own?

You can migrate from the S1, S2, and S3 performance levels to single partition collections programmatically by using the .NET SDK. You can do this on your own before the planned migration to benefit from the flexible throughput options available with single partition collections.

Migrate to single partition collections by using the .NET SDK

This section only covers changing a collection's performance level using the SQL .NET API, but the process is similar for our other SDKs.

Here is a code snippet for changing the collection throughput to 5,000 request units per second:

```
//Fetch the resource to be updated
Offer offer = client.CreateOfferQuery()
                  .Where(r => r.ResourceLink == collection.SelfLink)
                  .AsEnumerable()
                  .SingleOrDefault();

// Set the throughput to 5000 request units per second
offer = new OfferV2(offer, 5000);

//Now persist these changes to the database by replacing the original resource
await client.ReplaceOfferAsync(offer);
```

Visit MSDN to view additional examples and learn more about our offer methods:

- ReadOfferAsync
- ReadOffersFeedAsync
- ReplaceOfferAsync
- CreateOfferQuery

How am I impacted if I'm an EA customer?

EA customers will be price protected until the end of their current contract.

Next steps

To learn more about pricing and managing data with Azure Cosmos DB, explore these resources:

1. Partitioning data in Cosmos DB. Understand the difference between single partition container and partitioned

containers, as well as tips on implementing a partitioning strategy to scale seamlessly.

2. Cosmos DB pricing. Learn about the cost of provisioning throughput and consuming storage.
3. Request units. Understand the consumption of throughput for different operation types, for example Read, Write, Query.

Common Azure Cosmos DB use cases

11/2/2020 • 8 minutes to read • Edit Online

APPLIES TO: SQL API Cassandra API Gremlin API Table API Azure Cosmos DB API for MongoDB

This article provides an overview of several common use cases for Azure Cosmos DB. The recommendations in this article serve as a starting point as you develop your application with Cosmos DB.

After reading this article, you'll be able to answer the following questions:

- What are the common use cases for Azure Cosmos DB?
- What are the benefits of using Azure Cosmos DB for retail applications?
- What are the benefits of using Azure Cosmos DB as a data store for Internet of Things (IoT) systems?
- What are the benefits of using Azure Cosmos DB for web and mobile applications?

Introduction

Azure Cosmos DB is Microsoft's fast NoSQL database with open APIs for any scale. The service is designed to allow customers to elastically (and independently) scale throughput and storage across any number of geographical regions. Azure Cosmos DB is the first globally distributed database service in the market today to offer comprehensive service level agreements encompassing throughput, latency, availability, and consistency.

Azure Cosmos DB is a global distributed, multi-model database that is used in a wide range of applications and use cases. It is a good choice for any serverless application that needs low order-of-millisecond response times, and needs to scale rapidly and globally. It supports multiple data models (key-value, documents, graphs and columnar) and many APIs for data access including Azure Cosmos DB's API for MongoDB, SQL API, Gremlin API, and Tables API natively, and in an extensible manner.

The following are some attributes of Azure Cosmos DB that make it well-suited for high-performance applications with global ambition.

- Azure Cosmos DB natively partitions your data for high availability and scalability. Azure Cosmos DB offers 99.99% guarantees for availability, throughput, low latency, and consistency on all single-region accounts and all multi-region accounts with relaxed consistency, and 99.999% read availability on all multi-region database accounts.
- Azure Cosmos DB has SSD backed storage with low-latency order-of-millisecond response times.
- Azure Cosmos DB's support for consistency levels like eventual, consistent prefix, session, and bounded-staleness allows for full flexibility and low cost-to-performance ratio. No database service offers as much flexibility as Azure Cosmos DB in levels consistency.
- Azure Cosmos DB has a flexible data-friendly pricing model that meters storage and throughput independently.
- Azure Cosmos DB's reserved throughput model allows you to think in terms of number of reads/writes instead of CPU/memory/IOPs of the underlying hardware.
- Azure Cosmos DB's design lets you scale to massive request volumes in the order of trillions of requests per day.

These attributes are beneficial in web, mobile, gaming, and IoT applications that need low response times and need to handle massive amounts of reads and writes.

IoT and telematics

IoT use cases commonly share some patterns in how they ingest, process, and store data. First, these systems need to ingest bursts of data from device sensors of various locales. Next, these systems process and analyze streaming data to derive real-time insights. The data is then archived to cold storage for batch analytics. Microsoft Azure offers rich services that can be applied for IoT use cases including Azure Cosmos DB, Azure Event Hubs, Azure Stream Analytics, Azure Notification Hub, Azure Machine Learning, Azure HDInsight, and Power BI.

Bursts of data can be ingested by Azure Event Hubs as it offers high throughput data ingestion with low latency. Data ingested that needs to be processed for real-time insight can be funneled to Azure Stream Analytics for real-time analytics. Data can be loaded into Azure Cosmos DB for adhoc querying. Once the data is loaded into Azure Cosmos DB, the data is ready to be queried. In addition, new data and changes to existing data can be read on change feed. Change feed is a persistent, append only log that stores changes to Cosmos containers in sequential order. The all data or just changes to data in Azure Cosmos DB can be used as reference data as part of real-time analytics. In addition, data can further be refined and processed by connecting Azure Cosmos DB data to HDInsight for Pig, Hive, or Map/Reduce jobs. Refined data is then loaded back to Azure Cosmos DB for reporting.

For a sample IoT solution using Azure Cosmos DB, EventHubs and Storm, see the hdinsight-storm-examples repository on GitHub.

For more information on Azure offerings for IoT, see Create the Internet of Your Things.

Retail and marketing

Azure Cosmos DB is used extensively in Microsoft's own e-commerce platforms, that run the Windows Store and XBox Live. It is also used in the retail industry for storing catalog data and for event sourcing in order processing pipelines.

Catalog data usage scenarios involve storing and querying a set of attributes for entities such as people, places, and products. Some examples of catalog data are user accounts, product catalogs, IoT device registries, and bill of materials systems. Attributes for this data may vary and can change over time to fit application requirements.

Consider an example of a product catalog for an automotive parts supplier. Every part may have its own attributes in addition to the common attributes that all parts share. Furthermore, attributes for a specific part can change the following year when a new model is released. Azure Cosmos DB supports flexible schemas and hierarchical data, and thus it is well suited for storing product catalog data.

Azure Cosmos DB is often used for event sourcing to power event driven architectures using its change feed functionality. The change feed provides downstream microservices the ability to reliably and incrementally read inserts and updates (for example, order events) made to an Azure Cosmos DB. This functionality can be leveraged to provide a persistent event store as a message broker for state-changing events and drive order processing workflow between many microservices (which can be implemented as serverless Azure Functions).

In addition, data stored in Azure Cosmos DB can be integrated with HDInsight for big data analytics via Apache Spark jobs. For details on the Spark Connector for Azure Cosmos DB, see Run a Spark job with Cosmos DB and HDInsight.

Gaming

The database tier is a crucial component of gaming applications. Modern games perform graphical processing on mobile/console clients, but rely on the cloud to deliver customized and personalized content like in-game stats, social media integration, and high-score leaderboards. Games often require single-millisecond latencies for reads

and writes to provide an engaging in-game experience. A game database needs to be fast and be able to handle massive spikes in request rates during new game launches and feature updates.

Azure Cosmos DB is used by games like The Walking Dead: No Man's Land by Next Games, and Halo 5: Guardians. Azure Cosmos DB provides the following benefits to game developers:

- Azure Cosmos DB allows performance to be scaled up or down elastically. This allows games to handle updating profile and stats from dozens to millions of simultaneous gamers by making a single API call.
- Azure Cosmos DB supports millisecond reads and writes to help avoid any lags during game play.
- Azure Cosmos DB's automatic indexing allows for filtering against multiple different properties in real-time, for example, locate players by their internal player IDs, or their GameCenter, Facebook, Google IDs, or query based on player membership in a guild. This is possible without building complex indexing or sharding infrastructure.
- Social features including in-game chat messages, player guild memberships, challenges completed, high-score leaderboards, and social graphs are easier to implement with a flexible schema.
- Azure Cosmos DB as a managed platform-as-a-service (PaaS) required minimal setup and management work to allow for rapid iteration, and reduce time to market.

Web and mobile applications

Azure Cosmos DB is commonly used within web and mobile applications, and is well suited for modeling social interactions, integrating with third-party services, and for building rich personalized experiences. The Cosmos DB SDKs can be used build rich iOS and Android applications using the popular Xamarin framework.

Social Applications

A common use case for Azure Cosmos DB is to store and query user generated content (UGC) for web, mobile, and social media applications. Some examples of UGC are chat sessions, tweets, blog posts, ratings, and comments. Often, the UGC in social media applications is a blend of free form text, properties, tags, and relationships that are not bounded by rigid structure. Content such as chats, comments, and posts can be stored in Cosmos DB without requiring transformations or complex object to relational mapping layers. Data properties can be added or modified easily to match requirements as developers iterate over the application code, thus promoting rapid development.

Applications that integrate with third-party social networks must respond to changing schemas from these networks. As data is automatically indexed by default in Cosmos DB, data is ready to be queried at any time. Hence, these applications have the flexibility to retrieve projections as per their respective needs.

Many of the social applications run at global scale and can exhibit unpredictable usage patterns. Flexibility in scaling the data store is essential as the application layer scales to match usage demand. You can scale out by adding additional data partitions under a Cosmos DB account. In addition, you can also create additional Cosmos DB accounts across multiple regions. For Cosmos DB service region availability, see Azure Regions.

Personalization

Nowadays, modern applications come with complex views and experiences. These are typically dynamic, catering to user preferences or moods and branding needs. Hence, applications need to be able to retrieve personalized settings effectively to render UI elements and experiences quickly.

JSON, a format supported by Cosmos DB, is an effective format to represent UI layout data as it is not only lightweight, but also can be easily interpreted by JavaScript. Cosmos DB offers tunable consistency levels that allow fast reads with low latency writes. Hence, storing UI layout data including personalized settings as JSON documents in Cosmos DB is an effective means to get this data across the wire.

Next steps

- To get started with Azure Cosmos DB, follow our quick starts, which walk you through creating an account and getting started with Cosmos DB.

- If you'd like to read more about customers using Azure Cosmos DB, see the customer case studies page.

Going social with Azure Cosmos DB

11/2/2020 • 13 minutes to read • Edit Online

APPLIES TO: SQL API Cassandra API Gremlin API Table API Azure Cosmos DB API for MongoDB

Living in a massively interconnected society means that, at some point in life, you become part of a **social network**. You use social networks to keep in touch with friends, colleagues, family, or sometimes to share your passion with people with common interests.

As engineers or developers, you might have wondered how do these networks store and interconnect your data. Or you might have even been tasked to create or architect a new social network for a specific niche market. That's when the significant question arises: How is all this data stored?

Suppose you're creating a new and shiny social network where your users can post articles with related media, like pictures, videos, or even music. Users can comment on posts and give points for ratings. There will be a feed of posts that users will see and interact with on the main website landing page. This method doesn't sound complex at first, but for the sake of simplicity, let's stop there. (You can delve into custom user feeds affected by relationships, but it goes beyond the goal of this article.)

So, how do you store this data and where?

You might have experience on SQL databases or have a notion of relational modeling of data. You may start drawing something as follows:

A perfectly normalized and pretty data structure... that doesn't scale.

Don't get me wrong, I've worked with SQL databases all my life. They're great, but like every pattern, practice and software platform, it's not perfect for every scenario.

Why isn't SQL the best choice in this scenario? Let's look at the structure of a single post. If I wanted to show the post in a website or application, I'd have to do a query with... by joining eight tables(!) just to show one single post. Now picture a stream of posts that dynamically load and appear on the screen, and you might see where I'm going.

You could use an enormous SQL instance with enough power to solve thousands of queries with many joins to serve your content. But why would you, when a simpler solution exists?

The NoSQL road

This article guides you into modeling your social platform's data with Azure's NoSQL database Azure Cosmos DB cost-effectively. It also tells you how to use other Azure Cosmos DB features like the Gremlin API. Using a NoSQL approach, storing data, in JSON format and applying denormalization, the previously complicated post can be transformed into a single Document:

```
{
    "id":"ew12-res2-234e-544f",
    "title":"post title",
    "date":"2016-01-01",
    "body":"this is an awesome post stored on NoSQL",
    "createdBy":User,
    "images":["https://myfirstimage.png","https://mysecondimage.png"],
    "videos":[
        {"url":"https://myfirstvideo.mp4", "title":"The first video"},
        {"url":"https://mysecondvideo.mp4", "title":"The second video"}
    ],
    "audios":[
        {"url":"https://myfirstaudio.mp3", "title":"The first audio"},
        {"url":"https://mysecondaudio.mp3", "title":"The second audio"}
    ]
}
```

And it can be gotten with a single query, and with no joins. This query is much simple and straightforward, and, budget-wise, it requires fewer resources to achieve a better result.

Azure Cosmos DB makes sure that all properties are indexed with its automatic indexing. The automatic indexing can even be customized. The schema-free approach lets us store documents with different and dynamic structures. Maybe tomorrow you want posts to have a list of categories or hashtags associated with them? Cosmos DB will handle the new Documents with the added attributes without extra work required by us.

Comments on a post can be treated as other posts with a parent property. (This practice simplifies your object mapping.)

```
{
    "id":"1234-asd3-54ts-199a",
    "title":"Awesome post!",
    "date":"2016-01-02",
    "createdBy":User2,
    "parent":"ew12-res2-234e-544f"
}

{
    "id":"asd2-fee4-23gc-jh67",
    "title":"Ditto!",
    "date":"2016-01-03",
    "createdBy":User3,
    "parent":"ew12-res2-234e-544f"
}
```

And all social interactions can be stored on a separate object as counters:

```
{
    "id":"dfe3-thf5-232s-dse4",
    "post":"ew12-res2-234e-544f",
    "comments":2,
    "likes":10,
    "points":200
}
```

Creating feeds is just a matter of creating documents that can hold a list of post IDs with a given relevance order:

```
[
    {"relevance":9, "post":"ew12-res2-234e-544f"},
    {"relevance":8, "post":"fer7-mnb6-fgh9-2344"},
    {"relevance":7, "post":"w34r-qeg6-ref6-8565"}
]
```

You could have a "latest" stream with posts ordered by creation date. Or you could have a "hottest" stream with those posts with more likes in the last 24 hours. You could even implement a custom stream for each user based on logic like followers and interests. It would still be a list of posts. It's a matter of how to build these lists, but the reading performance stays unhindered. Once you acquire one of these lists, you issue a single query to Cosmos DB using the IN keyword to get pages of posts at a time.

The feed streams could be built using Azure App Services' background processes: Webjobs. Once a post is created, background processing can be triggered by using Azure Storage Queues and Webjobs triggered using the Azure Webjobs SDK, implementing the post propagation inside streams based on your own custom logic.

Points and likes over a post can be processed in a deferred manner using this same technique to create an eventually consistent environment.

Followers are trickier. Cosmos DB has a document size limit, and reading/writing large documents can impact the scalability of your application. So you may think about storing followers as a document with this structure:

```
{
    "id":"234d-sd23-rrf2-552d",
    "followersOf": "dse4-qwe2-ert4-aad2",
    "followers":[
        "ewr5-232d-tyrg-iuo2",
        "qejh-2345-sdf1-ytg5",
        //...
        "uie0-4tyg-3456-rwjh"
    ]
}
```

This structure might work for a user with a few thousands followers. If some celebrity joins the ranks, however, this approach will lead to a large document size, and it might eventually hit the document size cap.

To solve this problem, you can use a mixed approach. As part of the User Statistics document you can store the number of followers:

```
{
    "id":"234d-sd23-rrf2-552d",
    "user": "dse4-qwe2-ert4-aad2",
    "followers":55230,
    "totalPosts":452,
    "totalPoints":11342
}
```

You can store the actual graph of followers using Azure Cosmos DB Gremlin API to create vertexes for each user and edges that maintain the "A-follows-B" relationships. With the Gremlin API, you can get the followers of a certain user and create more complex queries to suggest people in common. If you add to the graph the Content Categories that people like or enjoy, you can start weaving experiences that include smart content discovery, suggesting content that those people you follow like, or finding people that you might have much in common with.

The User Statistics document can still be used to create cards in the UI or quick profile previews.

The "Ladder" pattern and data duplication

As you might have noticed in the JSON document that references a post, there are many occurrences of a user. And you'd have guessed right, these duplicates mean that the information that describes a user, given this denormalization, might be present in more than one place.

To allow for faster queries, you incur data duplication. The problem with this side effect is that if by some action, a user's data changes, you need to find all the activities the user ever did and update them all. Doesn't sound practical, right?

You're going to solve it by identifying the key attributes of a user that you show in your application for each activity. If you visually show a post in your application and show just the creator's name and picture, why store all of the user's data in the "createdBy" attribute? If for each comment you just show the user's picture, you don't really need the rest of the user's information. That's where something I call the "Ladder pattern" becomes involved.

Let's take user information as an example:

```
{
    "id":"dse4-qwe2-ert4-aad2",
    "name":"John",
    "surname":"Doe",
    "address":"742 Evergreen Terrace",
    "birthday":"1983-05-07",
    "email":"john@doe.com",
    "twitterHandle":"\@john",
    "username":"johndoe",
    "password":"some_encrypted_phrase",
    "totalPoints":100,
    "totalPosts":24
}
```

By looking at this information, you can quickly detect which is critical information and which isn't, thus creating a "Ladder":

id	name											
		totalPoints	totalPosts	surname	email	twitterHandle	following					
								username	password	phone	address	birthday

The smallest step is called a UserChunk, the minimal piece of information that identifies a user and it's used for data duplication. By reducing the duplicated data size to only the information you'll "show", you reduce the possibility of massive updates.

The middle step is called the user. It's the full data that will be used on most performance-dependent queries on Cosmos DB, the most accessed and critical. It includes the information represented by a UserChunk.

The largest is the Extended User. It includes the critical user information and other data that doesn't need to be read quickly or has eventual usage, like the sign-in process. This data can be stored outside of Cosmos DB, in Azure SQL Database or Azure Storage Tables.

Why would you split the user and even store this information in different places? Because from a performance point of view, the bigger the documents, the costlier the queries. Keep documents slim, with the right information to do all your performance-dependent queries for your social network. Store the other extra information for eventual scenarios like full profile edits, logins, and data mining for usage analytics and Big Data initiatives. You really don't care if the data gathering for data mining is slower, because it's running on Azure SQL Database. You do have concern though that your users have a fast and slim experience. A user stored on Cosmos DB would look like this code:

```
{
    "id":"dse4-qwe2-ert4-aad2",
    "name":"John",
    "surname":"Doe",
    "username":"johndoe"
    "email":"john@doe.com",
    "twitterHandle":"\@john"
}
```

And a Post would look like:

```
{
    "id":"1234-asd3-54ts-199a",
    "title":"Awesome post!",
    "date":"2016-01-02",
    "createdBy":{
        "id":"dse4-qwe2-ert4-aad2",
        "username":"johndoe"
    }
}
```

When an edit arises where a chunk attribute is affected, you can easily find the affected documents. Just use queries that point to the indexed attributes, such as `SELECT * FROM posts p WHERE p.createdBy.id == "edited_user_id"`, and then update the chunks.

The search box

Users will generate, luckily, much content. And you should be able to provide the ability to search and find content that might not be directly in their content streams, maybe because you don't follow the creators, or maybe you're just trying to find that old post you did six months ago.

Because you're using Azure Cosmos DB, you can easily implement a search engine using Azure Cognitive Search in a few minutes without typing any code, other than the search process and UI.

Why is this process so easy?

Azure Cognitive Search implements what they call Indexers, background processes that hook in your data repositories and automagically add, update or remove your objects in the indexes. They support an Azure SQL Database indexers, Azure Blobs indexers and thankfully, Azure Cosmos DB indexers. The transition of information from Cosmos DB to Azure Cognitive Search is straightforward. Both technologies store information in JSON format, so you just need to create your Index and map the attributes from your Documents you want indexed. That's it! Depending on the size of your data, all your content will be available to be searched upon within minutes by the best Search-as-a-Service solution in cloud infrastructure.

For more information about Azure Cognitive Search, you can visit the Hitchhiker's Guide to Search.

The underlying knowledge

After storing all this content that grows and grows every day, you might find thinking: What can I do with all this stream of information from my users?

The answer is straightforward: Put it to work and learn from it.

But what can you learn? A few easy examples include sentiment analysis, content recommendations based on a user's preferences, or even an automated content moderator that makes sure the content published by your social network is safe for the family.

Now that I got you hooked, you'll probably think you need some PhD in math science to extract these patterns and

information out of simple databases and files, but you'd be wrong.

Azure Machine Learning, part of the Cortana Intelligence Suite, is a fully managed cloud service that lets you create workflows using algorithms in a simple drag-and-drop interface, code your own algorithms in R, or use some of the already-built and ready to use APIs such as: Text Analytics, Content Moderator, or Recommendations.

To achieve any of these Machine Learning scenarios, you can use Azure Data Lake to ingest the information from different sources. You can also use U-SQL to process the information and generate an output that can be processed by Azure Machine Learning.

Another available option is to use Azure Cognitive Services to analyze your users content; not only can you understand them better (through analyzing what they write with Text Analytics API), but you could also detect unwanted or mature content and act accordingly with Computer Vision API. Cognitive Services includes many out-of-the-box solutions that don't require any kind of Machine Learning knowledge to use.

A planet-scale social experience

There is a last, but not least, important article I must address: **scalability**. When you design an architecture, each component should scale on its own. You will eventually need to process more data, or you will want to have a bigger geographical coverage. Thankfully, achieving both tasks is a **turnkey experience** with Cosmos DB.

Cosmos DB supports dynamic partitioning out-of-the-box. It automatically creates partitions based on a given **partition key**, which is defined as an attribute in your documents. Defining the correct partition key must be done at design time. For more information, see Partitioning in Azure Cosmos DB.

For a social experience, you must align your partitioning strategy with the way you query and write. (For example, reads within the same partition are desirable, and avoid "hot spots" by spreading writes on multiple partitions.) Some options are: partitions based on a temporal key (day/month/week), by content category, by geographical region, or by user. It all really depends on how you'll query the data and show the data in your social experience.

Cosmos DB will run your queries (including aggregates) across all your partitions transparently, so you don't need to add any logic as your data grows.

With time, you'll eventually grow in traffic and your resource consumption (measured in RUs, or Request Units) will increase. You will read and write more frequently as your user base grows. The user base will start creating and reading more content. So the ability of **scaling your throughput** is vital. Increasing your RUs is easy. You can do it with a few clicks on the Azure portal or by issuing commands through the API.

What happens if things keep getting better? Suppose users from another region, country, or continent notice your platform and start using it. What a great surprise!

But wait! You soon realize their experience with your platform isn't optimal. They're so far away from your operational region that the latency is terrible. You obviously don't want them to quit. If only there was an easy way of **extending your global reach**? There is!

Cosmos DB lets you replicate your data globally and transparently with a couple of clicks and automatically select among the available regions from your client code. This process also means that you can have multiple failover regions.

When you replicate your data globally, you need to make sure that your clients can take advantage of it. If you're using a web frontend or accessing APIs from mobile clients, you can deploy Azure Traffic Manager and clone your Azure App Service on all the desired regions, using a performance configuration to support your extended global coverage. When your clients access your frontend or APIs, they'll be routed to the closest App Service, which in turn, will connect to the local Cosmos DB replica.

Conclusion

This article sheds some light into the alternatives of creating social networks completely on Azure with low-cost services. it delivers results by encouraging the use of a multi-layered storage solution and data distribution called "Ladder".

The truth is that there's no silver bullet for this kind of scenarios. It's the synergy created by the combination of great services that allow us to build great experiences: the speed and freedom of Azure Cosmos DB to provide a great social application, the intelligence behind a first-class search solution like Azure Cognitive Search, the flexibility of Azure App Services to host not even language-agnostic applications but powerful background processes and the expandable Azure Storage and Azure SQL Database for storing massive amounts of data and the analytic power of Azure Machine Learning to create knowledge and intelligence that can provide feedback to your processes and help us deliver the right content to the right users.

Next steps

To learn more about use cases for Cosmos DB, see Common Cosmos DB use cases.