

# Contents

[Azure Cosmos DB Documentation](#)

[Overview](#)

[Welcome to Azure Cosmos DB](#)

[Quickstarts](#)

[Create Cosmos DB resources](#)

[Azure portal](#)

[Azure Resource Manager template](#)

[.NET app - V4 SDK](#)

[.NET app - V3 SDK](#)

[Java app](#)

[Node.js app](#)

[Python app](#)

[Xamarin app](#)

[Tutorials](#)

[1 - Create & manage data](#)

[Build a console app](#)

[.NET](#)

[Java](#)

[Async Java](#)

[Node.js](#)

[Build a web app](#)

[.NET](#)

[Java](#)

[Node.js](#)

[Xamarin](#)

[2 - Migrate data](#)

[Using 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 samples](#)

[Async Java samples](#)

[Node.js samples](#)

[Python samples](#)

[PowerShell](#)

[Azure CLI](#)

[Azure Resource Manager](#)

## Concepts

[NoSQL Vs relational databases](#)

[Global distribution](#)

[Overview](#)

[Consistency levels](#)

[Choose the right consistency level](#)

[Consistency levels and Azure Cosmos DB APIs](#)

[Consistency, availability, and performance tradeoffs](#)

[High availability](#)

[Globally scale provisioned throughput](#)

[Conflict types and resolution policies](#)

[Global distribution - under the hood](#)

[Regional presence](#)

[Transactional and analytical storage](#)

## Partitioning

[Overview](#)

[Partitioning and horizontal scaling](#)

[Create a synthetic partition key](#)

[Provisioned throughput](#)

[Request units](#)

[Provision throughput on containers and databases](#)

[Provision throughput - autopilot mode](#)

[Autopilot FAQ](#)

[SQL query reference](#)

[Getting started](#)

[Clauses](#)

[SELECT](#)

[FROM](#)

[WHERE](#)

[ORDER BY](#)

[GROUP BY](#)

[OFFSET LIMIT](#)

[Subquery](#)

[Joins](#)

[Aliasing](#)

[Arrays and objects](#)

[Keywords](#)

[Constants](#)

[Operators](#)

[Scalar expressions](#)

[Functions](#)

[User-defined functions](#)

[Aggregates](#)

[System functions](#)

[System functions](#)

[Array functions](#)

[Array functions](#)

[ARRAY\\_CONCAT](#)

[ARRAY\\_CONTAINS](#)

[ARRAY\\_LENGTH](#)

[ARRAY\\_SLICE](#)

[Date and time functions](#)

Date and time functions

GetCurrentDateTime

GetCurrentTimestamp

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](#)

[REPLACE](#)

[REPLICATE](#)

[REVERSE](#)

[RIGHT](#)

[RTRIM](#)

[STARTSWITH](#)

[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](#)
- [Parameterized query](#)
- [LINQ to SQL](#)
- [SQL query execution](#)

## Work with Azure Cosmos account

- [Containers and items](#)
  - [Work with databases, containers, and items](#)
  - [Model document data](#)
  - [Indexing](#)
    - [Overview](#)
    - [Indexing policies](#)

## Data types

- [DateTime](#)
- [Time to live](#)
- [Unique key constraints](#)
- [Transactions and optimistic concurrency control](#)

## Change feed

- [Overview](#)
- [Reading change feed](#)
- [Change feed processor](#)
- [Trigger Azure Functions](#)

## Globally distributed analytics and AI

[Analytics use cases](#)

[Analytics - solution architectures](#)

[Lambda architecture](#)

[Built-in Jupyter notebooks](#)

[Jupyter notebooks overview](#)

[Server-side programming](#)

[Stored procedures, triggers, and UDFs](#)

[JavaScript query API](#)

[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 query cost](#)

[Optimize storage cost](#)

[Optimize reads and writes cost](#)

[Optimize multi-region cost](#)

[Optimize development/testing cost](#)

[Optimize cost with reserved capacity](#)

[Security](#)

[Overview](#)

[Data encryption](#)

[Secure access to data](#)

[IP firewall](#)

[Access from virtual networks](#)

[Role-based access control](#)

[Advanced threat protection](#)

[Built-in security controls](#)

[Enterprise readiness](#)

[Online backup and on-demand restore](#)

[Compliance](#)

[Service quotas](#)

[Cassandra, MongoDB, and other APIs](#)

[Cassandra API](#)

[Introduction](#)

[Wire protocol support](#)

[Elastic scale](#)

[Quickstarts](#)

[.NET](#)

[Java](#)

[Node.js](#)

[Python](#)

[Tutorials](#)

[1 - Create & manage data](#)

[2 - Load data](#)

[3 - Query data](#)

[4 - Migrate data](#)

[How-to guides](#)

[Change feed](#)

[Store and manage Spring Data](#)

[Cassandra & Spark](#)

[Introduction](#)

[Connect using Databricks](#)

[Connect using HDInsight](#)

[Create keyspace & 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](#)

[MongoDB API](#)

[Introduction](#)

[Wire protocol support \(3.2\)](#)

[Wire protocol support \(3.6\)](#)

[Quickstarts](#)

[.NET](#)

[Java](#)

[Node.js](#)

[Python](#)

[Xamarin](#)

[Golang](#)

[Tutorials](#)

[1 - Create & 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 - Migrate data](#)

[3 - Query data](#)

[4 - Distribute data globally](#)

[How-to guides](#)

[Pre-migration guide](#)

[Post-migration guide](#)

[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
- Store and manage Spring Data
- MongoDB extension commands
- Manage using Resource Manager templates
- Manage using PowerShell
- Manage using Azure CLI
- Troubleshoot common issues

## Gremlin API

- Introduction
- Wire protocol support
- Quickstarts
  - Gremlin console
  - .NET
  - Java
  - Node.js
  - Python
  - PHP

## Tutorials

- 1 - Query data

## Reference

- Response Headers
- Limits
- Compatibility

## How-to guides

- Graph data modeling
- Graph data partitioning
- Import graph data

[Optimize Gremlin queries](#)

[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](#)

## Table API

[Introduction](#)

[Quickstarts](#)

[.NET](#)

[Java](#)

[Node.js](#)

[Python](#)

[Tutorials](#)

[1 - Create & manage data](#)

[2 - Migrate data](#)

[3 - Query data](#)

[4 - Distribute data globally](#)

[Samples](#)

[.NET](#)

[Java](#)

[Node.js](#)

[Python](#)

[PHP](#)

[C++](#)

[Ruby](#)

[F#](#)

[How-to guides](#)

[Build apps with Table API](#)

- [Table storage design guide](#)
- [Manage using Resource Manager templates](#)
- [Manage using PowerShell](#)
- [Manage using Azure CLI](#)
- [Reference](#)
  - [.NET](#)
  - [.NET Standard](#)
  - [Java](#)
  - [Node.js](#)
  - [Python](#)
  - [Table storage REST APIs](#)
- [Etcd API](#)
  - [Introduction](#)
  - [Azure Kubernetes with Azure Cosmos DB](#)
- [How-to guides](#)
  - [Work with Azure Cosmos account](#)
  - [Work with Azure Cosmos DB resources](#)
    - [Using PowerShell](#)
    - [Using Azure CLI](#)
    - [Using Resource Manager templates](#)
  - [Global distribution](#)
    - [Configure multi-master](#)
    - [Manage consistency levels](#)
    - [Configure conflict resolution policies](#)
  - [Provisioned throughput](#)
    - [Provision throughput on a container](#)
    - [Provision throughput on a database](#)
    - [Estimate RU/s with Cosmos capacity planner](#)
    - [Find request unit charge](#)
    - [Scale using Azure Functions timer](#)
  - [Work with containers and items](#)
  - [Work with Cosmos DB data](#)

- [Work with data using Azure Storage Explorer](#)
- [Work with data using Azure Cosmos explorer](#)
- [Create a container](#)
- [Create a container with large partition key](#)
- [Query a container](#)
- [Data modeling and partitioning - a real-world example](#)
- [Migrate to partitioned containers](#)
- [Configure time to live](#)
- [Manage indexing policies](#)
- [Define unique keys](#)
- [Best practices](#)
- [Performance tips](#)
  - [Get query execution metrics](#)
  - [Tune query performance](#)
  - [Performance tips for .NET SDK](#)
  - [Performance tips for Java SDK](#)
  - [Performance tips for Async Java SDK](#)
  - [Performance test - sample app](#)
  - [Cost-effective reads and writes](#)
- [Troubleshoot](#)
  - [Troubleshoot query performance](#)
  - [Troubleshoot Java Async SDK](#)
  - [Troubleshoot .NET SDK](#)
- [Change Feed](#)
  - [Azure Cosmos DB Trigger logs](#)
  - [Azure Cosmos DB Trigger connection policy](#)
  - [Multiple Azure Cosmos DB Triggers](#)
  - [Troubleshoot Azure Cosmos DB Triggers](#)
  - [Change feed processor start time](#)
  - [Using the change feed estimator](#)
  - [Migrate from change feed processor library](#)
  - [Built-in analytics with Apache Spark](#)

[Azure Databricks Spark connector](#)

[Streaming with Apache Kafka and Cosmos DB](#)

[Built-in Jupyter notebooks](#)

[Enable notebooks](#)

[Use built-in notebook commands](#)

[Server-side programming](#)

[Write stored procedures, triggers, & UDFs](#)

[Write stored procedures & triggers with JavaScript query API](#)

[Use stored procedures, triggers, & UDFs](#)

[Security](#)

[Configure IP firewall](#)

[Configure access from virtual networks](#)

[Configure access from private endpoints](#)

[Connect privately to a Cosmos DB account](#)

[Configure Cross Origin Resource Sharing\(CORS\)](#)

[Secure Azure Cosmos keys using Key Vault](#)

[Certificate-based authentication with Azure AD](#)

[Restrict user access to data operations only](#)

[Configure customer-managed keys](#)

[Enterprise readiness](#)

[Restore data from a backup](#)

[Monitor](#)

[Monitor Cosmos DB](#)

[Monitor with diagnostic logs](#)

[Azure Monitor for Cosmos \(preview\)](#)

[View metrics from Cosmos DB account](#)

[Application logging with Logic Apps](#)

[Monitoring data reference](#)

[Develop locally](#)

[Use the emulator](#)

[Export certificates](#)

[Visual Studio Code extension](#)

[Set up Azure DevOps CI/CD pipeline](#)

[Visualize Cosmos DB data](#)

- [Use Power BI](#)
- [Create a live weather dashboard](#)
- [Use Qlik Sense](#)

[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 Event Hubs and Azure Storage](#)
- [Use Spring Boot Starter](#)
- [Spring Data developer guide](#)
- [ODBC driver](#)

[Migrate data to Cosmos DB](#)

- [Options to migrate data into Cosmos DB](#)
- [Migrate to SQL API using Striim](#)
- [Migrate to Cassandra API using Striim](#)
- [Migrate - Oracle DB to Cassandra API using Blitzz](#)
- [Migrate - Apache Cassandra to Cassandra API using Blitzz](#)
- [Migrate hundreds of terabytes of data into Azure Cosmos DB](#)
- [Migrate one-to-few relational data](#)
- [Migrate - Couchbase to SQL API](#)

[Bulk executor library](#)

- [About bulk executor library](#)
- [Bulk executor - .NET library](#)
- [Bulk executor - Java library](#)

[Reference](#)

[.NET](#)

[.NET Core](#)  
[.NET Standard](#)  
[.NET change feed library](#)  
[.NET bulk executor library](#)

[Java](#)  
[Java bulk executor library](#)

[Async Java](#)

[Node.js](#)

[Python](#)

[REST](#)

[REST Resource Provider](#)

[Resource Manager template](#)

[Emulator release notes](#)

## Resources

[Build your skills with Microsoft Learn](#)

[Query cheat sheet](#)

[SQL playground](#)

[FAQ](#)

[Whitepapers](#)

[Partners](#)

[Videos](#)

[Azure Roadmap](#)

[Try Azure Cosmos DB for free](#)

[Pricing](#)

[Multi-master benefits](#)

[Use cases](#)

[Common use cases](#)

[Social media apps](#)

# Welcome to Azure Cosmos DB

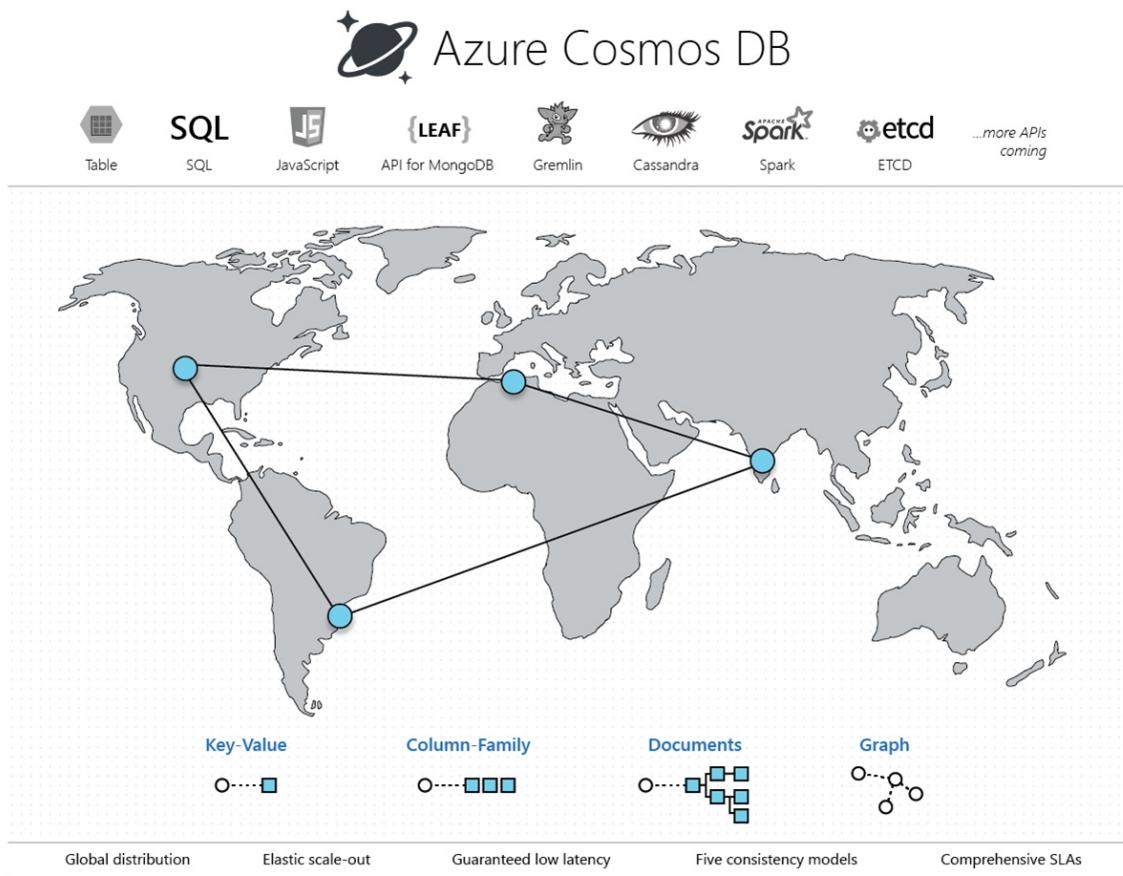
10/24/2019 • 6 minutes to read • [Edit Online](#)

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 Microsoft's globally distributed, multi-model database service. With a click of a button, Cosmos DB enables you to elastically and independently scale throughput and storage across any number of Azure regions worldwide. You can elastically scale throughput and storage, and take advantage of fast, single-digit-millisecond data access using your favorite API including SQL, MongoDB, Cassandra, Tables, or Gremlin. Cosmos DB provides comprehensive [service level agreements](#) (SLAs) for throughput, latency, availability, and consistency guarantees, something no other database service offers.

You can [Try Azure Cosmos DB for Free](#) without an Azure subscription, free of charge and commitments.

[Try Azure Cosmos DB for Free](#)



## Key Benefits

### Turnkey global distribution

Cosmos DB enables you to build highly responsive and highly available applications worldwide. Cosmos DB transparently replicates your data wherever your users are, so your users can interact with

a replica of the data that is closest to them.

Cosmos DB allows you to add or remove any of the Azure regions to your Cosmos account at any time, with a click of a button. Cosmos DB will seamlessly replicate your data to all the regions associated with your Cosmos account while your application continues to be highly available, thanks to the *multi-homing* capabilities of the service. For more information, see the [global distribution](#) article.

### **Always On**

By virtue of deep integration with Azure infrastructure and [transparent multi-master replication](#), Cosmos DB provides [99.999% high availability](#) for both reads and writes. Cosmos DB also provides you with the ability to programmatically (or via Portal) invoke the regional failover of your Cosmos account. This capability helps ensure that your application is designed to failover in the case of regional disaster.

### **Elastic scalability of throughput and storage, worldwide**

Designed with transparent horizontal partitioning and multi-master replication, Cosmos DB offers unprecedented elastic scalability for your writes and reads, all around the globe. You can elastically scale up from thousands to hundreds of millions of requests/sec around the globe, with a single API call and pay only for the throughput (and storage) you need. This capability helps you to deal with unexpected spikes in your workloads without having to over-provision for the peak. For more information, see [partitioning in Cosmos DB](#), [provisioned throughput on containers and databases](#), and [scaling provisioned throughput globally](#).

### **Guaranteed low latency at 99th percentile, worldwide**

Using Cosmos DB, you can build highly responsive, planet scale applications. With its novel multi-master replication protocol and latch-free and [write-optimized database engine](#), Cosmos DB guarantees less than 10-ms latencies for both, reads (indexed) and writes at the 99th percentile, all around the world. This capability enables sustained ingestion of data and blazing-fast queries for highly responsive apps.

### **Precisely defined, multiple consistency choices**

When building globally distributed applications in Cosmos DB, you no longer have to make extreme [tradeoffs between consistency, availability, latency, and throughput](#). Cosmos DB's multi-master replication protocol is carefully designed to offer [five well-defined consistency choices](#) - *strong*, *bounded staleness*, *session*, *consistent prefix*, and *eventual* — for an intuitive programming model with low latency and high availability for your globally distributed application.

### **No schema or index management**

Keeping database schema and indexes in-sync with an application's schema is especially painful for globally distributed apps. With Cosmos DB, you do not need to deal with schema or index management. The database engine is fully schema-agnostic. Since no schema and index management is required, you also don't have to worry about application downtime while migrating schemas. Cosmos DB [automatically indexes all data](#) and serves queries fast.

### **Battle tested database service**

Cosmos DB is a foundational service in Azure. For nearly a decade, Cosmos DB has been used by many of Microsoft's products for mission critical applications at global scale, including Skype, Xbox, Office 365, Azure, and many others. Today, Cosmos DB is one of the fastest growing services on Azure, used by many external customers and mission-critical applications that require elastic scale, turnkey global distribution, multi-master replication for low latency and high availability of both reads and writes.

### **Ubiquitous regional presence**

Cosmos DB is available in all Azure regions worldwide, including 54+ regions in public cloud, [Azure](#)

China 21Vianet, Azure Germany, Azure Government, and Azure Government for Department of Defense (DoD). See [Cosmos DB's regional presence](#).

### Secure by default and enterprise ready

Cosmos DB is certified for a [wide array of compliance standards](#). Additionally, all data in Cosmos DB is encrypted at rest and in motion. Cosmos DB provides row level authorization and adheres to strict security standards.

### Significant TCO savings

Since Cosmos DB is a fully managed service, you no longer need to manage and operate complex multi datacenter deployments and upgrades of your database software, pay for the support, licensing, or operations or have to provision your database for the peak workload. For more information, see [Optimize cost with Cosmos DB](#).

### Industry leading comprehensive SLAs

Cosmos DB is the first and only service to offer [industry-leading comprehensive SLAs](#) encompassing 99.999% high availability, read and write latency at the 99th percentile, guaranteed throughput, and consistency.

### Globally distributed operational analytics and AI with natively built-in Apache Spark

You can run [Spark](#) directly on data stored in Cosmos DB. This capability allows you to do low-latency, operational analytics at global scale without impacting transactional workloads operating directly against Cosmos DB. For more information, see [Globally distributed operational analytics](#).

### Develop applications on Cosmos DB using popular Open Source Software (OSS) APIs

Cosmos DB offers a choice of APIs to work with your data stored in your Cosmos database. By default, [you can use SQL](#) (a core API) for querying your Cosmos database. Cosmos DB also implements APIs for [Cassandra](#), [MongoDB](#), [Gremlin](#) and [Azure Table Storage](#). You can point client drivers (and tools) for the commonly used NoSQL (e.g., MongoDB, Cassandra, Gremlin) directly to your Cosmos database. By supporting the wire protocols of commonly used NoSQL APIs, Cosmos DB allows you to:

- 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 a fully-managed cloud service with industry leading, financially backed SLAs for the common NoSQL APIs.
- Elastically scale the provisioned throughput and storage for your databases based on your need and pay only for the throughput and storage you need. This leads to significant cost savings.

## 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

Read more about Cosmos DB's core concepts [turnkey 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](#)

# Quickstart: Create an Azure Cosmos account, database, container, and items from the Azure portal

2/21/2020 • 5 minutes to read • [Edit Online](#)

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.

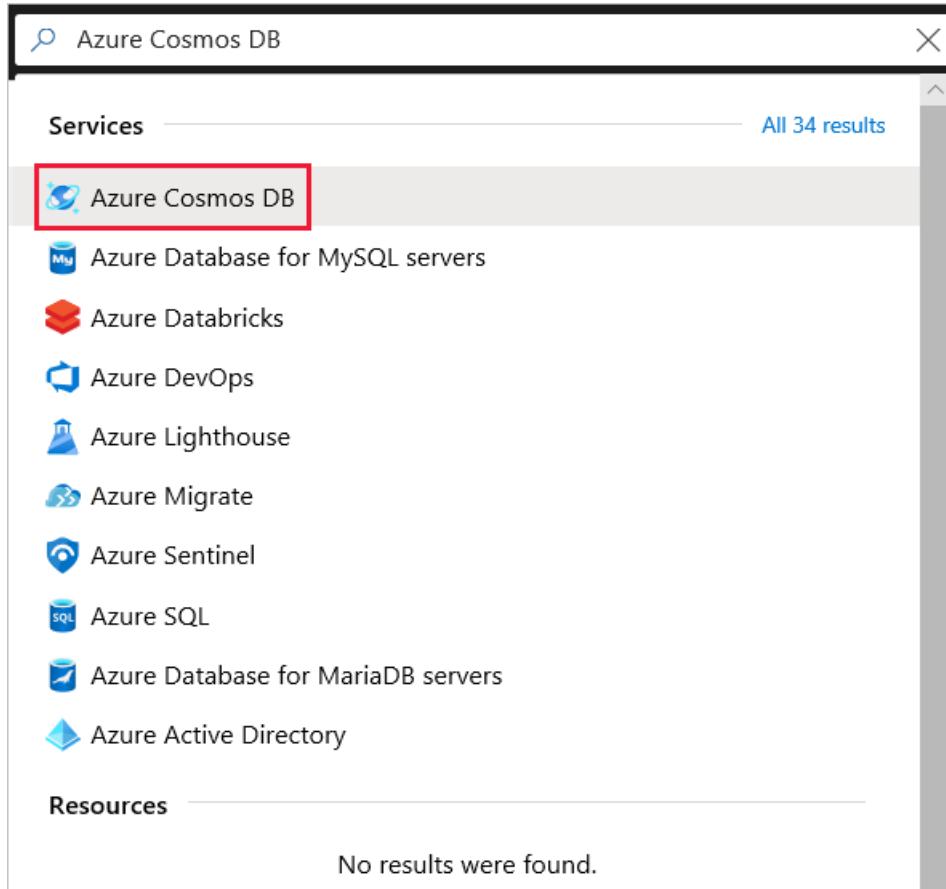
## Prerequisites

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, you can 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

1. Go to the [Azure portal](#) to create an Azure Cosmos DB account. Search for and select **Azure Cosmos DB**.



2. Select **Add**.

3. 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 <b>Create new</b> , then enter a unique name for the new resource group.
Account Name	A unique name	<p>Enter a name to identify your Azure Cosmos account. Because <i>documents.azure.com</i> is appended to the name that you provide to create your URI, use a unique name.</p> <p>The name can only contain lowercase letters, numbers, and the hyphen (-) character. It must be between 3-31 characters in length.</p>
API	The type of account to create	<p>Select <b>Core (SQL)</b> to create a document database and query by using SQL syntax.</p> <p>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.</p> <p><a href="#">Learn more about the SQL API.</a></p>
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.

## Create Azure Cosmos DB Account

[Basics](#) [Network](#) [Tags](#) [Review + create](#)

Azure Cosmos DB is a fully managed globally distributed, multi-model database service, transparently replicating your data across any number of Azure regions. You can elastically scale throughput and storage, and take advantage of fast, single-digit-millisecond data access using the API of your choice backed by 99.999 SLA. [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	Contoso Subscription
	▼
	(New) myResourceGroup
	▼
	<a href="#">Create new</a>

### INSTANCE DETAILS

* Account Name	mysqlapicosmosdb	✓
	documents.azure.com	
* API ⓘ	Core (SQL)	▼
* Location	West US	▼
Geo-Redundancy ⓘ	<a href="#">Enable</a>	<a href="#">Disable</a>
Multi-region Writes ⓘ	<a href="#">Enable</a>	<a href="#">Disable</a>

[Review + create](#)

[Previous](#)

[Next: Network](#)

4. Select **Review + create**. You can skip the **Network** and **Tags** sections.

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**.

Deployment

Search (Ctrl+ /) <>

[Delete](#) [Cancel](#) [Redeploy](#) [Refresh](#)

**Overview** [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	<a href="#">Operation details</a>

6. Select **Go to resource** to go to the Azure Cosmos DB account page.

The screenshot shows the 'Quick start' section of the Azure Cosmos DB account 'mysqlapicosmosdb'. On the left, a sidebar lists navigation options like Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Quick start (which is selected), Notifications, Data Explorer, Settings, Replicate data globally, Default consistency, and Firewall and virtual networks. The main content area displays a message: 'Congratulations! Your Azure Cosmos DB account was created.' It then instructs to connect using a sample app, showing platform selection (.NET, .NET Core, Xamarin, Java, Node.js, Python). Step 1, 'Add a collection', guides the user to create an 'Items' collection with 10GB storage and 400 RU/s throughput. Step 2, 'Download and run your .NET app', provides a download link for a sample .NET application.

## 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.

The screenshot shows the 'Data Explorer' section of the Azure Cosmos DB account 'mycosmosdbsqlapi'. The left sidebar includes 'Overview', 'Activity log', 'Access control (IAM)', 'Tags', 'Diagnose and solve problems', 'Quick start', 'Notifications', and 'Data Explorer' (which is selected). A red box highlights the 'Data Explorer' link. The main area shows the 'SQL API' and an 'Add Container' dialog. The dialog has a red border and contains fields for 'Database id' (set to 'Create new' with value 'ToDoList'), 'Provision database throughput' (checked), 'Throughput' (set to 400), and 'Container id' (set to 'Items'). It also includes a 'Partition key' field ('/category') and a note about unique keys. At the bottom is a large red box around the 'OK' button.

2. In the **Add container** pane, enter the settings for the new container.

SETTING	SUGGESTED VALUE	DESCRIPTION
Database ID	ToDoList	Enter <i>ToDoList</i> 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 <b>Provision database throughput</b> 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 <i>Items</i> 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 <i>/category</i> as the partition key.

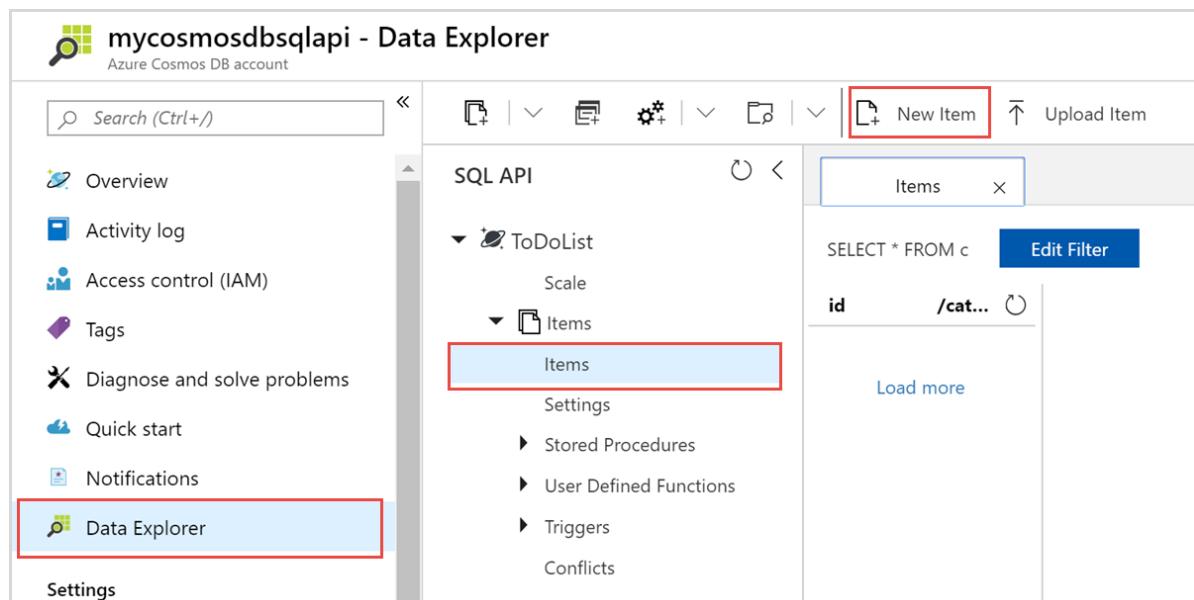
Don't add **Unique keys** for this example. Unique keys let you add a layer of data integrity to the database by ensuring the uniqueness of one or more values per partition key. For more information, see [Unique keys in Azure Cosmos DB](#).

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**.

The screenshot shows the Azure Cosmos DB Data Explorer interface. On the left, the navigation pane is open with the path: SQL API > ToDoList > Scale > Items > Items. The 'Items' node is selected. In the center, a table titled 'Items' displays a single document. The table has two columns: 'id' and '/cat...'. The 'id' column shows the document ID (1). The '/cat...' column shows the JSON document content. A red box highlights the 'Save' button in the top right corner of the interface.

id	/cat...
1	<pre> 1 {   "id": "1",   "category": "personal",   "name": "groceries",   "description": "Pick up apples and strawberries.",   "isComplete": false } </pre>

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.

- At the top of the **Documents** tab in Data Explorer, review the default query `SELECT * FROM c`. This query retrieves and displays all documents in the collection in ID order.

The screenshot shows the Azure Cosmos DB Data Explorer interface with the 'Documents' tab selected. At the top, the query `SELECT * FROM c` is highlighted with a red box. The results table shows two documents. The first document has an `id` of 1 and a `category` of 'personal'. The second document has an `id` of 2 and a `category` of 'business'. Below the table, a detailed view of the first document's properties is shown. A red box highlights the first few lines of the JSON document. The properties listed include `id`, `category`, `name`, `description`, `isComplete`, `_rid`, `_self`, `_etag`, and `attachments`.

id	/cate...
1	personal
2	business

- To change the query, select **Edit Filter**, replace the default query with `ORDER BY c._ts DESC`, and then select **Apply Filter**.

The screenshot shows the Azure Data Explorer interface with a query editor window. The query is:

```
SELECT * FROM c ORDER BY c._ts DESC
```

The results pane shows a table with columns `id` and `/category`. The first two rows are highlighted with a red border:

id	/category
2	business
1	personal

An `Edit Filter` button is visible above the results. A blue `Apply Filter` button is located at the bottom right of the query editor.

The modified query displays the documents in descending order based on their time stamp, so now your second document is listed first.

The screenshot shows the Azure Data Explorer interface with the modified query results. The results pane displays the JSON representation of the document with `id: 2` and `category: business`.

```
1: { "id": "2", "category": "business", "name": "meetings", "description": "Meet with Britta.", "isComplete": false, "rid": "mk00ALLgQtYCAAAAAAAA==", "self": "dbs/mk00AA==/colls/mk00ALLgQtY=/docs/mk00ALLgQtYCAAAAAAAA==", "etag": "\\"1000c09c-0000-0700-0000-5c9433560000\\\"", "attachments": "attachments/", "ts": 1553216342}
```

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

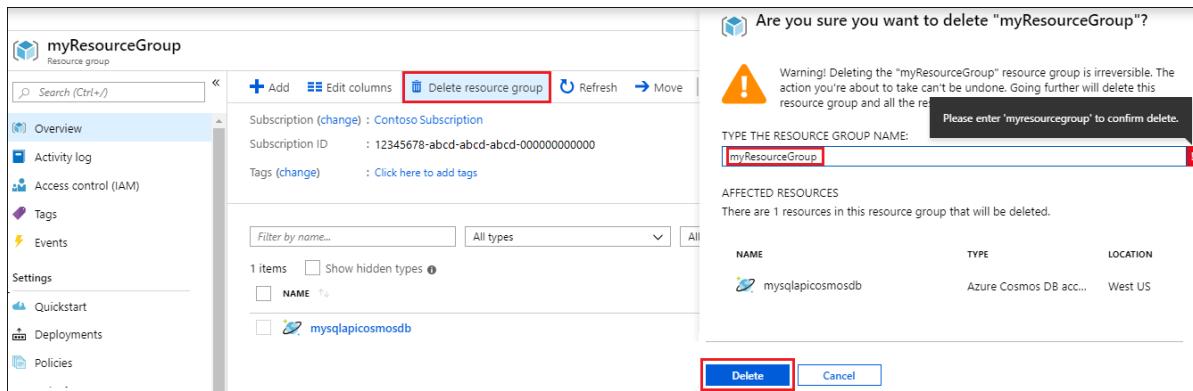
When you're done with your app and Azure Cosmos DB account, you can delete the Azure resources you created so you don't incur more charges. To delete the resources:

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.

The screenshot shows the Azure portal's Resource Groups blade. On the left, a list of resource groups is shown, with `myResourceGroup` selected and highlighted with a red border. The main pane shows the `Overview` page for `myResourceGroup`, displaying subscription information and tags.

Subscription (change)	: Contoso Subscription
Subscription ID	: 12345678-abcd-abcd-000000000000
Tags (change)	: Click here to add tags

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 Azure Resource Manager template

2/24/2020 • 5 minutes to read • [Edit Online](#)

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. This quickstart focuses on the process of deploying a Resource Manager template to create an Azure Cosmos database and a container within that database. You can later store data in this container.

[Resource Manager template](#) is a JavaScript Object Notation (JSON) file that defines the infrastructure and configuration for your project. The template uses declarative syntax, which lets you state what you intend to deploy without having to write the sequence of programming commands to create it. If you want to learn more about developing Resource Manager templates, see [Resource Manager documentation](#) and the [template reference](#).

If you don't have an Azure subscription, create a [free account](#) before you begin.

## 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, you can 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 account, database, container

The template used in this quickstart is from [Azure Quickstart templates](#).

```
{  
  "$schema": "https://schema.management.azure.com/schemas/2015-01-01/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."  
      }  
    }  
  },  
  "resources": [  
    {  
      "type": "Microsoft.DocumentDB/databaseAccounts/sqlDatabases",  
      "name": "[parameters('accountName')]",  
      "apiVersion": "2017-02-28",  
      "location": "[parameters('location')]",  
      "dependsOn": [],  
      "properties": {  
        "name": "[parameters('accountName')]",  
        "locations": [  
          {  
            "name": "[parameters('primaryRegion')]",  
            "isPrimary": true  
          }  
        ],  
        "enableAutomaticFailover": true,  
        "maxThroughput": 400  
      }  
    },  
    {  
      "type": "Microsoft.DocumentDB/databaseAccounts/sqlDatabases/containers",  
      "name": "[concat(parameters('accountName'), '/sqlDatabases/[parameters('name')]/containers/[resourceName]')]",  
      "apiVersion": "2017-02-28",  
      "location": "[parameters('location')]",  
      "dependsOn": ["[resourceId('Microsoft.DocumentDB/databaseAccounts/sqlDatabases', parameters('accountName'))]"],  
      "properties": {  
        "name": "[resourceName]",  
        "partitionKeyPath": "/id",  
        "ttl": 30  
      }  
    }  
  ]  
}
```

```
"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."
    }
},
"multipleWriteLocations": {
    "type": "bool",
    "defaultValue": false,
    "allowedValues": [
        true,
        false
    ],
    "metadata": {
        "description": "Enable multi-master to make all regions writable."
    }
},
"automaticFailover": {
    "type": "bool",
    "defaultValue": false,
    "allowedValues": [
        true,
        false
    ],
    "metadata": {
        "description": "Enable automatic failover for regions. Ignored when Multi-Master is enabled"
    }
},
"databaseName": {
    "type": "string",
    "defaultValue": "Database1",
    "metadata": {
        "description": "The name for the Azure Cosmos database"
    }
},
"containerName": {
    "type": "string",
    "defaultValue": "Container1",
    "metadata": {
        "description": "The name for the container with dedicated throughput"
    }
},
"throughput": {
    "type": "int",
    "defaultValue": 400,
    " minValue": 400,
    "maxValue": 1000000,
    "metadata": {
        "description": "The throughput for the container with dedicated throughput"
    }
},
"variables": {
    "accountName": "[toLowerCase(parameters('accountName'))]",
    "consistencyPolicy": {
        "Session": {
            "defaultConsistencyLevel": "Session"
```

```
        },
      },
      "locations": [
        {
          "locationName": "[parameters('primaryRegion')]",
          "failoverPriority": 0,
          "isZoneRedundant": false
        },
        {
          "locationName": "[parameters('secondaryRegion')]",
          "failoverPriority": 1,
          "isZoneRedundant": false
        }
      ]
    },
    "resources": [
      {
        "type": "Microsoft.DocumentDB/databaseAccounts",
        "name": "[variables('accountName')]",
        "apiVersion": "2019-08-01",
        "kind": "GlobalDocumentDB",
        "location": "[parameters('location')]",
        "properties": {
          "consistencyPolicy": "[variables('consistencyPolicy')[parameters('defaultConsistencyLevel')]]",
          "locations": "[variables('locations')]",
          "databaseAccountOfferType": "Standard",
          "enableAutomaticFailover": "[parameters('automaticFailover')]",
          "enableMultipleWriteLocations": "[parameters('multipleWriteLocations')]"
        }
      },
      {
        "type": "Microsoft.DocumentDB/databaseAccounts/sqlDatabases",
        "name": "[concat(variables('accountName'), '/', parameters('databaseName'))]",
        "apiVersion": "2019-08-01",
        "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": "2019-08-01",
        "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/*"
        }
      ]
    },
    "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](#).

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.

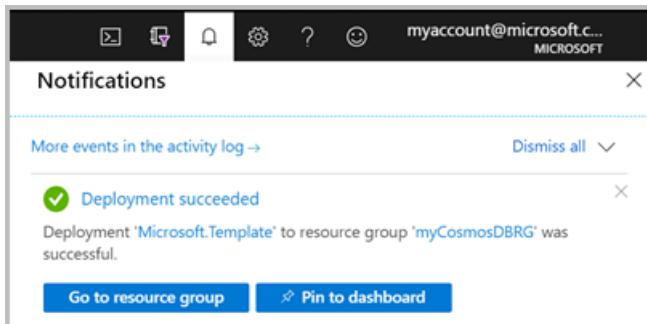
The screenshot shows the Azure portal interface for creating an Azure Cosmos account. At the top, there's a breadcrumb navigation: Home > Create an Azure Cosmos account for SQL API. Below it, the title is 'Create an Azure Cosmos account for SQL API' with a note 'Azure quickstart template'. A 'TEMPLATE' section shows a purple icon for '101-cosmosdb-create' and '3 resources'. There are three buttons: 'Edit template', 'Edit param...', and 'Learn more'. The main form is divided into sections:

- BASICS**:
  - Subscription: Content Testing
  - Resource group: (New) myCosmosDBRG or Create new
  - Location: (US) East US
- SETTINGS**:
  - Account Name: myCosmosDBAccount
  - Location: East US
  - Primary Region: East US
  - Secondary Region: West US
  - Default Consistency Level: Session
  - Multiple Write Locations: false
  - Automatic Failover: false
  - Database Name: Database1
  - Container Name: Container1
  - Throughput: 400
- TERMS AND CONDITIONS**:
  - Links: Template information | Azure Marketplace Terms | Azure Marketplace
  - A detailed terms and conditions text box is present.
  - A checkbox: I agree to the terms and conditions stated above.
- Purchase**: A blue button at the bottom.

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.
- **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 on to work with subsequent and tutorials, you may wish 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 Azure Resource Manager 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.

2/24/2020 • 13 minutes to read • [Edit Online](#)

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 globally distributed multi-model database service. 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-master 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 in the following article.

[Import data into Azure Cosmos DB](#)

# Quickstart: Build a .NET console app to manage Azure Cosmos DB SQL API resources

12/2/2019 • 12 minutes to read • [Edit Online](#)

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 globally distributed multi-model database service. 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-master 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 Microsoft.Azure.Cosmos` 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 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

2/24/2020 • 13 minutes to read • [Edit Online](#)

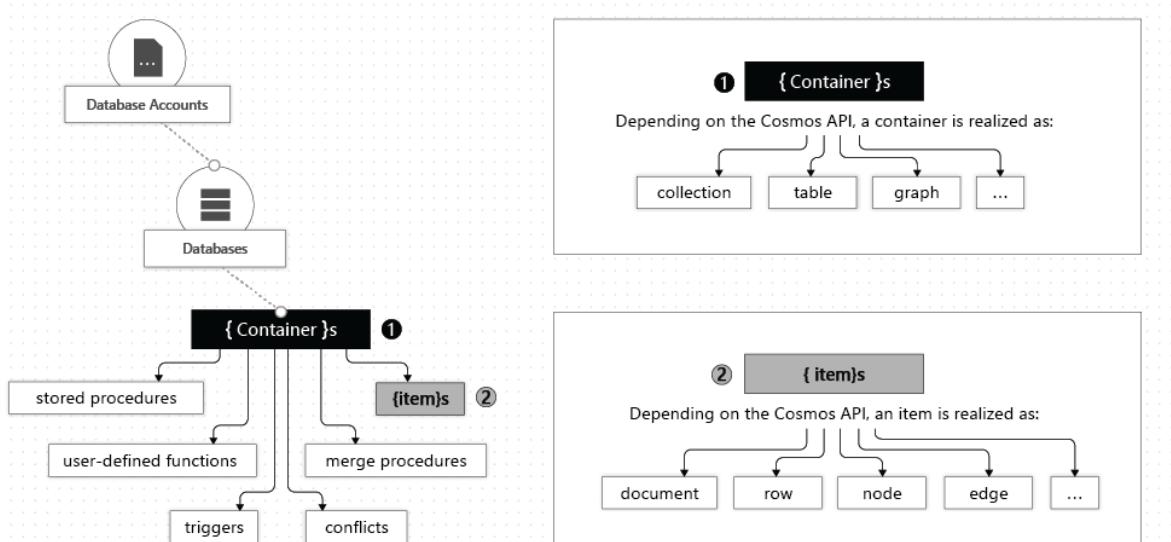
In this quickstart, you create and manage an Azure Cosmos DB SQL API account from the Azure portal, and by using a Java app cloned from GitHub. First, you create an Azure Cosmos DB SQL API account using the Azure portal, then create a Java app using the SQL Java SDK, and then add resources to your Cosmos 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.

## 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+ob0N8A7Cgv30VRDJWEHLM+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*.

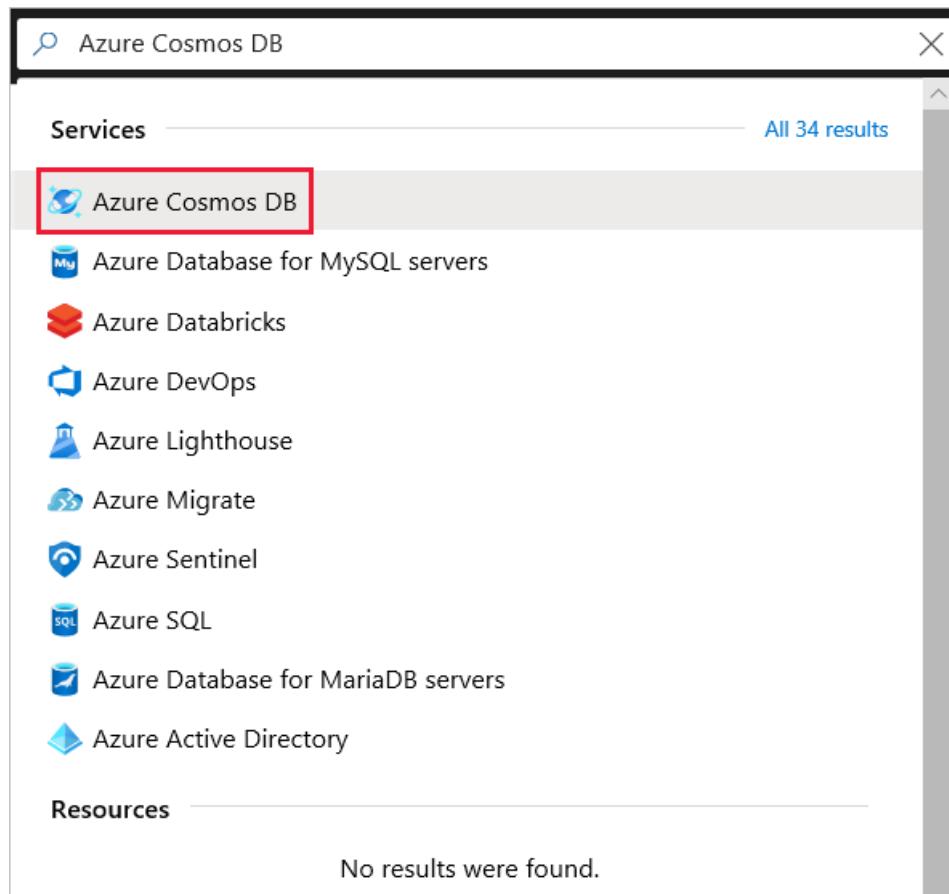
The provisioned throughput is measured in Request Units (*RUs*) which have a monetary price and are a substantial determining factor in the operating cost of the account. Provisioned throughput can be selected at per-container granularity or per-database granularity, however container-level throughput 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. Go to the [Azure portal](#) to create an Azure Cosmos DB account. Search for and select **Azure Cosmos DB**.



The screenshot shows the Azure portal's search interface. The search bar at the top contains the text "Azure Cosmos DB". Below the search bar, there are two sections: "Services" and "Resources". In the "Services" section, the "Azure Cosmos DB" result is highlighted with a red box. Other listed services include Azure Database for MySQL servers, Azure Databricks, Azure DevOps, Azure Lighthouse, Azure Migrate, Azure Sentinel, Azure SQL, Azure Database for MariaDB servers, and Azure Active Directory. In the "Resources" section, it says "No results were found."

2. Select **Add**.
3. 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 <b>Create new</b> , then enter a unique name for the new resource group.

Setting	Value	Description
Account Name	A unique name	<p>Enter a name to identify your Azure Cosmos account. Because <i>documents.azure.com</i> is appended to the name that you provide to create your URI, use a unique name.</p> <p>The name can only contain lowercase letters, numbers, and the hyphen (-) character. It must be between 3-31 characters in length.</p>
API	The type of account to create	<p>Select <b>Core (SQL)</b> to create a document database and query by using SQL syntax.</p> <p>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.</p> <p><a href="#">Learn more about the SQL API.</a></p>
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.

## Create Azure Cosmos DB Account

[Basics](#) [Network](#) [Tags](#) [Review + create](#)

Azure Cosmos DB is a fully managed globally distributed, multi-model database service, transparently replicating your data across any number of Azure regions. You can elastically scale throughput and storage, and take advantage of fast, single-digit-millisecond data access using the API of your choice backed by 99.999 SLA. [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	Contoso Subscription	▼
└─ * Resource Group	(New) myResourceGroup	▼
	<a href="#">Create new</a>	

### INSTANCE DETAILS

* Account Name	mysqlapicosmosdb	✓
	documents.azure.com	
* API ⓘ	Core (SQL)	▼
* Location	West US	▼
Geo-Redundancy ⓘ	<a href="#">Enable</a> <a href="#">Disable</a>	
Multi-region Writes ⓘ	<a href="#">Enable</a> <a href="#">Disable</a>	

[Review + create](#)

[Previous](#)

[Next: Network](#)

4. Select **Review + create**. You can skip the **Network** and **Tags** sections.

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**.

Dashboard > Microsoft.Azure.CosmosDB-2019032100000 - Overview

**Microsoft.Azure.CosmosDB-2019032100000 - Overview**

Deployment

Search (Ctrl+/  
)

Delete Cancel Redeploy Refresh

**Overview** **Inputs** **Outputs** **Template**

✓ Your deployment is complete

Go to resource

Deployment name: Microsoft.Azure.CosmosDB-2019032100000  
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	<a href="#">Operation details</a>

6. Select **Go to resource** to go to the Azure Cosmos DB account page.

Congratulations! Your Azure Cosmos DB account was created.

Now, let's connect to it using a sample app:

**Choose a platform**

- .NET
- .NET Core
- Xamarin
- Java
- Node.js
- Python

**1 Add a collection**

In Azure Cosmos DB, data is stored in collections.

**Create 'Items' collection**

Create 'Items' collection with 10GB storage capacity and 400 Request Units per second (RU/s) throughput capacity, for free!

**2 Download and run your .NET app**

Once collection is created, download a sample .NET app connected to it, extract, build and run.

**Download**

## 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.

**New Container**

**Add Container**

**Database id**  Create new  Use existing  
Tasks

Provision database throughput

**Throughput (400 - 100,000 RU/s)**  Manual  
Autopilot (preview)

400

Estimated spend (USD): \$<price>hourly / \$<price>daily / \$<price>monthly (2 regions, 400RU/s, \$<price>/RU)

**Container id** Items

**Partition key** /category

My partition key is larger than 100 bytes

Unique keys

+ Add unique key

### 2. In the **Add container** page, enter the settings for the new container.

SETTING	SUGGESTED VALUE	DESCRIPTION
---------	-----------------	-------------

SETTING	SUGGESTED VALUE	DESCRIPTION
<b>Database ID</b>	Tasks	Enter <i>Tasks</i> 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 <b>Provision database throughput</b> 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.
<b>Throughput</b>	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.
<b>Container ID</b>	Items	Enter <i>Items</i> as the name for your new container. Container IDs have the same character requirements as database names.
<b>Partition key</b>	/category	The sample described in this article uses <i>/category</i> 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**.

The screenshot shows the Azure Cosmos DB Data Explorer interface. On the left, there's a sidebar with various navigation options like Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Quick start, Notifications, and Data Explorer. The Data Explorer option is highlighted with a red box. Below it are Settings, Replicate data globally, and Default consistency. The main area is titled 'SQL API' and shows a tree view under 'FamilyDatabase' with 'Tasks' expanded, showing 'Items' selected. A red box highlights the 'Items' node. To the right, there's a query editor with 'SELECT \* FROM c' and an 'Edit Filter' button. Below the query is a table with columns 'id' and '/cate...', and a 'Load more' button.

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**.

The screenshot shows the Azure Cosmos DB Data Explorer interface. The 'Documents' tab is selected, showing a list of documents. One document is highlighted with a red box. The JSON content of the document is displayed in the preview pane:

```
1  {  
2      "id": "1",  
3      "category": "personal",  
4      "name": "groceries",  
5      "description": "Pick up apples and strawberries.",  
6      "isComplete": false  
7  }
```

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.

- At the top of the **Documents** tab in Data Explorer, review the default query `SELECT * FROM c`. This query retrieves and displays all documents in the collection in ID order.

The screenshot shows the Azure Data Explorer interface with the 'Documents' tab selected. At the top, there are buttons for 'New Document', 'Update', 'Discard', and 'Delete'. Below these are two tabs: 'SELECT \* FROM c' (which is highlighted with a red box) and 'Edit Filter'. The main area displays a table with columns 'id' and '/category...'. Two rows are visible: row 1 has id '1' and category 'personal'; row 2 has id '2' and category 'business'. A 'Load more' button is present. To the right of the table is a JSON document preview:

```
1 {
2   "id": "1",
3   "category": "personal",
4   "name": "groceries",
5   "description": "Pick up apples and strawberries.",
6   "isComplete": false,
7   "_rid": "a017ANxuEAABAAAAAA==",
8   "_self": "dbs/a017AA==/colls/a017ANxuEAA=/docs/a017ANxuEAABAAAAAA==/",
9   "_etag": "\"00007e09-0000-0000-0000-59e8aab80000\"",
10  "attachments": "attachments/".
```

- To change the query, select **Edit Filter**, replace the default query with `ORDER BY c._ts DESC`, and then select **Apply Filter**.

The screenshot shows the same Data Explorer interface. The 'Edit Filter' tab is now selected. The query in the predicate box is `SELECT * FROM c ORDER BY c._ts DESC` (with 'ORDER BY c.\_ts DESC' highlighted with a red box). There is a blue 'Apply Filter' button to the right of the predicate box.

The modified query displays the documents in descending order based on their time stamp, so now your second document is listed first.

The screenshot shows the results of the modified query. The table now lists the documents in descending order of timestamp. Row 2 (id '2', category 'business') is at the top, and row 1 (id '1', category 'personal') is below it. The JSON preview on the right shows the full document structure for both rows, including the timestamp field '\_ts':

```
1 {
2   "id": "2",
3   "category": "business",
4   "name": "meetings",
5   "description": "Meet with Britta.",
6   "isComplete": false,
7   "_rid": "mk00ALLgQtYCAAAAAAAA==",
8   "_self": "dbs/mk00AA==/colls/mk00ALLgQtY=/docs/mk00ALLgQtYCAAAAAAAA==",
9   "_etag": "\"1000c09c-0000-0700-0000-5c9433560000\"",
10  "attachments": "attachments/",
11  "_ts": 1553216342
```

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](#).

### 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()  
    .setEndpoint(AccountSettings.HOST)  
    .setKey(AccountSettings.MASTER_KEY)  
    .setConnectionPolicy(defaultPolicy)  
    .setConsistencyLevel(ConsistencyLevel.EVENTUAL)  
    .buildClient();
```

- `CosmosDatabase` creation.

```
database = client.createDatabaseIfNotExists(databaseName).getDatabase();
```

- `CosmosContainer` creation.

```
CosmosContainerProperties containerProperties =  
    new CosmosContainerProperties(containerName, "/lastName");  
  
// Create container with 400 RU/s  
container = database.createContainerIfNotExists(containerProperties, 400).getContainer();
```

- 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.getRequestLatency();  
    System.out.println(String.format("Item successfully read with id %s with a charge of %.2f and  
    within duration %s",  
        item.getResource().getId(), requestCharge, requestLatency));  
} catch (CosmosClientException e) {  
    e.printStackTrace();  
    System.err.println(String.format("Read Item failed with %s", e));  
}
```

- SQL queries over JSON are performed using the `queryItems` method.

```
// Set some common query options
FeedOptions queryOptions = new FeedOptions();
queryOptions.setMaxItemCount(10);
//queryOptions.setEnableCrossPartitionQuery(true); //No longer necessary in SDK v4
// Set populate query metrics to get metrics around query executions
queryOptions.populateQueryMetrics(true);

CosmosContinuablePagedIterable<Family> familiesPagedIterable = container.queryItems(
    "SELECT * FROM Family WHERE Family.lastName IN ('Andersen', 'Wakefield', 'Johnson')",
    queryOptions, Family.class);

familiesPagedIterable.iterableByPage().forEach(cosmosItemPropertiesFeedResponse -> {
    System.out.println("Got a page of query result with " +
        cosmosItemPropertiesFeedResponse.getResults().size() + " items(s)"
        + " and request charge of " + cosmosItemPropertiesFeedResponse.getRequestCharge());

    System.out.println("Item Ids " + cosmosItemPropertiesFeedResponse
        .getResults()
        .stream()
        .map(Family::getId)
        .collect(Collectors.toList()));
});

});
```

## Managing database resources using the asynchronous (async) API

- Async API calls return immediately, without waiting for a response from the server. In light of this, the following code snippets show proper design patterns for accomplishing all of the preceding management tasks using async API.
- `CosmosAsyncClient` initialization. The `CosmosAsyncClient` provides client-side logical representation for the Azure Cosmos database service. This client is used to configure and execute asynchronous requests against the service.

```
client = new CosmosClientBuilder()
    .setEndpoint(AccountSettings.HOST)
    .setKey(AccountSettings.MASTER_KEY)
    .setConnectionPolicy(defaultPolicy)
    .setConsistencyLevel(ConsistencyLevel.EVENTUAL)
    .buildAsyncClient();
```

- `CosmosAsyncDatabase` creation.

```
Mono<CosmosAsyncDatabaseResponse> databaseIfNotExists =
client.createDatabaseIfNotExists(databaseName);
databaseIfNotExists.flatMap(databaseResponse -> {
    database = databaseResponse.getDatabase();
    System.out.println("Checking database " + database.getId() + " completed!\n");
    return Mono.empty();
}).block();
```

- `CosmosAsyncContainer` creation.

```

CosmosContainerProperties containerProperties = new CosmosContainerProperties(containerName,
"/lastName");
Mono<CosmosAsyncContainerResponse> containerIfNotExists =
database.createContainerIfNotExists(containerProperties, 400);

// Create container with 400 RU/s
containerIfNotExists.flatMap(containerResponse -> {
    container = containerResponse.getContainer();
    System.out.println("Checking container " + container.getId() + " completed!\n");
    return Mono.empty();
}).block();

```

- As with the sync API, item creation is accomplished using the `createItem` method. This example shows how to efficiently issue numerous async `createItem` requests by subscribing to a Reactive Stream which issues the requests and prints notifications. Since this simple example runs to completion and terminates, `CountDownLatch` instances are used to ensure the program does not terminate during item creation. **The proper asynchronous programming practice is not to block on async calls - in realistic use-cases requests are generated from a main() loop that executes indefinitely, eliminating the need to latch on async calls.**

```

final CountDownLatch completionLatch = new CountDownLatch(1);

// Combine multiple item inserts, associated success println's, and a final aggregate stats println
// into one Reactive stream.
families.flatMap(family -> {
    return container.createItem(family);
}) //Flux of item request responses
.flatMap(itemResponse -> {
    System.out.println(String.format("Created item with request charge of %.2f within" +
        " duration %s",
        itemResponse.getRequestCharge(), itemResponse.getRequestLatency()));
    System.out.println(String.format("Item ID: %s\n", itemResponse.getResource().getId()));
    return Mono.just(itemResponse.getRequestCharge());
}) //Flux of request charges
.reduce(0.0,
    (charge_n,charge_nplus1) -> charge_n + charge_nplus1
) //Mono of total charge - there will be only one item in this stream
.subscribe(charge -> {
    System.out.println(String.format("Created items with total request charge of %.2f\n",
        charge));
},
err -> {
    if (err instanceof CosmosClientException) {
        //Client-specific errors
        CosmosClientException cerr = (CosmosClientException)err;
        cerr.printStackTrace();
        System.err.println(String.format("Read Item failed with %s\n", cerr));
    } else {
        //General errors
        err.printStackTrace();
    }
}

completionLatch.countDown();
},
() -> {completionLatch.countDown();}
); //Preserve the total charge and print aggregate charge/item count stats.

try {
    completionLatch.await();
} catch (InterruptedException err) {
    throw new AssertionError("Unexpected Interruption",err);
}

```

- As with the sync API, point reads are performed using `readItem` method.

```

final CountDownLatch completionLatch = new CountDownLatch(1);

familiesToCreate.flatMap(family -> {
    Mono<CosmosAsyncItemResponse<Family>> asyncItemResponseMono =
    container.readItem(family.getId(), new PartitionKey(family.getLastName()), Family.class);
    return asyncItemResponseMono;
})
.subscribe(
    itemResponse -> {
        double requestCharge = itemResponse.getRequestCharge();
        Duration requestLatency = itemResponse.getRequestLatency();
        System.out.println(String.format("Item successfully read with id %s with a
charge of %.2f and within duration %s",
            itemResponse.getResource().getId(), requestCharge, requestLatency));
    },
    err -> {
        if (err instanceof CosmosClientException) {
            //Client-specific errors
            CosmosClientException cerr = (CosmosClientException)err;
            cerr.printStackTrace();
            System.err.println(String.format("Read Item failed with %s\n", cerr));
        } else {
            //General errors
            err.printStackTrace();
        }
    }

    completionLatch.countDown();
},
() -> {completionLatch.countDown();}
);

try {
    completionLatch.await();
} catch (InterruptedException err) {
    throw new AssertionError("Unexpected Interruption",err);
}

```

- As with the sync API, SQL queries over JSON are performed using the `queryItems` method.

```

// Set some common query options

FeedOptions queryOptions = new FeedOptions();
queryOptions.maxItemCount(10);
//queryOptions.setEnableCrossPartitionQuery(true); //No longer needed in SDK v4
// Set populate query metrics to get metrics around query executions
queryOptions.populateQueryMetrics(true);

CosmosContinuablePagedFlux<Family> pagedFluxResponse = container.queryItems(
    "SELECT * FROM Family WHERE Family.lastName IN ('Andersen', 'Wakefield', 'Johnson')",
    queryOptions, Family.class);

final CountDownLatch completionLatch = new CountDownLatch(1);

pagedFluxResponse.byPage().subscribe(
    fluxResponse -> {
        System.out.println("Got a page of query result with " +
            fluxResponse.getResults().size() + " items(s)"
            + " and request charge of " + fluxResponse.getRequestCharge());

        System.out.println("Item Ids " + fluxResponse
            .getResults()
            .stream()
            .map(Family::getId)
            .collect(Collectors.toList()));
    },
    err -> {
        if (err instanceof CosmosClientException) {
            //Client-specific errors
            CosmosClientException cerr = (CosmosClientException)err;
            cerr.printStackTrace();
            System.err.println(String.format("Read Item failed with %s\n", cerr));
        } else {
            //General errors
            err.printStackTrace();
        }
    }

    completionLatch.countDown();
},
() -> {completionLatch.countDown();}
);

try {
    completionLatch.await();
} catch (InterruptedException err) {
    throw new AssertionError("Unexpected Interruption",err);
}

```

## 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.

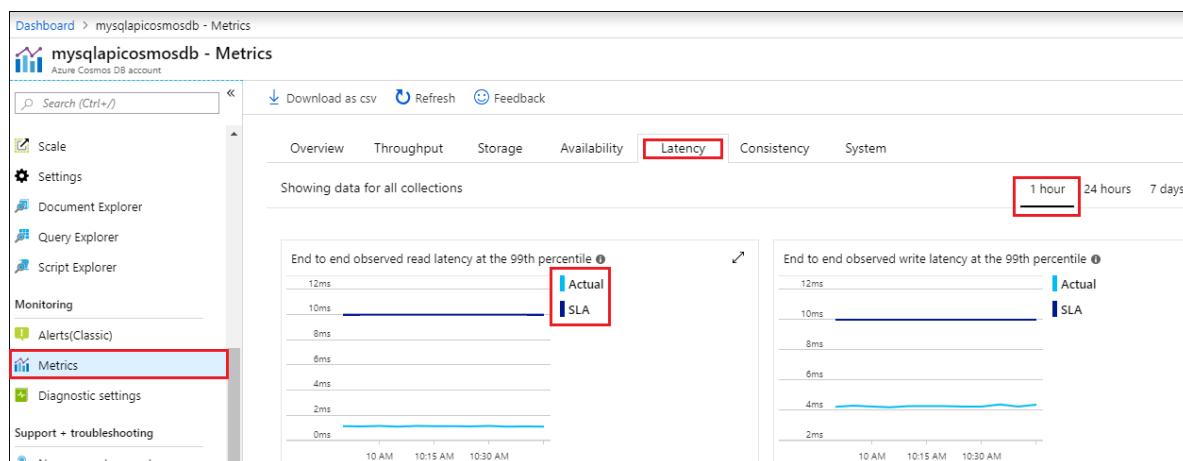
4. The app creates database with name `AzureSampleFamilyDB`
5. The app creates container with name `FamilyContainer`
6. The app will perform point reads using object IDs and partition key value (which is lastName in our sample).
7. The app will query items to retrieve all families with last name in ('Andersen', '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.

The screenshot shows the Azure Resource Groups blade. On the left, a list of resource groups is displayed, including 'myResourceGroupA', 'DefaultResourceGroup', 'DefaultResourceGroupA', and 'myResourceGroup'. The 'myResourceGroup' item is highlighted with a red border. On the right, the details for 'myResourceGroup' are shown, including its subscription information ('Contoso Subscription'), tags ('Click here to add tags'), and a list of resources under it. One resource, 'mysqlapicosmosdb', is listed as an 'Azure Cosmos DB account' located in 'West US'.

3. On the resource group **Overview** page, select **Delete resource group**.

The screenshot shows a confirmation dialog titled 'Are you sure you want to delete "myResourceGroup"?'. It includes a warning message about the irreversibility of the action, a field to type the resource group name ('myResourceGroup'), and a list of affected resources ('mysqlapicosmosdb'). The 'Delete' button is highlighted with a red border.

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: Use Node.js to connect and query data from Azure Cosmos DB SQL API account

2/27/2020 • 8 minutes to read • [Edit Online](#)

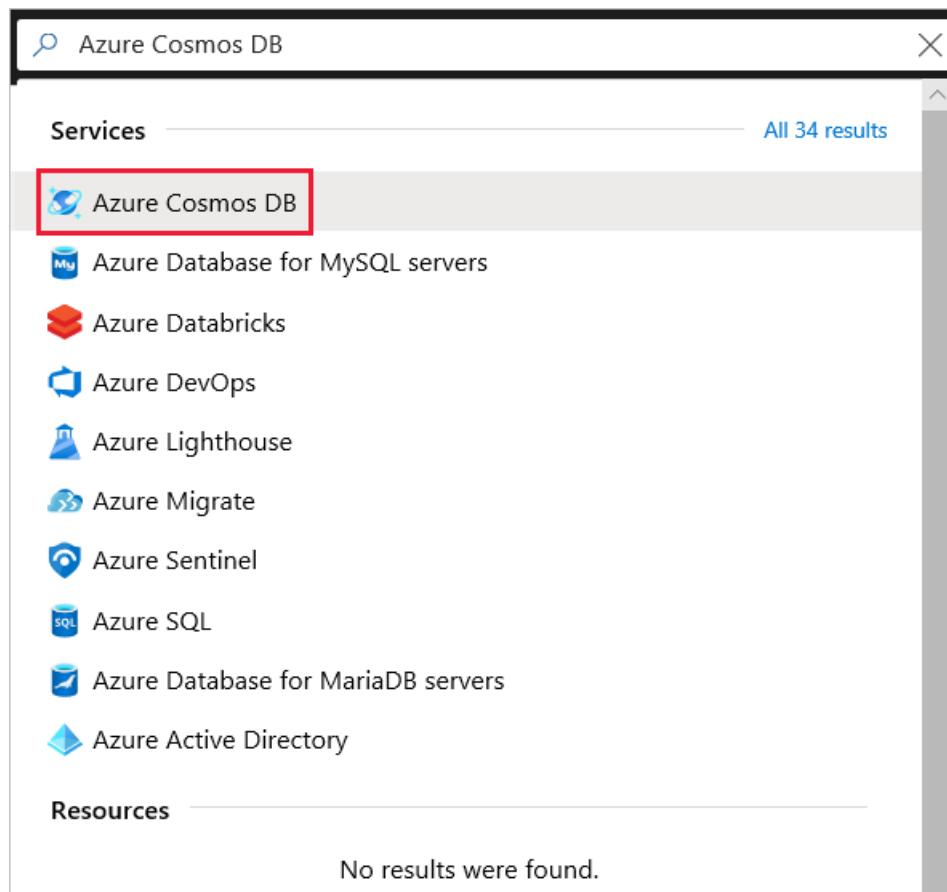
In this quickstart, you create and manage an Azure Cosmos DB SQL API account from the Azure portal, and by using a Node.js app cloned from GitHub. Azure Cosmos DB is a multi-model database service that lets you quickly create and query document, table, key-value, and graph databases with global distribution and horizontal scale capabilities.

## 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 a database

1. Go to the [Azure portal](#) to create an Azure Cosmos DB account. Search for and select **Azure Cosmos DB**.



2. Select **Add**.
3. 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 <b>Create new</b> , then enter a unique name for the new resource group.
Account Name	A unique name	<p>Enter a name to identify your Azure Cosmos account. Because <i>documents.azure.com</i> is appended to the name that you provide to create your URI, use a unique name.</p> <p>The name can only contain lowercase letters, numbers, and the hyphen (-) character. It must be between 3-31 characters in length.</p>
API	The type of account to create	<p>Select <b>Core (SQL)</b> to create a document database and query by using SQL syntax.</p> <p>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.</p> <p><a href="#">Learn more about the SQL API.</a></p>
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.

## Create Azure Cosmos DB Account

[Basics](#) [Network](#) [Tags](#) [Review + create](#)

Azure Cosmos DB is a fully managed globally distributed, multi-model database service, transparently replicating your data across any number of Azure regions. You can elastically scale throughput and storage, and take advantage of fast, single-digit-millisecond data access using the API of your choice backed by 99.999 SLA. [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.

<p>* Subscription</p> <p>Resource Group</p>	<div style="border: 1px solid #ccc; padding: 2px;">Contoso Subscription</div> <div style="border: 1px solid #ccc; padding: 2px;">(New) myResourceGroup</div> <a href="#">Create new</a>
---	---

### INSTANCE DETAILS

<p>* Account Name</p> <p>* API</p> <p>* Location</p> <p>Geo-Redundancy</p> <p>Multi-region Writes</p>	<div style="border: 1px solid #ccc; padding: 2px;">mysqlapicosmosdb</div> <div style="border: 1px solid #ccc; padding: 2px;">Core (SQL)</div> <div style="border: 1px solid #ccc; padding: 2px;">West US</div> <div style="border: 1px solid #ccc; padding: 2px; display: flex; justify-content: space-around;"><span>Enable</span> <span>Disable</span></div> <div style="border: 1px solid #ccc; padding: 2px; display: flex; justify-content: space-around;"><span>Enable</span> <span>Disable</span></div>
---	--

[Review + create](#)

[Previous](#)

[Next: Network](#)

4. Select **Review + create**. You can skip the **Network** and **Tags** sections.

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**.

Dashboard > Microsoft.Azure.CosmosDB-2019032100000 - Overview

**Microsoft.Azure.CosmosDB-2019032100000 - Overview**

Deployment

Search (Ctrl+ /) < Delete Cancel Redeploy Refresh

**Overview** **Inputs** **Outputs** **Template**

**✓ Your deployment is complete**

Go to resource

Deployment name: Microsoft.Azure.CosmosDB-2019032100000  
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	<a href="#">Operation details</a>

6. Select **Go to resource** to go to the Azure Cosmos DB account page.

Congratulations! Your Azure Cosmos DB account was created.

Now, let's connect to it using a sample app:

**Choose a platform**

- .NET
- .NET Core
- Xamarin
- Java
- Node.js
- Python

- 1 Add a collection**

In Azure Cosmos DB, data is stored in collections.

**Create 'Items' collection**

Create 'Items' collection with 10GB storage capacity and 400 Request Units per second (RU/s) throughput capacity, for free!

- 2 Download and run your .NET app**

Once collection is created, download a sample .NET app connected to it, extract, build and run.

**Download**

## 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.

**New Container**

**Add Container**

**Start at \$<price>/mo per database, multiple containers included**

**Database id**: Items

**Throughput**: 400 RU/s

**Container id**: Items

**Partition key**: /category

**Description**: Estimated spend (USD): \$<price>hourly / \$<price>daily / \$<price>monthly (2 regions, 400RU/s, \$<price>/RU)

**Tasks**: Provision database throughput

**Autopilot (preview)**: Manual

**My partition key is larger than 100 bytes**

### 2. In the **Add container** page, enter the settings for the new container.

SETTING	SUGGESTED VALUE	DESCRIPTION
---------	-----------------	-------------

SETTING	SUGGESTED VALUE	DESCRIPTION
<b>Database ID</b>	Tasks	Enter <i>Tasks</i> 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 <b>Provision database throughput</b> 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.
<b>Throughput</b>	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.
<b>Container ID</b>	Items	Enter <i>Items</i> as the name for your new container. Container IDs have the same character requirements as database names.
<b>Partition key</b>	/category	The sample described in this article uses <i>/category</i> 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**.

The screenshot shows the Azure Cosmos DB Data Explorer interface. On the left, there's a sidebar with various options like Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Quick start, Notifications, and Data Explorer. The Data Explorer option is selected and highlighted with a red box. The main area is titled 'SQL API' and shows a tree structure. Under 'FamilyDatabase', the 'Tasks' node is expanded, and its child 'Items' node is selected and highlighted with a red box. To the right of the tree, there's a query editor with the SQL command 'SELECT \* FROM c' and a 'Edit Filter' button. At the top right, there are buttons for 'New Item' (highlighted with a red box) and 'Upload Item'. Below the tree, there are links for 'Scale & Settings', 'Stored Procedures', 'User Defined Functions', and 'Triggers'. A 'Load more' button is visible at the bottom right.

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**.

The screenshot shows the Azure Cosmos DB Data Explorer interface with the 'Documents' tab selected. The top navigation bar includes 'New Item', 'Save' (highlighted with a red box), 'Discard', and 'Upload Item'. The left sidebar shows the 'SQL API' tree with 'FamilyDatabase' expanded, 'Clothing' selected, and 'Items' highlighted with a blue box. The main area displays the JSON document structure. The 'id' field is highlighted with a red box. The JSON content is as follows:

```
1  {  
2      "id": "1",  
3      "category": "personal",  
4      "name": "groceries",  
5      "description": "Pick up apples and strawberries.",  
6      "isComplete": false  
7  }
```

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.

- At the top of the **Documents** tab in Data Explorer, review the default query `SELECT * FROM c`. This query retrieves and displays all documents in the collection in ID order.

The screenshot shows the Azure Data Explorer interface with the 'Documents' tab selected. At the top, there are buttons for 'New Document', 'Update', 'Discard', and 'Delete'. Below the toolbar, a red box highlights the query input field containing `SELECT * FROM c`. To the right of the field is a blue button labeled 'Edit Filter'. The main area displays a table with two rows. The first row has an 'id' of 1 and a 'category' of 'personal'. The second row has an 'id' of 2 and a 'category' of 'business'. A 'Load more' button is visible at the bottom. On the right side of the table, the document's JSON content is shown in a scrollable pane:

```
1 {
2   "id": "1",
3   "category": "personal",
4   "name": "groceries",
5   "description": "Pick up apples and strawberries.",
6   "isComplete": false,
7   "_rid": "a017ANxuEAABAAAAAA==",
8   "_self": " dbs/a017AA==/colls/a017ANxuEAA=/docs/a017ANxuEAABAAAAAA==/",
9   "_etag": "\"00007e09-0000-0000-0000-59e8aab80000\"",
10  "attachments": "attachments/".
```

- To change the query, select **Edit Filter**, replace the default query with `ORDER BY c._ts DESC`, and then select **Apply Filter**.

The screenshot shows the same Data Explorer interface. The query input field now contains `SELECT * FROM c ORDER BY c._ts DESC`. A red box highlights the 'Apply Filter' button to the right of the query field. The table below shows the results: the document with 'id' 2 and 'category' 'business' is now listed first, followed by the document with 'id' 1 and 'category' 'personal'. The JSON content on the right is identical to the previous screenshot.

The modified query displays the documents in descending order based on their time stamp, so now your second document is listed first.

The screenshot shows the Data Explorer interface with the modified query. The table now lists the documents in descending timestamp order: the document with 'id' 2 and 'category' 'business' is at the top, and the document with 'id' 1 and 'category' 'personal' is below it. The JSON content on the right shows the full document structure for both items.

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.

- 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 following snippets are all taken from the *app.js* file.

- The `CosmosClient` object is initialized.

```
const client = new CosmosClient({ endpoint, key });
```

- Select the "Tasks" database.

```
const database = await client.databases(databaseId);
```

- Select the "Items" container/collection.

```
const container = await client.databases(containerId);
```

- Select all the items in the "Items" container.

```
// query to return all items
const querySpec = {
  query: "SELECT * from c"
};

const { resources: results } = await container.items
  .query(querySpec)
  .fetchAll();

return results;
```

- Create a new item

```
const { resource: createdItem } = await container.items.create(newItem);
```

- Update an item

```
const { id, category } = createdItem;

createdItem.isComplete = true;
const { resource: itemToUpdate } = await container
  .item(id, category)
  .replace(itemToUpdate);

return result;
```

- Delete an item

```
const { resource: result } = await this.container.item(id, category).delete();
```

#### NOTE

In both the "update" and "delete" methods, the item has to be selected from the database by calling `conatiner.item()`. The two parameters passed in are the id of the item and the item's partition key. In this case, the partition 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 required npm modules

2. Run `node app.js` in a terminal to start your node application.

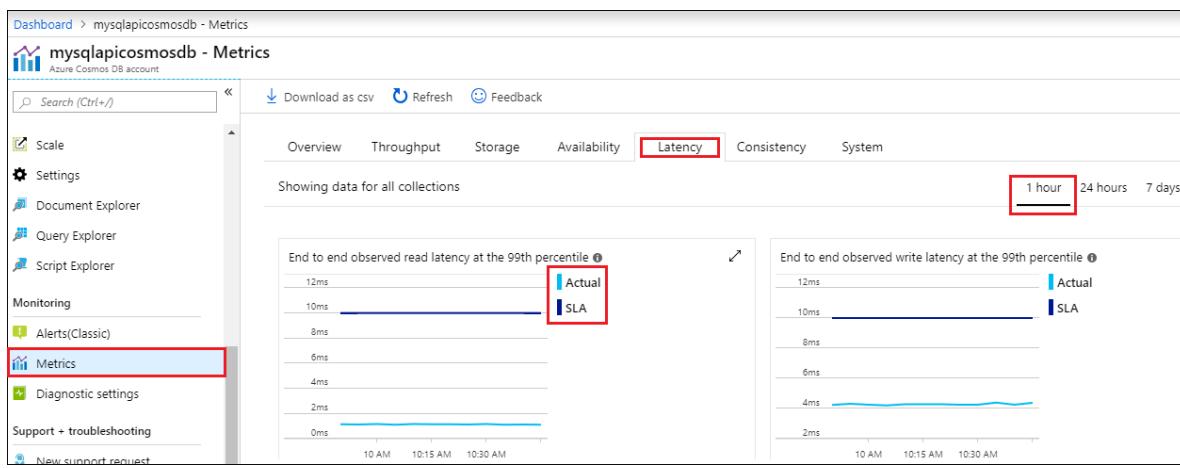
You can now go back to Data Explorer, modify, and work with this new data.

## 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.

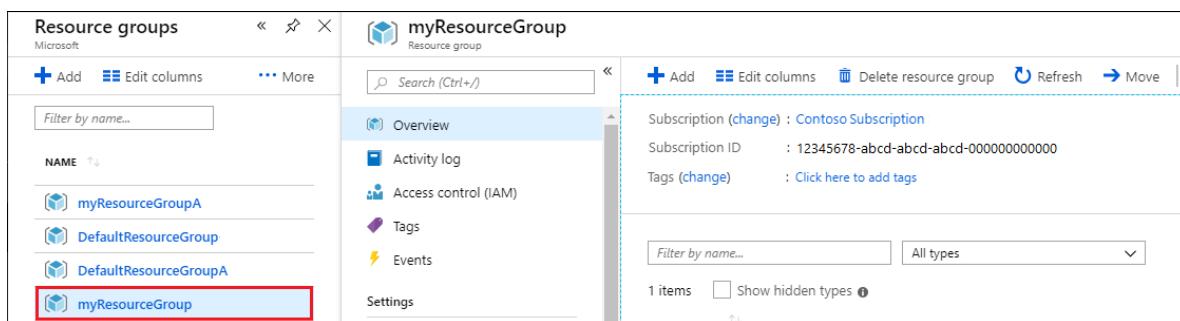


3. Review the metrics on the other tabs.

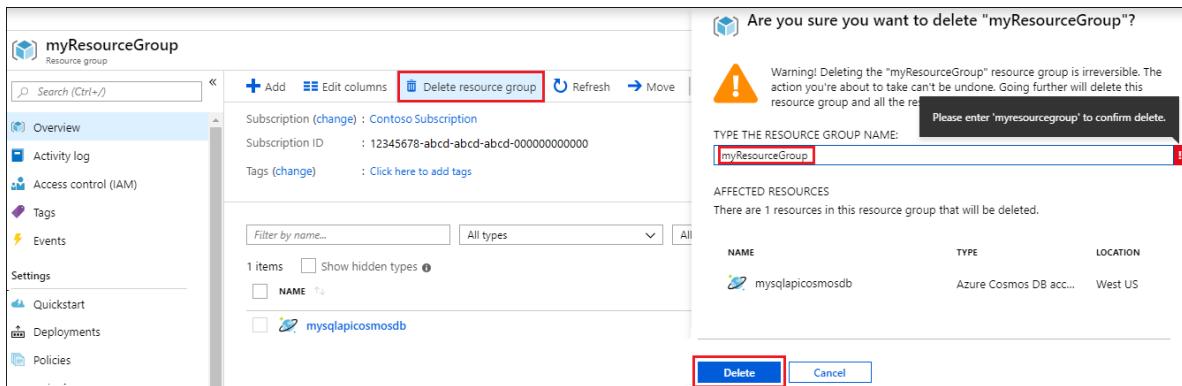
## Clean up resources

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 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

2/24/2020 • 9 minutes to read • [Edit Online](#)

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

- 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+ob0N8A7Cgv30VRDJWEHLM+4QDU5DE2nQ9nDuVTqobD4b8mGGyPMbIZnqyMsEcaGQy67XIw/Jw==`.
- [Python 3.6+](#), with the `python` executable in your `PATH`.
- [Visual Studio Code](#).
- The [Python extension for Visual Studio Code](#).
- [Git](#).

## Create a database account

1. Go to the [Azure portal](#) to create an Azure Cosmos DB account. Search for and select **Azure Cosmos DB**.

The screenshot shows the Azure portal's search interface. The search bar at the top has 'Azure Cosmos DB' typed into it. Below the search bar, there are two main sections: 'Services' and 'Resources'. In the 'Services' section, several items are listed with their respective icons: Azure Cosmos DB (highlighted with a red box), Azure Database for MySQL servers, Azure Databricks, Azure DevOps, Azure Lighthouse, Azure Migrate, Azure Sentinel, Azure SQL, Azure Database for MariaDB servers, and Azure Active Directory. In the 'Resources' section, it says 'No results were found.'

2. Select **Add**.

3. 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 <b>Create new</b> , then enter a unique name for the new resource group.
Account Name	A unique name	<p>Enter a name to identify your Azure Cosmos account. Because <i>documents.azure.com</i> is appended to the name that you provide to create your URI, use a unique name.</p> <p>The name can only contain lowercase letters, numbers, and the hyphen (-) character. It must be between 3-31 characters in length.</p>
API	The type of account to create	<p>Select <b>Core (SQL)</b> to create a document database and query by using SQL syntax.</p> <p>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.</p> <p><a href="#">Learn more about the SQL API.</a></p>
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.

Dashboard > New > Create Azure Cosmos DB Account

## Create Azure Cosmos DB Account

**Basics** **Network** **Tags** **Review + create**

Azure Cosmos DB is a fully managed globally distributed, multi-model database service, transparently replicating your data across any number of Azure regions. You can elastically scale throughput and storage, and take advantage of fast, single-digit-millisecond data access using the API of your choice backed by 99.999 SLA. [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: Contoso Subscription

\* Resource Group: (New) myResourceGroup [Create new](#)

**INSTANCE DETAILS**

\* Account Name: mysqlapicosmosdb [documents.azure.com](#)

\* API: Core (SQL)

\* Location: West US

Geo-Redundancy: [Enable](#) [Disable](#)

Multi-region Writes: [Enable](#) [Disable](#)

**Review + create** [Previous](#) [Next: Network](#)

4. Select **Review + create**. You can skip the **Network** and **Tags** sections.
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**.

Dashboard > Microsoft.Azure.CosmosDB-2019032100000 - Overview

### Microsoft.Azure.CosmosDB-2019032100000 - Overview

Deployment

Search (Ctrl+/  
)

[Delete](#) [Cancel](#) [Redeploy](#) [Refresh](#)

[Overview](#) [Inputs](#) [Outputs](#) [Template](#)

**✓ Your deployment is complete**

[Go to resource](#)

Deployment name: Microsoft.Azure.CosmosDB-2019032100000  
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
<a href="#">mysqlapicosmosdb</a>	Microsoft.DocumentDb/databaseAcc...	OK	<a href="#">Operation details</a>

6. Select **Go to resource** to go to the Azure Cosmos DB account page.

Congratulations! Your Azure Cosmos DB account was created.

Now, let's connect to it using a sample app:

**Choose a platform**

- .NET
- .NET Core
- Xamarin
- Java
- Node.js
- Python

- 1 Add a collection**

In Azure Cosmos DB, data is stored in collections.

**Create 'Items' collection**

Create 'Items' collection with 10GB storage capacity and 400 Request Units per second (RU/s) throughput capacity, for free!

- 2 Download and run your .NET app**

Once collection is created, download a sample .NET app connected to it, extract, build and run.

**Download**

## 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.

**New Container**

**Add Container**

Start at <price>/mo per database, multiple containers incl... [More details](#)

\* Database id  Create new  Use existing  
Tasks

Provision database throughput  Throughput (400 - 100,000 RU/s)  Autopilot (preview)  Manual  
400

Estimated spend (USD): <price> (2 regions, 400RU/s, \$<price>/RU)

\* Container id  Items

\* Partition key  /category

My partition key is larger than 100 bytes

Unique keys  + Add unique key

### 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 <i>ToDoList</i> 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 <b>Provision database throughput</b> 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 <i>Items</i> 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 <i>/category</i> 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.

- From the **Data Explorer**, expand the **Tasks** database, expand the **Items** container. Select **Items**, and then select **New Item**.

The screenshot shows the Azure Cosmos DB Data Explorer interface. On the left, there's a navigation sidebar with various options like Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Quick start, Notifications, and Data Explorer (which is highlighted with a red box). Below these are Settings, Replicate data globally, and Default consistency. The main area is titled 'SQL API' and shows a tree view under 'FamilyDatabase' with 'Tasks' expanded, showing 'Items' (also highlighted with a red box). To the right, there's a query editor with the following code:

```
SELECT * FROM c
```

Below the query, there's a table with columns 'id' and '/cate...', and a 'Load more' button.

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**.

The screenshot shows the Azure Cosmos DB Data Explorer interface. The top bar has buttons for New Item, Save (highlighted with a red box), Discard, and Upload Item. The main area is titled 'SQL API' and shows a tree view under 'FamilyDatabase' with 'Tasks' expanded, showing 'Items' (highlighted with a blue box). To the right, there's a query editor with the same JSON document as above, and a preview pane showing the document structure with line numbers 1 through 7.

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 **Documents** tab in Data Explorer, review the default query `SELECT * FROM c`. This query retrieves and displays all documents in the collection in ID order.

The screenshot shows the Azure Data Explorer interface with the 'Documents' tab selected. At the top, there are buttons for 'New Document', 'Update', 'Discard', and 'Delete'. Below these, a search bar contains the query `SELECT * FROM c`, which is highlighted with a red box. To the right of the search bar is a blue button labeled 'Edit Filter'. The main area displays a table with two rows of data. The columns are labeled 'id' and '/category'. Row 1 has 'id' value 1 and '/category' value 'personal'. Row 2 has 'id' value 2 and '/category' value 'business'. A 'Load more' button is visible at the bottom of the table. On the right side of the table, the document's JSON content is shown in a scrollable pane, starting with:

```
1 {
  "id": "1",
  "category": "personal",
  "name": "groceries",
  "description": "Pick up apples and strawberries.",
  "isComplete": false,
  "_rid": "a017ANxuEAABAAAAAA==",
  "_self": " dbs/a017AA==/colls/a017ANxuEAA=/docs/a017ANxuEAABAAAAAA==/",
  "_etag": "\\"00007e09-0000-0000-0000-59e8aab80000\\",
  "attachments": "attachments/".
```

2. To change the query, select **Edit Filter**, replace the default query with `ORDER BY c._ts DESC`, and then select **Apply Filter**.

The screenshot shows the same Azure Data Explorer interface as before, but with a modified query in the search bar. The original query `SELECT * FROM c` is still present, but the addition `ORDER BY c._ts DESC` is highlighted with a red box. To the right of the search bar is a blue button labeled 'Apply Filter', which is also highlighted with a red box.

The modified query displays the documents in descending order based on their time stamp, so now your second document is listed first.

The screenshot shows the results of the modified query. The table now lists the documents in descending order of time stamp. Row 2 (business) is now at the top, and Row 1 (personal) is below it. The JSON content on the right side of the table shows the updated order:

```
1 {
  "id": "2",
  "category": "business",
  "name": "meetings",
  "description": "Meet with Britta.",
  "isComplete": false,
  "_rid": "mk00ALLgQtYCAAAAAAAA==",
  "_self": " dbs/mk00AA==/colls/mk00ALLgQtY=/docs/mk00ALLgQtYCAAAAAAAA==",
  "_etag": "\\"1000c09c-0000-0700-0000-5c9433560000\\",
  "attachments": "attachments/",
  "_ts": 1553216342
```

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"
```

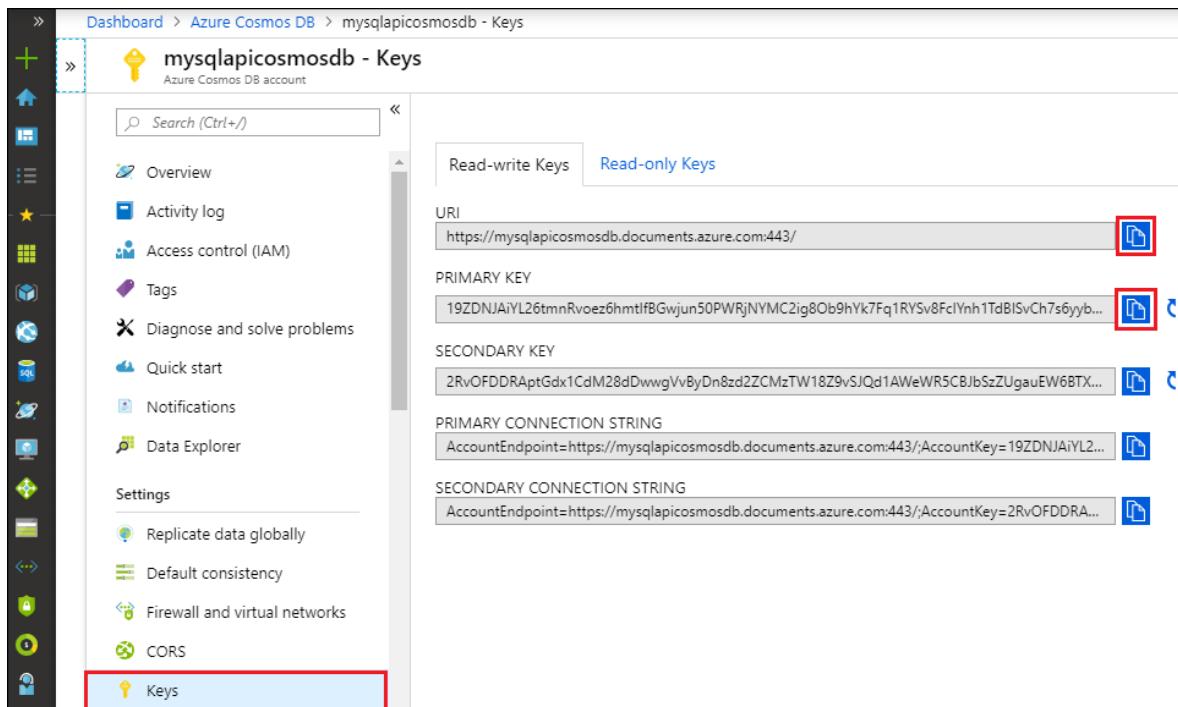
3. Run the following command to clone the sample repository. This command creates a copy of the sample app on your computer.

```
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'
```

- 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:

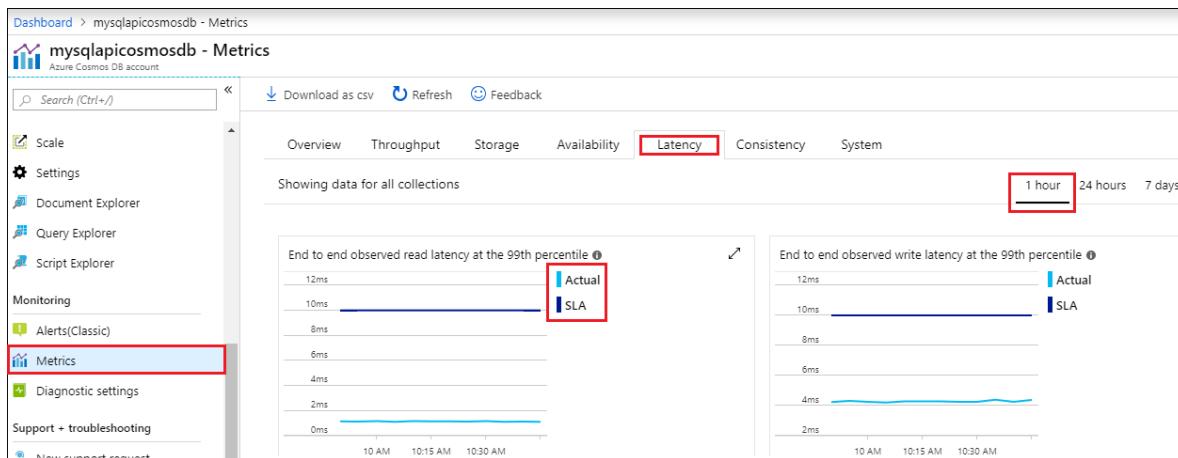
```
{
  "id": "Andersen-1569479288379",
  "lastName": "Andersen",
  "district": "WA5",
  "parents": [
    {
      "familyName": null,
      "firstName": "Thomas"
    },
    {
      "familyName": null,
      "firstName": "Mary Kay"
    }
  ],
  "children": null,
  "address": {
    "state": "WA",
    "county": "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.

The screenshot shows the Azure portal's 'Resource groups' blade. On the left, a list of resource groups is shown, including 'myResourceGroupA', 'DefaultResourceGroup', and 'myResourceGroup'. The 'myResourceGroup' item is highlighted with a red box. On the right, the 'Overview' page for 'myResourceGroup' is displayed. It shows the subscription information ('Contoso Subscription'), resource group ID ('12345678-abcd-abcd-000000000000'), and a link to add tags. Below this, there are sections for 'Activity log', 'Access control (IAM)', 'Tags', 'Events', and 'Settings'. A 'Filter by name...' input field and a 'All types' dropdown are also present.

3. On the resource group **Overview** page, select **Delete resource group**.

The screenshot shows a confirmation dialog titled 'Are you sure you want to delete "myResourceGroup"?'. It includes a warning message about the irreversibility of the action. A text input field is pre-filled with 'myResourceGroup'. Below the input field, a table lists the affected resources: 'mysqlapicosmosdb' (Azure Cosmos DB account) located in West US. At the bottom of the dialog are 'Delete' and 'Cancel' buttons, with the 'Delete' button also highlighted with a red box.

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

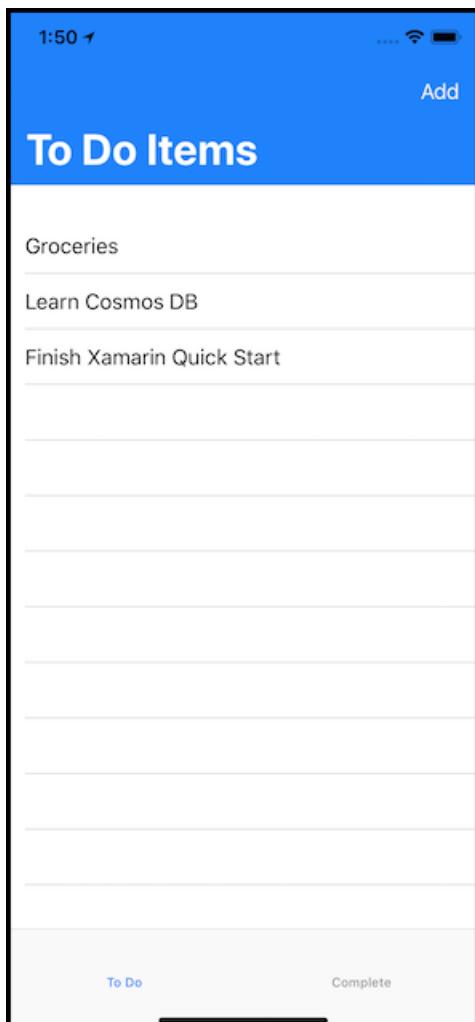
2/24/2020 • 12 minutes to read • [Edit Online](#)

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 web 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.

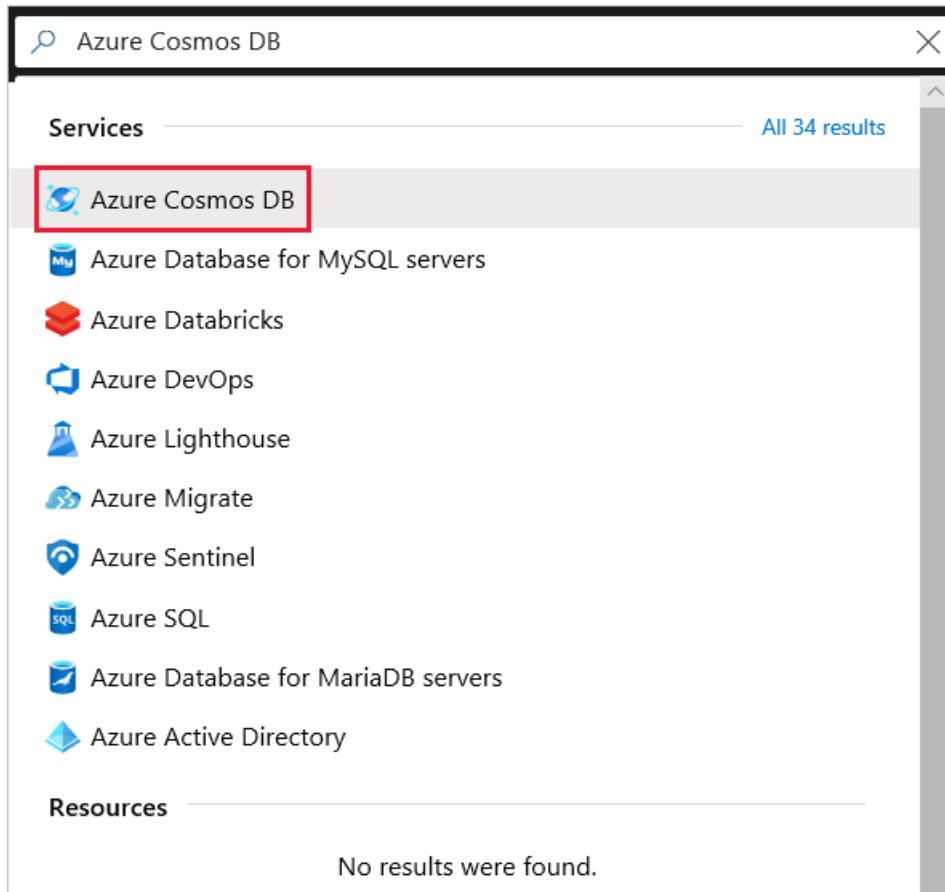
If you are using a Mac, you can download the [free Visual Studio for Mac](#).

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, you can 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. Go to the [Azure portal](#) to create an Azure Cosmos DB account. Search for and select **Azure Cosmos DB**.



2. Select **Add**.

3. 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 <b>Create new</b> , then enter a unique name for the new resource group.

Setting	Value	Description
Account Name	A unique name	<p>Enter a name to identify your Azure Cosmos account. Because <i>documents.azure.com</i> is appended to the name that you provide to create your URI, use a unique name.</p> <p>The name can only contain lowercase letters, numbers, and the hyphen (-) character. It must be between 3-31 characters in length.</p>
API	The type of account to create	<p>Select <b>Core (SQL)</b> to create a document database and query by using SQL syntax.</p> <p>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.</p> <p><a href="#">Learn more about the SQL API.</a></p>
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.

## Create Azure Cosmos DB Account

[Basics](#) [Network](#) [Tags](#) [Review + create](#)

Azure Cosmos DB is a fully managed globally distributed, multi-model database service, transparently replicating your data across any number of Azure regions. You can elastically scale throughput and storage, and take advantage of fast, single-digit-millisecond data access using the API of your choice backed by 99.999 SLA. [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.

<p>* Subscription</p> <p>Resource Group</p>	<div style="border: 1px solid #ccc; padding: 2px;">Contoso Subscription</div> <div style="border: 1px solid #ccc; padding: 2px;">(New) myResourceGroup</div> <a href="#">Create new</a>
---	---

### INSTANCE DETAILS

<p>* Account Name</p> <p>* API</p> <p>* Location</p> <p>Geo-Redundancy</p> <p>Multi-region Writes</p>	<div style="border: 1px solid #ccc; padding: 2px;">mysqlapicosmosdb</div> <div style="border: 1px solid #ccc; padding: 2px;">Core (SQL)</div> <div style="border: 1px solid #ccc; padding: 2px;">West US</div> <div style="border: 1px solid #ccc; padding: 2px; display: flex; justify-content: space-around;"><span>Enable</span> <span>Disable</span></div> <div style="border: 1px solid #ccc; padding: 2px; display: flex; justify-content: space-around;"><span>Enable</span> <span>Disable</span></div>
---	--

[Review + create](#)

[Previous](#)

[Next: Network](#)

4. Select **Review + create**. You can skip the **Network** and **Tags** sections.

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**.

Dashboard > Microsoft.Azure.CosmosDB-2019032100000 - Overview

**Microsoft.Azure.CosmosDB-2019032100000 - Overview**

Deployment

Search (Ctrl+ /) < Delete Cancel Redeploy Refresh

**Overview** **Inputs** **Outputs** **Template**

**✓ Your deployment is complete**

Go to resource

Deployment name: Microsoft.Azure.CosmosDB-2019032100000  
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	<a href="#">Operation details</a>

6. Select **Go to resource** to go to the Azure Cosmos DB account page.

Congratulations! Your Azure Cosmos DB account was created.

Now, let's connect to it using a sample app:

**Choose a platform**

- .NET
- .NET Core
- Xamarin
- Java
- Node.js
- Python

- 1 Add a collection**

In Azure Cosmos DB, data is stored in collections.

**Create 'Items' collection**

Create 'Items' collection with 10GB storage capacity and 400 Request Units per second (RU/s) throughput capacity, for free!

- 2 Download and run your .NET app**

Once collection is created, download a sample .NET app connected to it, extract, build and run.

**Download**

## 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.

**New Container**

**Add Container**

**Start at \$<price>/mo per database, multiple containers included**

**Database id**: Items

**Throughput**: 400 RU/s

**Container id**: Items

**Partition key**: /category

**Description**: Unique keys

**Tasks**: Provision database throughput

**Autopilot (preview)**: Manual

**Estimated spend (USD)**: \$<price>hourly / \$<price>daily / \$<price>monthly (2 regions, 400RU/s, \$<price>/RU)

**My partition key is larger than 100 bytes**

### 2. In the **Add container** page, enter the settings for the new container.

SETTING	SUGGESTED VALUE	DESCRIPTION
---------	-----------------	-------------

SETTING	SUGGESTED VALUE	DESCRIPTION
<b>Database ID</b>	Tasks	Enter <i>Tasks</i> 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 <b>Provision database throughput</b> 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.
<b>Throughput</b>	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.
<b>Container ID</b>	Items	Enter <i>Items</i> as the name for your new container. Container IDs have the same character requirements as database names.
<b>Partition key</b>	/category	The sample described in this article uses <i>/category</i> 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**.

The screenshot shows the Azure Cosmos DB Data Explorer interface. On the left, there's a sidebar with various options like Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Quick start, Notifications, and Data Explorer. The Data Explorer option is selected and highlighted with a red box. The main area is titled 'SQL API' and shows a tree structure. Under 'FamilyDatabase', the 'Tasks' node is expanded, and its 'Items' child node is selected and highlighted with a red box. To the right of the tree, there's a query editor with the SQL command 'SELECT \* FROM c' and a 'Edit Filter' button. At the top right, there are buttons for 'New Item' (highlighted with a red box) and 'Upload Item'. Below the tree, there are links for 'Scale & Settings', 'Stored Procedures', 'User Defined Functions', and 'Triggers'. A 'Load more' button is visible at the bottom right.

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**.

The screenshot shows the Azure Cosmos DB Data Explorer interface with the 'Documents' tab selected. The top navigation bar includes 'New Item' and 'Save' (highlighted with a red box). The main area displays the JSON document structure from step 2. The 'id' field is highlighted with a red box. The 'category', 'name', 'description', and 'isComplete' fields are also visible. The 'category' field contains 'personal', 'name' contains 'groceries', 'description' contains 'Pick up apples and strawberries.', and 'isComplete' is set to false. There are also line numbers 1 through 7 on the left side of the JSON code.

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.

- At the top of the **Documents** tab in Data Explorer, review the default query `SELECT * FROM c`. This query retrieves and displays all documents in the collection in ID order.

The screenshot shows the Azure Data Explorer interface with the 'Documents' tab selected. At the top, there are buttons for 'New Document', 'Update', 'Discard', and 'Delete'. Below the toolbar, a red box highlights the query input field containing `SELECT * FROM c`. To the right of the input field is a blue button labeled 'Edit Filter'. The main area displays a table with two rows. The first row has an 'id' of 1 and a 'category' of 'personal'. The second row has an 'id' of 2 and a 'category' of 'business'. A 'Load more' button is visible at the bottom. On the right side of the table, the document details are shown in JSON format, starting with `1 {` and ending with `"attachments": "attachments/",`.

- To change the query, select **Edit Filter**, replace the default query with `ORDER BY c._ts DESC`, and then select **Apply Filter**.

The screenshot shows the same Azure Data Explorer interface as the previous one, but with a modified query. The query input field now contains `SELECT * FROM c ORDER BY c._ts DESC`. A red box highlights the `ORDER BY c._ts DESC` part. To the right of the input field is a blue button labeled 'Apply Filter'.

The modified query displays the documents in descending order based on their time stamp, so now your second document is listed first.

The screenshot shows the results of the modified query. The table now lists the documents in descending order of time stamp. The second document, which had an 'id' of 2 and a 'category' of 'business', is now the first item in the list. The first document, with an 'id' of 1 and a 'category' of 'personal', is now the second item. The JSON details on the right show the updated order, with the first document starting with `1 {` and the second with `2 {`.

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.

- 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.

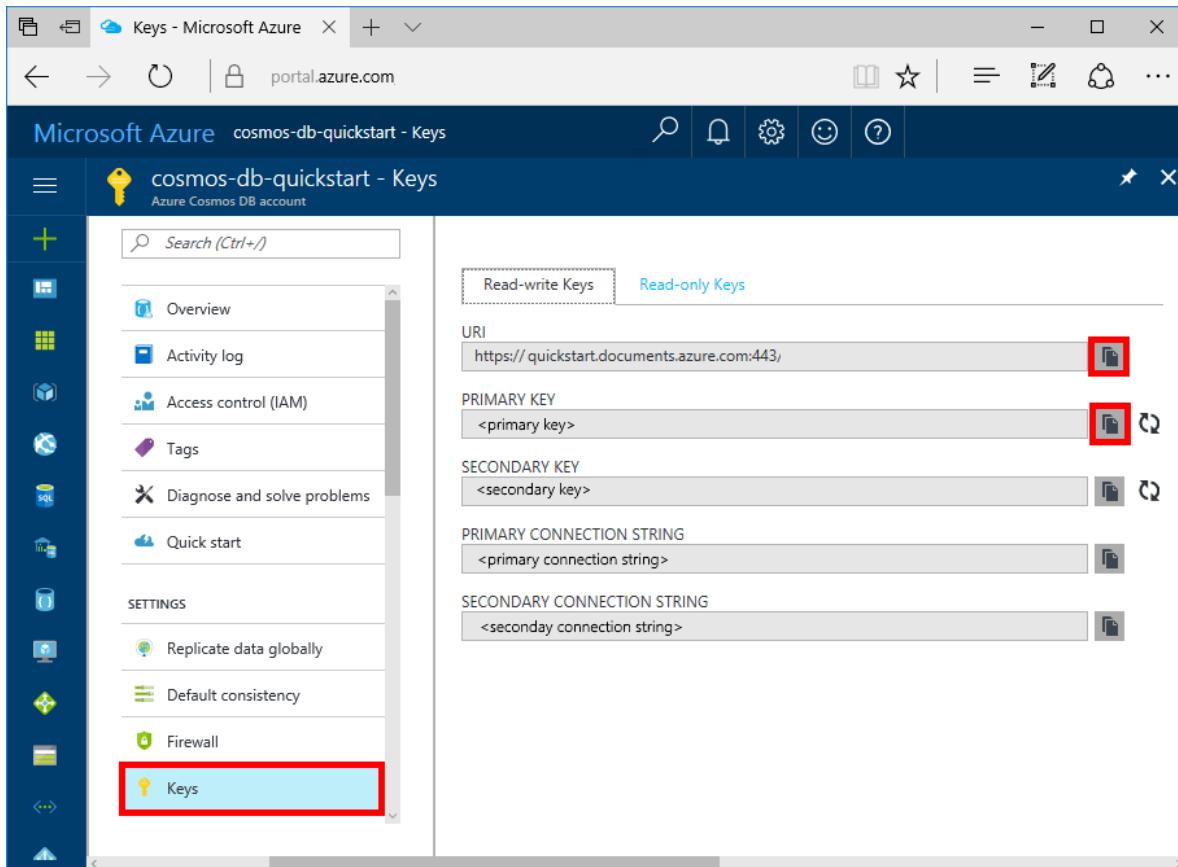
```
git clone https://github.com/Azure-Samples/azure-cosmos-db-sql-xamarin-getting-started.git
```

4. Then open the `ToDoltems.sln` file from the `samples/xamarin/ToDoItems` folder in Visual Studio.

## 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 either Visual Studio 2019 or Visual Studio for Mac, open the `APIKeys.cs` file in the `azure-documentdb-dotnet/samples/xamarin/ToDoItems/ToDoItems.Core/Helpers` folder.
3. Copy your URI value from the portal (using the copy button) and make it the value of the `CosmosEndpointUrl` variable in `APIKeys.cs`.

```
public static readonly string CosmosEndpointUrl = "{Azure Cosmos DB account URL}";
```

4. Then copy your PRIMARY KEY value from the portal and make it the value of the `Cosmos Auth Key` in `APIKeys.cs`.

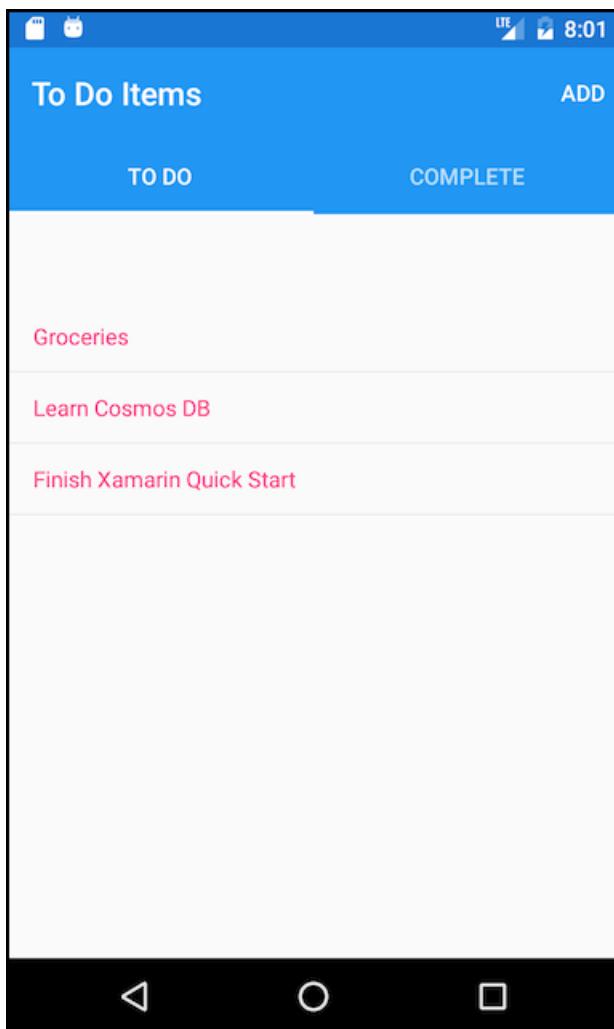
```
public static readonly string CosmosAuthKey = "{Azure Cosmos DB secret}";
```

#### 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 `azure-documentdb-dotnet/samples/xamarin/ToDoItems/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 `azure-documentdb-dotnet/samples/xamarin/ToDoItems/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 `CosmosDBService.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 URL 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

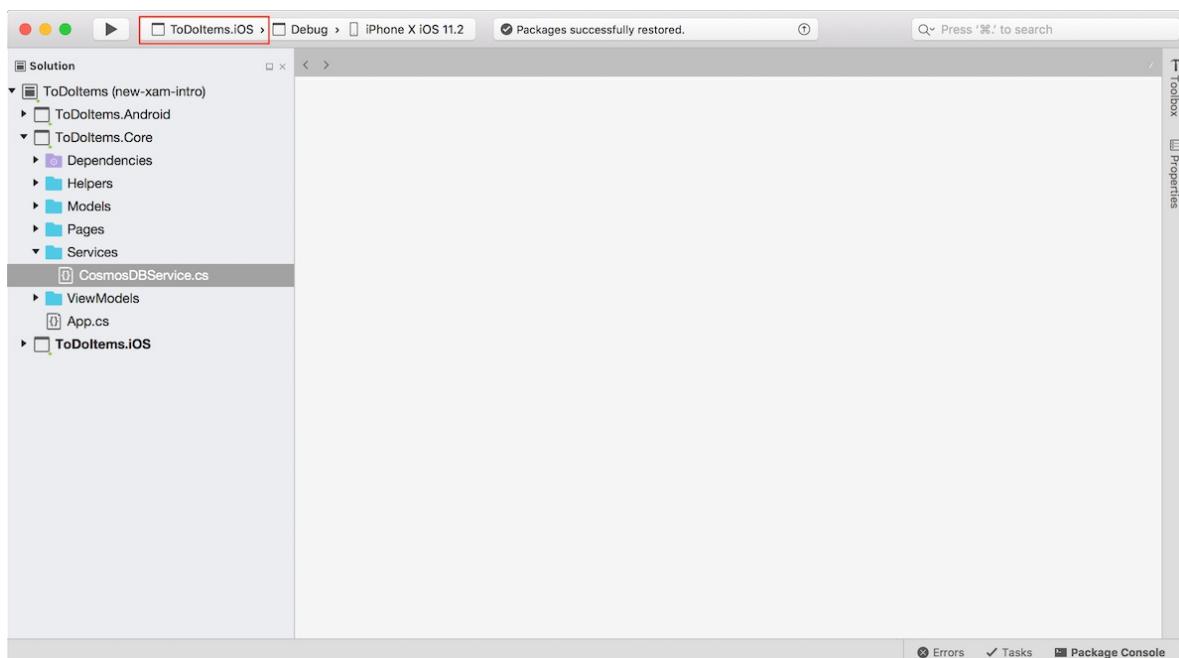
You've now updated your app with all the info it needs to communicate with Azure Cosmos DB.

The following steps will demonstrate how to run the app using the Visual Studio for Mac debugger.

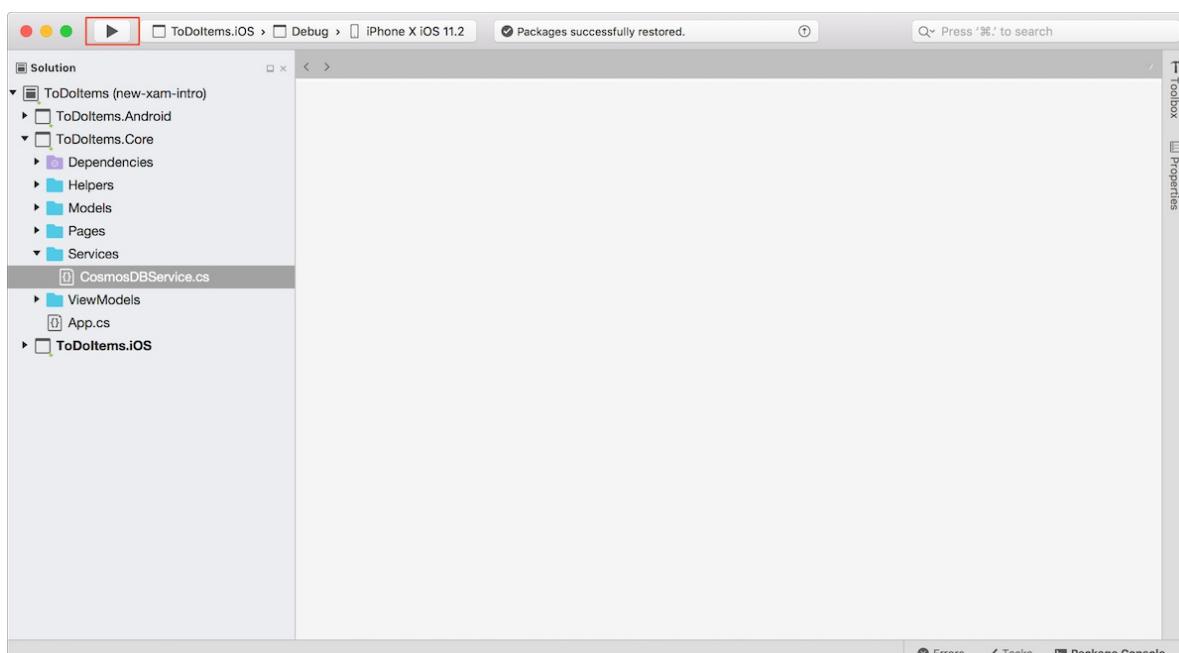
### NOTE

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](#).

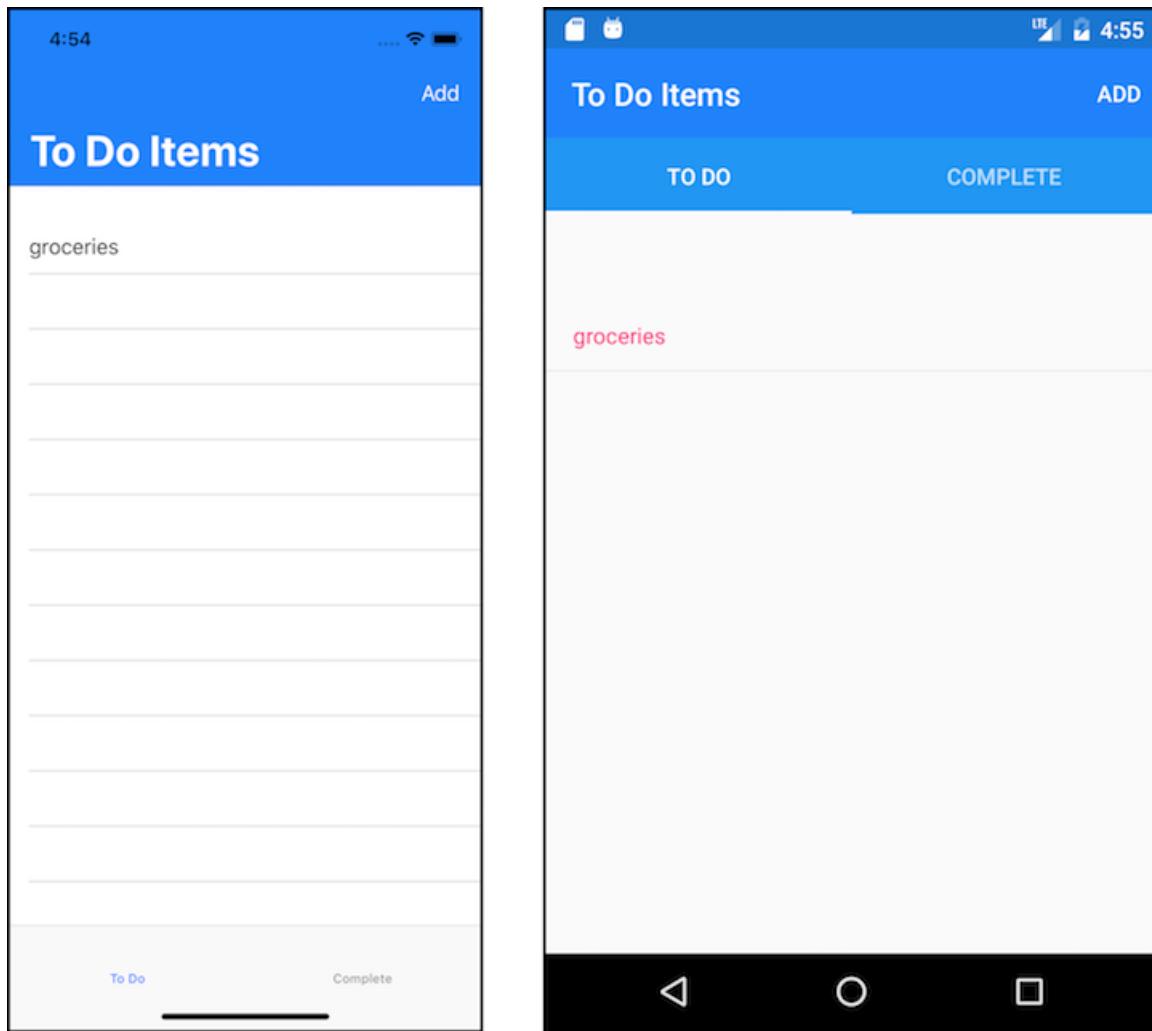
1. First select the platform you wish to target by clicking on the dropdown highlighted and selecting either `ToDoltems.iOS` for iOS or `ToDoltems.Android` for Android.



2. To start debugging the app, either press cmd+Enter or click the play button.



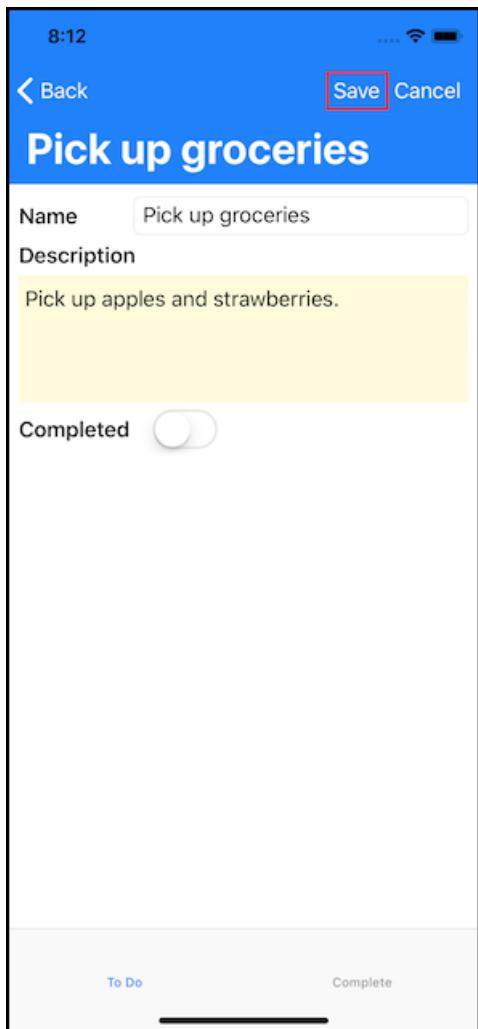
3. When the iOS simulator or Android emulator finishes launching, the app will display 2 tabs at the bottom of the screen for iOS and the top of the screen for Android. The first shows todo items which are not completed, the second shows todo items which are completed.



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.

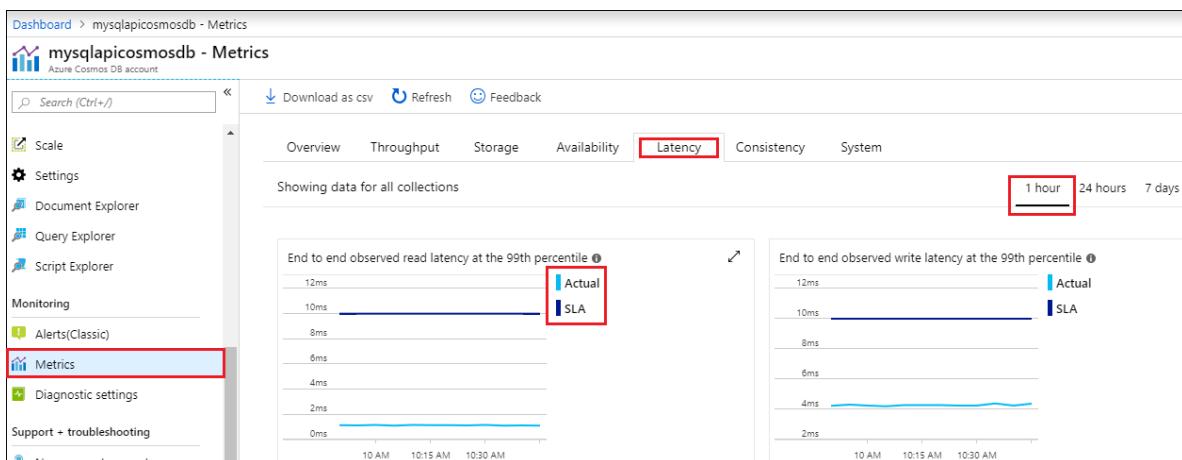


## Review SLAs in the Azure portal

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.

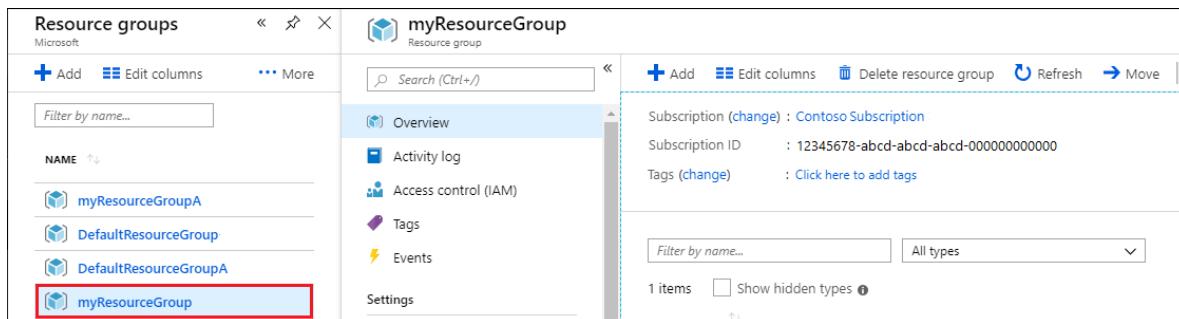


3. Review the metrics on the other tabs.

# Clean up resources

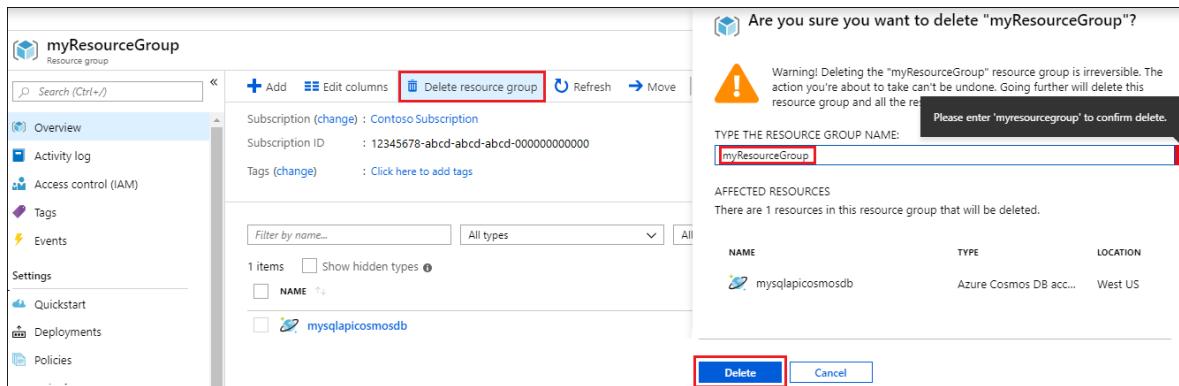
When you're done with your app and Azure Cosmos DB account, you can delete the Azure resources you created so you don't incur more charges. To delete the resources:

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.



The screenshot shows the Azure portal's Resource groups blade. On the left, there's a list of resource groups: 'myResourceGroupA', 'DefaultResourceGroup', 'DefaultResourceGroupA', and 'myResourceGroup'. The 'myResourceGroup' item is highlighted with a red box. On the right, the details for 'myResourceGroup' are shown, including its subscription information ('Contoso Subscription'), subscription ID ('12345678-abcd-abcd-000000000000'), and tags ('Click here to add tags'). There are also links for Overview, Activity log, Access control (IAM), Tags, Events, and Settings.

3. On the resource group **Overview** page, select **Delete resource group**.



The screenshot shows a confirmation dialog for deleting the 'myResourceGroup' resource group. The title is 'Are you sure you want to delete "myResourceGroup"?'. It contains a warning message: 'Warning! Deleting the "myResourceGroup" resource group is irreversible. The action you're about to take can't be undone. Going further will delete this resource group and all the resources it contains.' Below the warning, it says 'Please enter "myResourceGroup" to confirm delete' and has a text input field containing 'myResourceGroup'. On the right, it lists 'AFFECTED RESOURCES' with one item: 'mysqlapicosmosdb' (Azure Cosmos DB account, West US). At the bottom are 'Delete' and 'Cancel' buttons, with 'Delete' being highlighted with a red box.

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

2/24/2020 • 23 minutes to read • [Edit Online](#)

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, you can 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. Go to the [Azure portal](#) to create an Azure Cosmos DB account. Search for and select **Azure Cosmos DB**.

Azure Cosmos DB

Services

- Azure Cosmos DB**
- Azure Database for MySQL servers
- Azure Databricks
- Azure DevOps
- Azure Lighthouse
- Azure Migrate
- Azure Sentinel
- Azure SQL
- Azure Database for MariaDB servers
- Azure Active Directory

Resources

No results were found.

2. Select **Add**.
3. 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 <b>Create new</b> , then enter a unique name for the new resource group.
Account Name	A unique name	<p>Enter a name to identify your Azure Cosmos account. Because <i>documents.azure.com</i> is appended to the name that you provide to create your URI, use a unique name.</p> <p>The name can only contain lowercase letters, numbers, and the hyphen (-) character. It must be between 3-31 characters in length.</p>

SETTING	VALUE	DESCRIPTION
API	The type of account to create	<p>Select <b>Core (SQL)</b> to create a document database and query by using SQL syntax.</p> <p>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.</p> <p><a href="#">Learn more about the SQL API.</a></p>
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.

Dashboard > New > Create Azure Cosmos DB Account

## Create Azure Cosmos DB Account

[Basics](#) [Network](#) [Tags](#) [Review + create](#)

Azure Cosmos DB is a fully managed globally distributed, multi-model database service, transparently replicating your data across any number of Azure regions. You can elastically scale throughput and storage, and take advantage of fast, single-digit-millisecond data access using the API of your choice backed by 99.999 SLA. [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	Contoso Subscription
└─ * Resource Group	(New) myResourceGroup
	<a href="#">Create new</a>

**INSTANCE DETAILS**

* Account Name	mysqlapicosmosdb
	documents.azure.com
* API ⓘ	Core (SQL)
* Location	West US
Geo-Redundancy ⓘ	<a href="#">Enable</a> <a href="#">Disable</a>
Multi-region Writes ⓘ	<a href="#">Enable</a> <a href="#">Disable</a>

[Review + create](#) [Previous](#) [Next: Network](#)

4. Select **Review + create**. You can skip the **Network** and **Tags** sections.
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**.

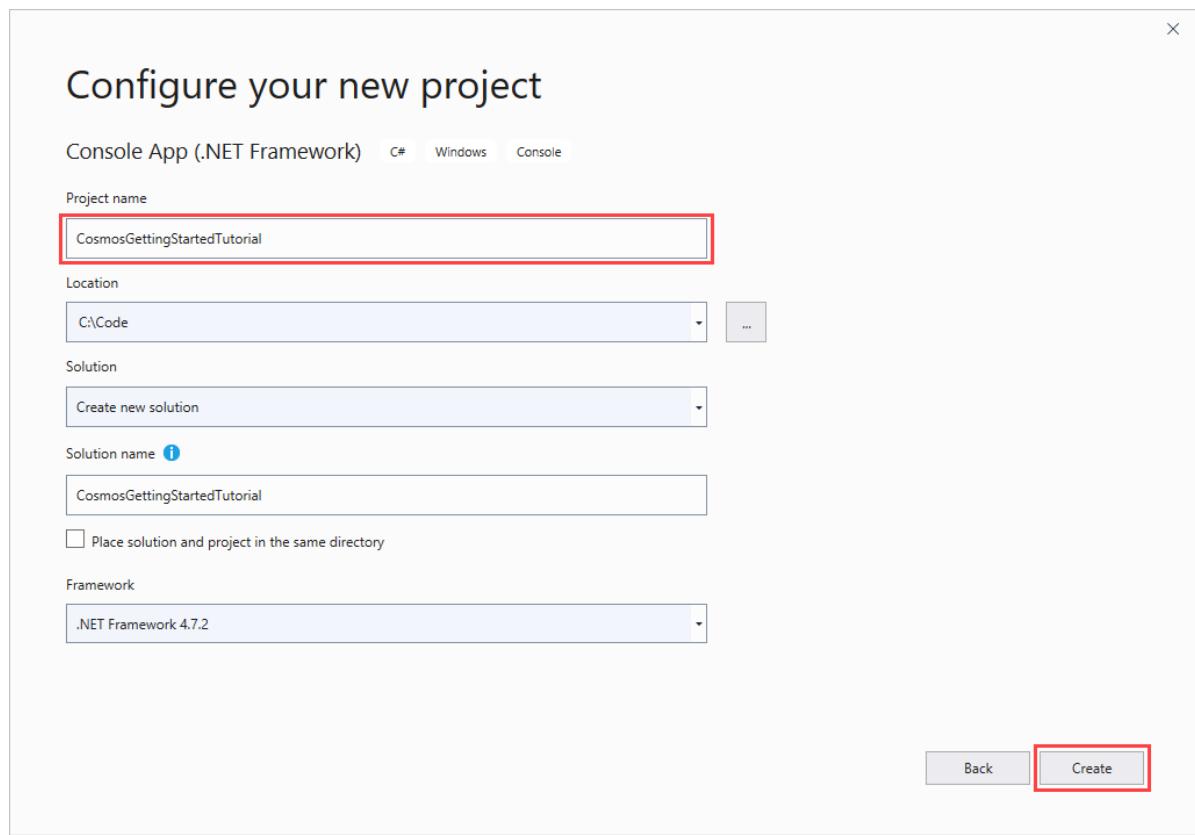
The screenshot shows the Azure Deployment Center interface. At the top, it says "Microsoft.Azure.CosmosDB-20190321000000 · Overview". On the left, there's a navigation menu with "Overview", "Inputs", "Outputs", and "Template". In the center, a message says "Your deployment is complete" with a "Go to resource" button. Below that, deployment details are listed: name (Microsoft.Azure.CosmosDB-20190321000000), subscription (Contoso Subscription), and resource group (myResourceGroup). Deployment details show start time (3/21/2019, 5:00:03 PM), duration (5 minutes 38 seconds), and correlation ID (8e0be948-0c60-4da0-0000-000000000000). A table at the bottom shows one resource: "mysqlapicosmosdb" (Type: Microsoft.DocumentDb/databaseAcc... Status: OK), with a "Operation details" link.

6. Select **Go to resource** to go to the Azure Cosmos DB account page.

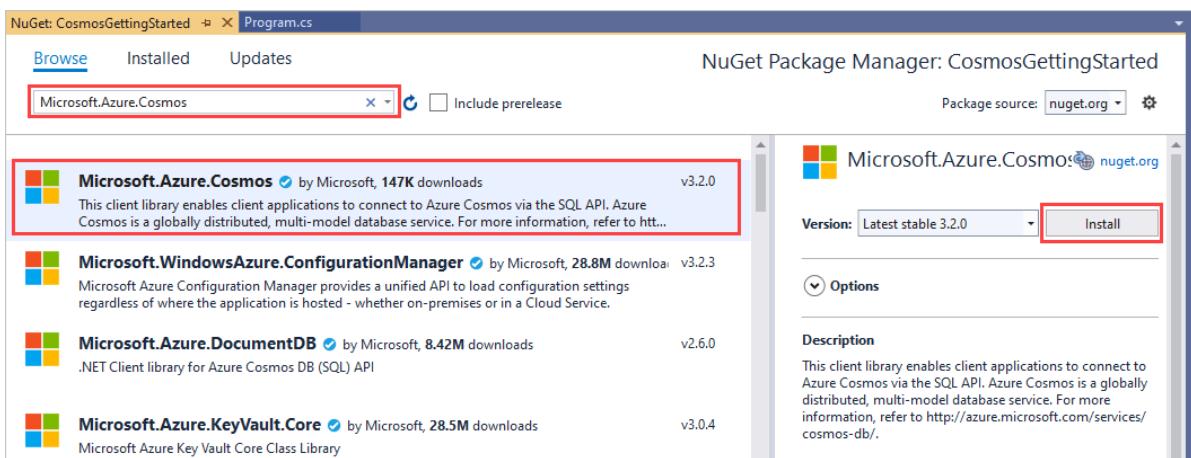
The screenshot shows the "mysqlapicosmosdb - Quick start" page for an Azure Cosmos DB account. The left sidebar has links for "Overview", "Activity log", "Access control (IAM)", "Tags", "Diagnose and solve problems", "Quick start" (which is selected), "Notifications", "Data Explorer", "Settings", "Replicate data globally", "Default consistency", and "Firewall and virtual networks". The main content area says "Congratulations! Your Azure Cosmos DB account was created." and "Now, let's connect to it using a sample app:". It features a "Choose a platform" section with ".NET", ".NET Core", "Xamarin", "Java", "Node.js", and "Python" options. Step 1, "Add a collection", is described as adding an "Items" collection with 10GB storage capacity and 400 Request Units per second (RU/s) throughput. Step 2, "Download and run your .NET app", involves downloading a sample .NET app. A "Create 'Items' collection" button is highlighted.

## 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**.

The screenshot shows the Azure portal interface for managing a Cosmos DB account named 'contoso-demo'. The 'Keys' section is highlighted with a red box. It displays four fields: 'URI' (https://contoso-demo.documents.azure.com:443/), 'PRIMARY KEY' (MECyqfOL6eGkv73YUuZoQY8ej4ZpQJ2frW30YgP2GNLTYaaNoHooRYbOPNdRLQjOwNk5oxe3sPEFn0PQ2T0V7==), 'SECONDARY KEY' (A1qltpGX6GyekSpIjcYfNV0RAO91eCaMNLi1Ge60b6GtYszQYBU5pWZvXzuWVJB1X4T0e0sfXYNuWe95py==), and 'SECONDARY CONNECTION STRING' (AccountEndpoint=https://contoso-demo.documents.azure.com:443/AccountKey=WFa5AYX1OLxpDrWZJ0OrjOeJmWLhN4pVuXS5QjSQX7...). Each field has a copy icon to its right.

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.

```
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 = ConfigurationManager.AppSettings["EndPointUri"];

        // The primary key for the Azure Cosmos account.
        private static readonly string PrimaryKey = ConfigurationManager.AppSettings["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";

        // <Main>
        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();
    }
}

// </Main>

// <GetStartedDemoAsync>
/// <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();
}
// </GetStartedDemoAsync>

// <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);
}
// </CreateDatabaseAsync>

// <CreateContainerAsync>
/// <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);
}
// </CreateContainerAsync>

// <AddItemsToContainerAsync>
/// <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 = "Anna" },
            new Child { FirstName = "Hans" }
        }
    };
}
// </AddItemsToContainerAsync>

```

```

    {
        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);
    }
}
// </AddItemsToContainerAsync>

// <QueryItemsAsync>
/// <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);
        }
    }
}
// </QueryItemsAsync>

// <ReplaceFamilyItemAsync>
/// <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);
}

// </ReplaceFamilyItemAsync>

// <DeleteFamilyItemAsync>
/// <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);
}
// </DeleteFamilyItemAsync>

// <DeleteDatabaseAndCleanupAsync>
/// <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();
}
// </DeleteDatabaseAndCleanupAsync>
}
}

```

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.

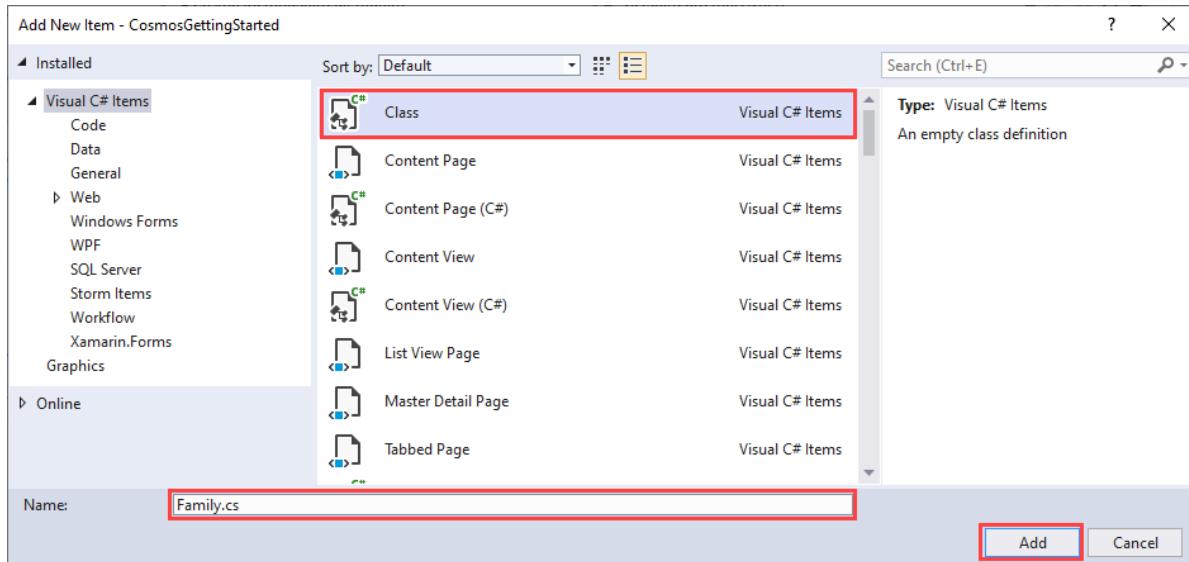
Congratulations! You've successfully created an Azure Cosmos container.

## 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; }
    }
}

```

- 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*.

- 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();
}

```

- 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.

- 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](#).

# NoSQL tutorial: Build a SQL API Java console application

11/6/2019 • 5 minutes to read • [Edit Online](#)

Welcome to the NoSQL tutorial for the SQL API for Azure Cosmos DB Java SDK! After following this tutorial, you'll have a console application that creates and queries Azure Cosmos DB resources.

We cover:

- Creating and connecting to an Azure Cosmos DB account
- Configuring your Visual Studio Solution
- Creating an online database
- Creating a collection
- Creating JSON documents
- Querying the collection
- Creating JSON documents
- Querying the collection
- Replacing a document
- Deleting a document
- Deleting the database

Now let's get started!

## Prerequisites

Make sure you have the following:

- 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, you can use the [Azure Cosmos DB Emulator](#) with a URI of `https://localhost:8081`. For the key to use with the emulator, see [Authenticating requests](#).

- [Git](#).
- [Java Development Kit \(JDK\) 7+](#).
- [Maven](#).

## 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, you can skip ahead to [Clone the GitHub project](#). 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 [Clone the GitHub project](#).

1. Go to the [Azure portal](#) to create an Azure Cosmos DB account. Search for and select **Azure Cosmos DB**.

Azure Cosmos DB

Services

- Azure Cosmos DB**
- Azure Database for MySQL servers
- Azure Databricks
- Azure DevOps
- Azure Lighthouse
- Azure Migrate
- Azure Sentinel
- Azure SQL
- Azure Database for MariaDB servers
- Azure Active Directory

Resources

No results were found.

2. Select **Add**.
3. 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 <b>Create new</b> , then enter a unique name for the new resource group.
Account Name	A unique name	<p>Enter a name to identify your Azure Cosmos account. Because <i>documents.azure.com</i> is appended to the name that you provide to create your URI, use a unique name.</p> <p>The name can only contain lowercase letters, numbers, and the hyphen (-) character. It must be between 3-31 characters in length.</p>

SETTING	VALUE	DESCRIPTION
API	The type of account to create	<p>Select <b>Core (SQL)</b> to create a document database and query by using SQL syntax.</p> <p>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.</p> <p><a href="#">Learn more about the SQL API.</a></p>
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.

Dashboard > New > Create Azure Cosmos DB Account

## Create Azure Cosmos DB Account

[Basics](#) [Network](#) [Tags](#) [Review + create](#)

Azure Cosmos DB is a fully managed globally distributed, multi-model database service, transparently replicating your data across any number of Azure regions. You can elastically scale throughput and storage, and take advantage of fast, single-digit-millisecond data access using the API of your choice backed by 99.999 SLA. [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	Contoso Subscription
└─ * Resource Group	(New) myResourceGroup
	<a href="#">Create new</a>

**INSTANCE DETAILS**

* Account Name	mysqlapicosmosdb
	documents.azure.com
* API ⓘ	Core (SQL)
* Location	West US
Geo-Redundancy ⓘ	<a href="#">Enable</a> <a href="#">Disable</a>
Multi-region Writes ⓘ	<a href="#">Enable</a> <a href="#">Disable</a>

[Review + create](#) [Previous](#) [Next: Network](#)

4. Select **Review + create**. You can skip the **Network** and **Tags** sections.
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**.

The screenshot shows the 'Deployment' section of the Azure Cosmos DB overview. It displays a success message: 'Your deployment is complete'. Below this, it shows deployment details: name (Microsoft.Azure.CosmosDB-2019032100000), subscription (Contoso Subscription), and resource group (myResourceGroup). Deployment details include start time (3/21/2019, 5:00:03 PM), duration (5 minutes 38 seconds), and correlation ID (8e0be948-0c60-4da0-0000-000000000000). A table lists the resource 'mysqlapicosmosdb' as Microsoft.DocumentDb/databaseAcc... with status OK and operation details.

6. Select **Go to resource** to go to the Azure Cosmos DB account page.

The screenshot shows the 'Quick start' section of the Azure Cosmos DB account. It congratulates the user on account creation and provides instructions to connect using a sample app. It offers to choose a platform (.NET, .NET Core, Xamarin, Java, Node.js, Python) and provides steps to add a collection and download/run a .NET app. The left sidebar lists other account management options like Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Notifications, Data Explorer, Settings, Replicate data globally, Default consistency, and Firewall and virtual networks.

## Step 2: Clone the GitHub project

You can get started by cloning the GitHub repository for [Get Started with Azure Cosmos DB and Java](#). For example, from a local directory run the following to retrieve the sample project locally.

```
git clone git@github.com:Azure-Samples/azure-cosmos-db-documentdb-java-getting-started.git
cd azure-cosmos-db-documentdb-java-getting-started
```

The directory contains a `pom.xml` for the project and a `src` folder containing Java source code including `Program.java` which shows how to perform simple operations with Azure Cosmos DB like creating documents and querying data within a collection. The `pom.xml` includes a dependency on the [Azure Cosmos DB Java SDK on Maven](#).

```
<dependency>
    <groupId>com.microsoft.azure</groupId>
    <artifactId>azure-documentdb</artifactId>
    <version>LATEST</version>
</dependency>
```

## Step 3: Connect to an Azure Cosmos DB account

Next, head back to the [Azure portal](#) to retrieve your endpoint and primary master key. The Azure Cosmos DB endpoint and primary key are necessary for your application to understand where to connect to, and for Azure Cosmos DB to trust your application's connection.

In the Azure portal, navigate to your Azure Cosmos DB account, and then click **Keys**. Copy the URI from the portal and paste it into `https://FILLME.documents.azure.com` in the Program.java file. Then copy the PRIMARY KEY from the portal and paste it into `FILLME`.

```
this.client = new DocumentClient(  
    "https://FILLME.documents.azure.com",  
    "FILLME"  
, new ConnectionPolicy(),  
    ConsistencyLevel.Session);
```

The screenshot shows the Azure portal interface for managing an Azure Cosmos DB account. The main title is 'mysqlapicosmosdb - Keys'. On the left, there's a sidebar with various icons and links. The 'Keys' link is highlighted with a red box. The main content area has tabs for 'Read-write Keys' and 'Read-only Keys', with 'Read-write Keys' being active. It displays the 'URI' (https://mysqlapicosmosdb.documents.azure.com:443/), 'PRIMARY KEY' (a long string of characters), 'SECONDARY KEY' (another long string), 'PRIMARY CONNECTION STRING' (AccountEndpoint=https://mysqlapicosmosdb.documents.azure.com:443/;AccountKey=19ZDNJ...), and 'SECONDARY CONNECTION STRING' (AccountEndpoint=https://mysqlapicosmosdb.documents.azure.com:443/;AccountKey=2RvOFDDRA...). Each of these strings has a blue copy icon next to it.

## Step 4: Create a database

Your Azure Cosmos DB [database](#) can be created by using the `createDatabase` method of the **DocumentClient** class. A database is the logical container of JSON document storage partitioned across collections.

```
Database database = new Database();  
database.setId("familydb");  
this.client.createDatabase(database, null);
```

## Step 5: Create a collection

### WARNING

**createCollection** creates a new collection with reserved throughput, which has pricing implications. For more details, visit our [pricing page](#).

A collection can be created by using the `createCollection` method of the **DocumentClient** class. A collection is a container of JSON documents and associated JavaScript application logic.

```

DocumentCollection collectionInfo = new DocumentCollection();
collectionInfo.setId("familycoll");

// Azure Cosmos containers can be reserved with throughput specified in request units/second.
// Here we create a collection with 400 RU/s.
RequestOptions requestOptions = new RequestOptions();
requestOptions.setOfferThroughput(400);

this.client.createCollection("/dbs/familydb", collectionInfo, requestOptions);

```

## Step 6: Create JSON documents

A document can be created by using the [createDocument](#) method of the **DocumentClient** class. Documents are user-defined (arbitrary) JSON content. We can now insert one or more documents. If you already have data you'd like to store in your database, you can use Azure Cosmos DB's [Data Migration tool](#) to import the data into a database.

```

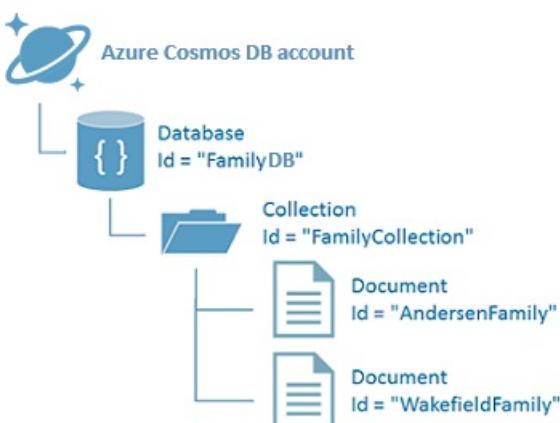
// Insert your Java objects as documents
Family andersenFamily = new Family();
andersenFamily.setId("Andersen.1");
andersenFamily.setLastName("Andersen")

// More initialization skipped for brevity. You can have nested references
andersenFamily.setParents(new Parent[] { parent1, parent2 });
andersenFamily.setDistrict("WA5");
Address address = new Address();
address.setCity("Seattle");
address.setCounty("King");
address.setState("WA");

andersenFamily.setAddress(address);
andersenFamily.setRegistered(true);

this.client.createDocument("/dbs/familydb/colls/familycoll", family, new RequestOptions(), true);

```



## Step 7: Query Azure Cosmos DB resources

Azure Cosmos DB supports rich [queries](#) against JSON documents stored in each collection. The following sample code shows how to query documents in Azure Cosmos DB using SQL syntax with the [queryDocuments](#) method.

```
FeedResponse<Document> queryResults = this.client.queryDocuments(
    "/dbs/familydb/colls/familycoll",
    "SELECT * FROM Family WHERE Family.lastName = 'Andersen'",
    null);

System.out.println("Running SQL query...");
for (Document family : queryResults.getQueryIterable()) {
    System.out.println(String.format("\tRead %s", family));
}
```

## Step 8: Replace JSON document

Azure Cosmos DB supports updating JSON documents using the [replaceDocument](#) method.

```
// Update a property
andersenFamily.Children[0].Grade = 6;

this.client.replaceDocument(
    "/dbs/familydb/colls/familycoll/docs/Andersen.1",
    andersenFamily,
    null);
```

## Step 9: Delete JSON document

Similarly, Azure Cosmos DB supports deleting JSON documents using the [deleteDocument](#) method.

```
this.client.delete("/dbs/familydb/colls/familycoll/docs/Andersen.1", null);
```

## Step 10: Delete the database

Deleting the created database removes the database and all children resources (collections, documents, etc.).

```
this.client.deleteDatabase("/dbs/familydb", null);
```

## Step 11: Run your Java console application all together!

To run the application from the console, navigate to the project folder and compile using Maven:

```
mvn package
```

Running `mvn package` downloads the latest Azure Cosmos DB library from Maven and produces `GetStarted-0.0.1-SNAPSHOT.jar`. Then run the app by running:

```
mvn exec:java -D exec.mainClass=GetStarted.Program
```

Congratulations! You've completed this NoSQL tutorial and have a working Java console application!

## Next steps

- Want a Java web app tutorial? See [Build a web application with Java using Azure Cosmos DB](#).
- Learn how to [monitor an Azure Cosmos DB account](#).

- Run queries against our sample dataset in the [Query Playground](#).

# Tutorial: Build a Java app with the Async Java SDK to manage data stored in a SQL API account

12/13/2019 • 6 minutes to read • [Edit Online](#)

As developer, you might have applications that use NoSQL document data. You can use the SQL API account in Azure Cosmos DB to store and access this document data. This tutorial shows you how to build a Java application with the Async Java SDK to store and manage document data.

This tutorial covers the following tasks:

- Creating and connecting to an Azure Cosmos account
- Configuring your solution
- Creating a collection
- Creating JSON documents
- Querying the collection

## 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 account](#).
- [Git](#).
- [Java Development Kit \(JDK\) 8+](#).
- [Maven](#).

## Create an Azure Cosmos DB account

Create an Azure Cosmos account by using the following steps:

1. Go to the [Azure portal](#) to create an Azure Cosmos DB account. Search for and select **Azure Cosmos DB**.

Azure Cosmos DB

Services

- Azure Cosmos DB**
- Azure Database for MySQL servers
- Azure Databricks
- Azure DevOps
- Azure Lighthouse
- Azure Migrate
- Azure Sentinel
- Azure SQL
- Azure Database for MariaDB servers
- Azure Active Directory

Resources

No results were found.

2. Select **Add**.
3. 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 <b>Create new</b> , then enter a unique name for the new resource group.
Account Name	A unique name	<p>Enter a name to identify your Azure Cosmos account. Because <i>documents.azure.com</i> is appended to the name that you provide to create your URI, use a unique name.</p> <p>The name can only contain lowercase letters, numbers, and the hyphen (-) character. It must be between 3-31 characters in length.</p>

SETTING	VALUE	DESCRIPTION
API	The type of account to create	<p>Select <b>Core (SQL)</b> to create a document database and query by using SQL syntax.</p> <p>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.</p> <p><a href="#">Learn more about the SQL API.</a></p>
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.

Dashboard > New > Create Azure Cosmos DB Account

## Create Azure Cosmos DB Account

[Basics](#) [Network](#) [Tags](#) [Review + create](#)

Azure Cosmos DB is a fully managed globally distributed, multi-model database service, transparently replicating your data across any number of Azure regions. You can elastically scale throughput and storage, and take advantage of fast, single-digit-millisecond data access using the API of your choice backed by 99.999 SLA. [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	Contoso Subscription
└─ * Resource Group	(New) myResourceGroup
	<a href="#">Create new</a>

**INSTANCE DETAILS**

* Account Name	mysqlapicosmosdb
	documents.azure.com
* API ⓘ	Core (SQL)
* Location	West US
Geo-Redundancy ⓘ	<a href="#">Enable</a> <a href="#">Disable</a>
Multi-region Writes ⓘ	<a href="#">Enable</a> <a href="#">Disable</a>

[Review + create](#) [Previous](#) [Next: Network](#)

4. Select **Review + create**. You can skip the **Network** and **Tags** sections.
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**.

The screenshot shows the Azure Deployment Center interface. At the top, there's a search bar and standard navigation buttons: Delete, Cancel, Redeploy, and Refresh. On the left, a sidebar lists 'Overview', 'Inputs', 'Outputs', and 'Template'. The main area displays a green checkmark icon and the message 'Your deployment is complete'. Below this, it shows deployment details: name 'Microsoft.Azure.CosmosDB-20190321000000', subscription 'Contoso Subscription', and resource group 'myResourceGroup'. A table provides deployment summary information:

RESOURCE	TYPE	STATUS	OPERATION DETAILS
mysqlapicosmosdb	Microsoft.DocumentDb/databaseAcc...	OK	<a href="#">Operation details</a>

- Select **Go to resource** to go to the Azure Cosmos DB account page.

The screenshot shows the 'Quick start' page for the 'mysqlapicosmosdb' Azure Cosmos DB account. The left sidebar includes options like Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Quick start (which is selected), Notifications, Data Explorer, Settings, Replicate data globally, Default consistency, and Firewall and virtual networks. The main content area displays a success message: 'Congratulations! Your Azure Cosmos DB account was created.' It also provides instructions for connecting using different platforms: .NET, .NET Core, Xamarin, Java, Node.js, and Python. Step 1, 'Add a collection', is described with a note about storage in collections and a 'Create 'Items' collection' button. Step 2, 'Download and run your .NET app', is described with a note about downloading a sample .NET app and a 'Download' button.

## Clone the GitHub repository

Clone the GitHub repository for [Get Started with Azure Cosmos DB and Java](#). For example, from a local directory, run the following to retrieve the sample project locally.

```
git clone https://github.com/Azure-Samples/azure-cosmos-db-sql-api-async-java-getting-started.git
cd azure-cosmos-db-sql-api-async-java-getting-started
cd azure-cosmosdb-get-started
```

The directory contains a `pom.xml` file and a `src/main/java/com/microsoft/azure/cosmosdb/sample` folder containing Java source code, including `Main.java`. The project contains code required to perform operations with Azure Cosmos DB, like creating documents and querying data within a collection. The `pom.xml` file includes a dependency on the [Azure Cosmos DB Java SDK on Maven](#).

```
<dependency>
<groupId>com.microsoft.azure</groupId>
<artifactId>azure-documentdb</artifactId>
<version>2.0.0</version>
</dependency>
```

# Connect to an Azure Cosmos account

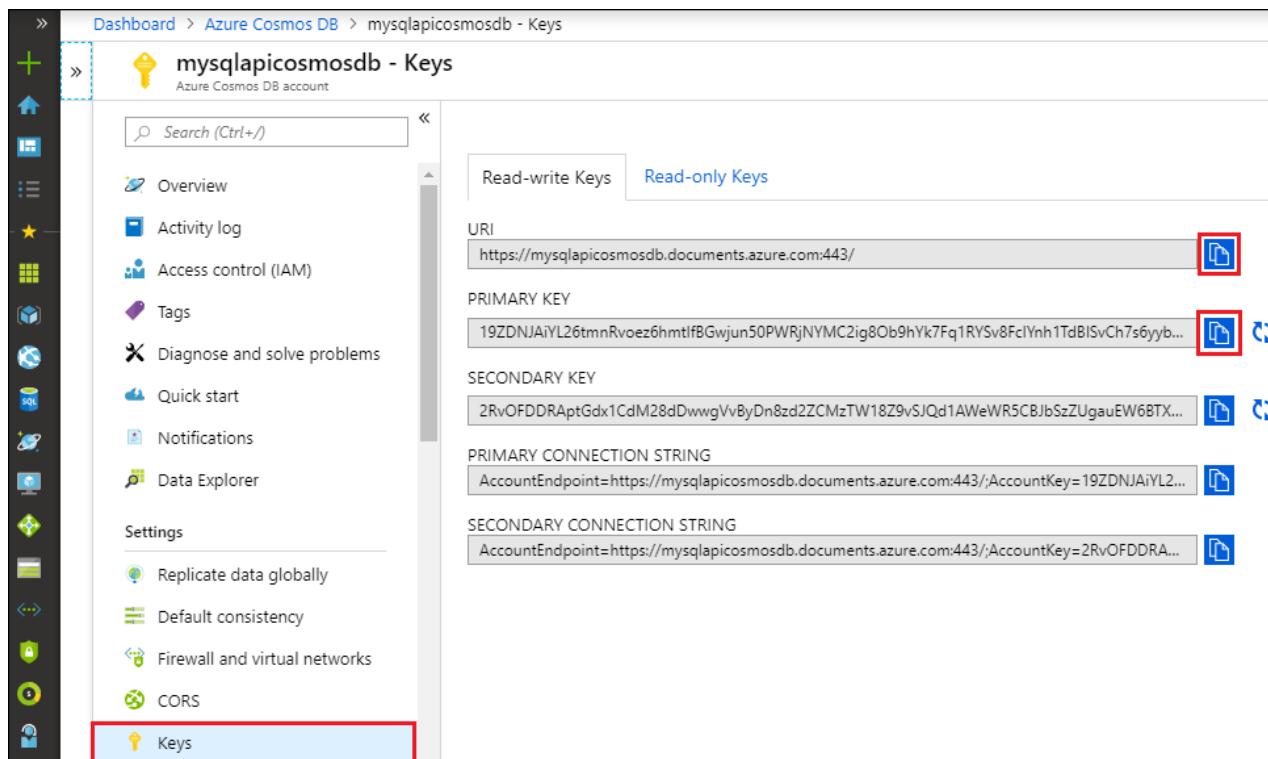
Next, head back to the [Azure portal](#) to retrieve your endpoint and primary master key. The Azure Cosmos DB endpoint and primary key are necessary for your application to understand where to connect to, and for Azure Cosmos DB to trust your application's connection. The `AccountSettings.java` file holds the primary key and URI values.

In the Azure portal, go to your Azure Cosmos account, and then click **Keys**. Copy the URI and the PRIMARY KEY from the portal and paste it into the `AccountSettings.java` file.

```
public class AccountSettings
{
    // Replace MASTER_KEY and HOST with values from your Azure Cosmos account.

    // <!--[SuppressMessage("Microsoft.Security", "CS002:SecretInNextLine")]->
    public static String MASTER_KEY = System.getProperty("ACCOUNT_KEY",
        StringUtils.defaultString(StringUtils.trimOrNull(
            System.getenv().get("ACCOUNT_KEY")), "<Fill your Azure Cosmos account key>"));

    public static String HOST = System.getProperty("ACCOUNT_HOST",
        StringUtils.defaultString(StringUtils.trimOrNull(
            System.getenv().get("ACCOUNT_HOST")), "<Fill your Azure Cosmos DB URI>"));
}
```



The screenshot shows the Azure portal interface for managing an Azure Cosmos DB account named "mysqlapicosmosdb". The left sidebar has a "Keys" icon highlighted with a red box. The main content area is titled "mysqlapicosmosdb - Keys" and displays the "Read-write Keys" tab. It shows the "URI" as "https://mysqlapicosmosdb.documents.azure.com:443/" with a copy icon. Below it are the "PRIMARY KEY" and "SECONDARY KEY" fields, each with a copy icon. Further down are the "PRIMARY CONNECTION STRING" and "SECONDARY CONNECTION STRING" fields, also with copy icons. The "Read-only Keys" tab is visible but not selected.

## Initialize the client object

Initialize the client object by using the host URI and primary key values defined in the "AccountSettings.java" file.

```
client = new AsyncDocumentClient.Builder()
    .withServiceEndpoint(AccountSettings.HOST)
    .withMasterKey(AccountSettings.MASTER_KEY)
    .withConnectionPolicy(ConnectionPolicy.GetDefault())
    .withConsistencyLevel(ConsistencyLevel.Session)
    .build();
```

## Create a database

Create your Azure Cosmos database by using the `createDatabaseIfNotExists()` method of the `DocumentClient` class. A database is the logical container of JSON document storage partitioned across collections.

```
private void createDatabaseIfNotExists() throws Exception
{
    writeToConsoleAndPromptToContinue("Check if database " + databaseName + " exists.");

    String databaseLink = String.format("/dbs/%s", databaseName);

    Observable<ResourceResponse<Database>> databaseReadObs = client.readDatabase(databaseLink, null);

    Observable<ResourceResponse<Database>> databaseExistenceObs = databaseReadObs
        .doOnNext(x -> {System.out.println("database " + databaseName + " already
exists."});}.onErrorResumeNext(e -> {
        // if the database doesn't already exists
        // readDatabase() will result in 404 error
        if (e instanceof DocumentClientException)
        {
            DocumentClientException de = (DocumentClientException) e;
            // if database
            if (de.getStatusCode() == 404) {
                // if the database doesn't exist, create it.
                System.out.println("database " + databaseName + " doesn't existed," + " creating it...");
                Database dbDefinition = new Database();
                dbDefinition.setId(databaseName);
                return client.createDatabase(dbDefinition, null);
            }
        }
    });

    // some unexpected failure in reading database happened.
    // pass the error up.
    System.err.println("Reading database " + databaseName + " failed.");
    return Observable.error(e);
});

// wait for completion
databaseExistenceObs.toCompletableFuture().await();

System.out.println("Checking database " + databaseName + " completed!\n");
}
```

## Create a collection

You can create a collection by using the `createDocumentCollectionIfNotExists()` method of the `DocumentClient` class. A collection is a container of JSON documents and the associated JavaScript application logic.

### WARNING

**createCollection** creates a new collection with reserved throughput, which has pricing implications. For more details, visit our [pricing page](#).

```

private void createDocumentCollectionIfNotExists() throws Exception
{

    writeToConsoleAndPromptToContinue("Check if collection " + collectionName + " exists.");

    // query for a collection with a given id
    // if it exists nothing else to be done
    // if the collection doesn't exist, create it.

    String databaseLink = String.format("/dbs/%s", databaseName);

    // we know there is only single page of result (empty or with a match)
    client.queryCollections(databaseLink, new SqlQuerySpec("SELECT * FROM r where r.id = @id",
        new SqlParameterCollection(new SqlParameter("@id", collectionName))), null).single()
        .flatMap(page -> {
            if (page.getResults().isEmpty()) {
                // if there is no matching collection create the collection.
                DocumentCollection collection = new DocumentCollection();
                collection.setId(collectionName);
                System.out.println("Creating collection " + collectionName);

                return client.createCollection(databaseLink, collection, null);
            }
            else {
                // collection already exists, nothing else to be done.
                System.out.println("Collection " + collectionName + " already exists");
                return Observable.empty();
            }
        })
        .toCompletableFuture().await();

    System.out.println("Checking collection " + collectionName + " completed!\n");
}

```

## Create JSON documents

Create a document by using the `createDocument` method of the `DocumentClient` class. Documents are user-defined (arbitrary) JSON content. We can now insert one or more documents. The "src/main/java/com/microsoft/azure/cosmosdb/sample/Families.java" file defines the family JSON documents.

```

public static Family getJohnsonFamilyDocument() {
    Family andersenFamily = new Family();
    andersenFamily.setId("Johnson" + System.currentTimeMillis());
    andersenFamily.setLastName("Johnson");

    Parent parent1 = new Parent();
    parent1.setFirstName("John");

    Parent parent2 = new Parent();
    parent2.setFirstName("Lili");

    return andersenFamily;
}

```

## Query Azure Cosmos DB resources

Azure Cosmos DB supports rich queries against JSON documents stored in each collection. The following sample code shows how to query documents in Azure Cosmos DB using SQL syntax with the `queryDocuments` method.

```

private void executeSimpleQueryAsyncAndRegisterListenerForResult(CountDownLatch completionLatch)
{
    // Set some common query options
    FeedOptions queryOptions = new FeedOptions();
    queryOptions.setMaxItemCount(100);
    queryOptions.setEnableCrossPartitionQuery(true);

    String collectionLink = String.format("/dbs/%s/colls/%s", databaseName, collectionName);
    Observable<FeedResponse<Document>> queryObservable = client.queryDocuments(collectionLink,
        "SELECT * FROM Family WHERE Family.lastName = 'Andersen'", queryOptions);

    queryObservable.subscribe(queryResultPage -> {
        System.out.println("Got a page of query result with " +
            queryResultPage.getResults().size() + " document(s)" +
            " and request charge of " + queryResultPage.getRequestCharge());
    },
    // terminal error signal
    e -> {
        e.printStackTrace();
        completionLatch.countDown();
    },
    // terminal completion signal
    () -> {
        completionLatch.countDown();
    });
}

```

## Run your Java console application

To run the application from the console, go to the project folder and compile using Maven:

```
mvn package
```

Running `mvn package` downloads the latest Azure Cosmos DB library from Maven and produces `GetStarted-0.0.1-SNAPSHOT.jar`. Then run the app by running:

```
mvn exec:java -DACCOUNT_HOST=<YOUR_COSMOS_DB_HOSTNAME> -DACCOUNT_KEY= <YOUR_COSMOS_DB_MASTER_KEY>
```

You've now completed this NoSQL tutorial and have a working Java console application.

## 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 build a Java app with the Async Java SDK to manage SQL API data in Azure Cosmos DB. You can now proceed to the next article:

[Build a Node.js console app with JavaScript SDK and Azure Cosmos DB](#)

# Tutorial: Build a Node.js console app with the JavaScript SDK to manage Azure Cosmos DB SQL API data

2/24/2020 • 16 minutes to read • [Edit Online](#)

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, you can 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. Go to the [Azure portal](#) to create an Azure Cosmos DB account. Search for and select **Azure Cosmos DB**.

The screenshot shows the Azure search interface with the title bar "Azure Cosmos DB". Under the "Services" heading, there is a list of services. The first item, "Azure Cosmos DB", is highlighted with a red box. Other items in the list include "Azure Database for MySQL servers", "Azure Databricks", "Azure DevOps", "Azure Lighthouse", "Azure Migrate", "Azure Sentinel", "Azure SQL", "Azure Database for MariaDB servers", and "Azure Active Directory". Below the "Services" heading, under "Resources", it says "No results were found."

2. Select **Add**.
3. 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 <b>Create new</b> , then enter a unique name for the new resource group.
Account Name	A unique name	<p>Enter a name to identify your Azure Cosmos account. Because <i>documents.azure.com</i> is appended to the name that you provide to create your URL, use a unique name.</p> <p>The name can only contain lowercase letters, numbers, and the hyphen (-) character. It must be between 3-31 characters in length.</p>

Setting	Value	Description
API	The type of account to create	<p>Select <b>Core (SQL)</b> to create a document database and query by using SQL syntax.</p> <p>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.</p> <p><a href="#">Learn more about the SQL API.</a></p>
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.

Dashboard > New > Create Azure Cosmos DB Account

## Create Azure Cosmos DB Account

[Basics](#) [Network](#) [Tags](#) [Review + create](#)

Azure Cosmos DB is a fully managed globally distributed, multi-model database service, transparently replicating your data across any number of Azure regions. You can elastically scale throughput and storage, and take advantage of fast, single-digit-millisecond data access using the API of your choice backed by 99.999 SLA. [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	Contoso Subscription
└─ * Resource Group	(New) myResourceGroup
	<a href="#">Create new</a>

**INSTANCE DETAILS**

* Account Name	mysqlapicosmosdb	documents.azure.com
* API ⓘ	Core (SQL)	
* Location	West US	
Geo-Redundancy ⓘ	<a href="#">Enable</a> <a href="#">Disable</a>	
Multi-region Writes ⓘ	<a href="#">Enable</a> <a href="#">Disable</a>	

[Review + create](#) [Previous](#) [Next: Network](#)

4. Select **Review + create**. You can skip the **Network** and **Tags** sections.
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**.

The screenshot shows the 'Deployment' section of the Azure CosmosDB overview. It displays a success message: 'Your deployment is complete'. Below this, it shows deployment details: name (Microsoft.Azure.CosmosDB-20190321000000), subscription (Contoso Subscription), and resource group (myResourceGroup). Deployment occurred at 3/21/2019, 5:00:03 PM, took 5 minutes 38 seconds, and has a correlation ID of 8e0be948-0c60-4da0-0000-000000000000. A table lists the deployed resource, type (Microsoft.DocumentDb/databaseAcc...), status (OK), and operation details.

RESOURCE	TYPE	STATUS	OPERATION DETAILS
mysqlapicosmosdb	Microsoft.DocumentDb/databaseAcc...	OK	<a href="#">Operation details</a>

6. Select **Go to resource** to go to the Azure Cosmos DB account page.

The screenshot shows the 'Quick start' section of the Azure Cosmos DB account page for 'mysqlapicosmosdb'. It congratulates the user on account creation and provides instructions to connect using a sample app. It offers to choose a platform (.NET, .NET Core, Xamarin, Java, Node.js, Python) and provides steps to add a collection and download a .NET app. The 'Quick start' option is highlighted in the left sidebar.

## 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 two empty JavaScript files with the following commands:

- Windows:

- `fsutil file createnew app.js 0`
- `fsutil file createnew config.js 0`

- Linux/OS X:

- `touch app.js`
- `touch config.js`

4. Create and initialize a `package.json` file. Use the following command:

- `npm init -y`

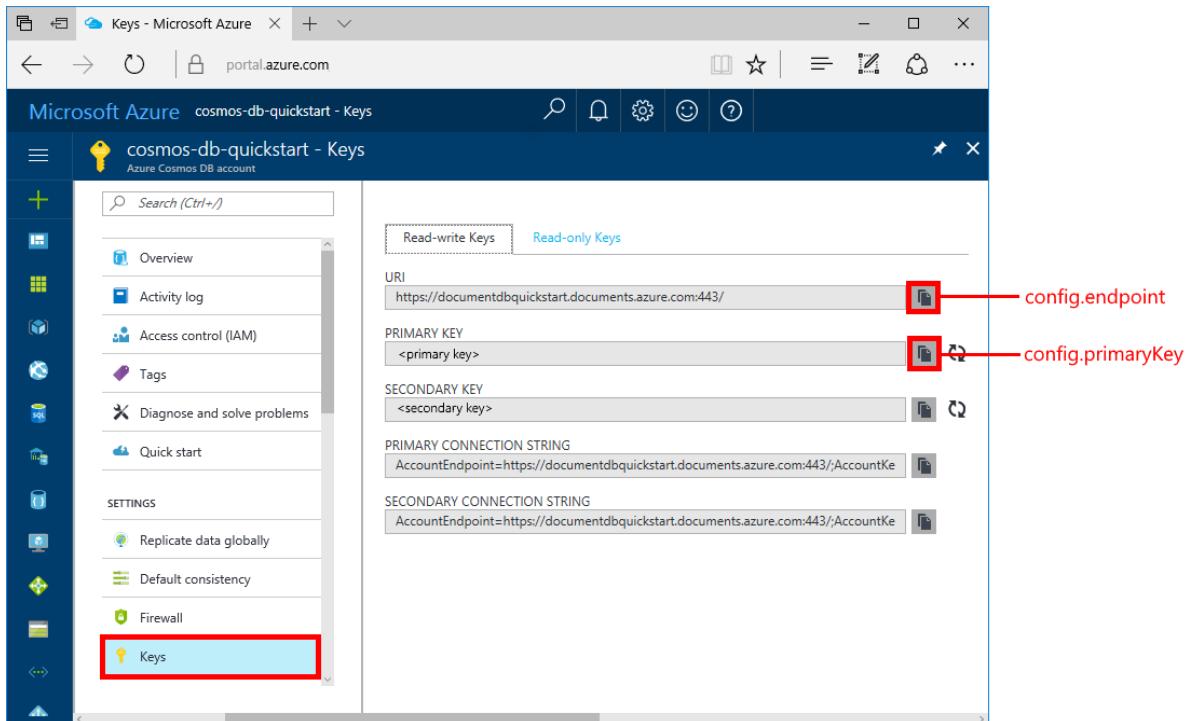
5. Install the `@azure/cosmos` module via npm. Use the following command:

- `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 `config.js` in your favorite text editor.
2. Copy and paste the code snippet below and set properties `config.endpoint` and `config.key` to your Azure Cosmos DB endpoint URI and primary key. Both these configurations can be found in the [Azure portal](#).



```
// ADD THIS PART TO YOUR CODE
var config = {}

config.endpoint = "~your Azure Cosmos DB endpoint uri here~";
config.key = "~your primary key here~";
```

3. Copy and paste the `database`, `container`, and `items` data to your `config` object below where you set your `config.endpoint` and `config.key` properties. If you already have data you'd like to store in your database, you can use the Data Migration tool in Azure Cosmos DB rather than defining the data here. Your `config.js` file should have the following code:

```
// @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;
```

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.

```
module.exports = config;
```

code is used to export your `config` object, so that you can reference it within the `app.js` file.

## Connect to an Azure Cosmos DB account

1. Open your empty `app.js` file in the text editor. Copy and paste the code below to import the `@azure/cosmos` module and your newly created `config` module.

```
// ADD THIS PART TO YOUR CODE
const CosmosClient = require('@azure/cosmos').CosmosClient;

const config = require('./config');
```

2. Copy and paste the code to use the previously saved `config.endpoint` and `config.key` to create a new `CosmosClient`.

```
const config = require('./config');

// ADD THIS PART TO YOUR CODE
const endpoint = config.endpoint;
const key = config.key;

const client = new CosmosClient({ endpoint, key });
```

### NOTE

If connecting to the **Cosmos DB Emulator**, disable SSL 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.

## Create a database

1. Copy and paste the code below to set the database ID, and the container ID. These IDs are how the Azure Cosmos DB client will find the right database and container.

```
const client = new CosmosClient({ endpoint, key });

// ADD THIS PART TO YOUR CODE
const HttpStatusCode = { NOTFOUND: 404 };

const databaseId = config.database.id;
const containerId = config.container.id;
const partitionKey = { kind: "Hash", paths: ["/Country"] };
```

A database can be created by using either the `createIfNotExists` or `create` function of the **Databases** class. A database is the logical container of items partitioned across containers.

2. Copy and paste the **createDatabase** and **readDatabase** methods into the `app.js` file under the `databaseId` and `containerId` definition. The **createDatabase** function will create a new database with ID

`FamilyDatabase`, specified from the `config` object if it does not already exist. The **readDatabase** function will read the database's definition to ensure that the database exists.

```
/**  
 * Create the database if it does not exist  
 */  
async function createDatabase() {  
    const { database } = await client.databases.createIfNotExists({ id: databaseId });  
    console.log(`Created database:\n${database.id}\n`);  
}  
  
/**  
 * Read the database definition  
 */  
async function readDatabase() {  
    const { resource: databaseDefinition } = await client.database(databaseId).read();  
    console.log(`Reading database:\n${databaseDefinition.id}\n`);  
}
```

3. Copy and paste the code below where you set the **createDatabase** and **readDatabase** functions to add the helper function **exit** that will print the exit message.

```
// ADD THIS PART TO YOUR CODE  
function exit(message) {  
    console.log(message);  
    console.log('Press any key to exit');  
    process.stdin.setRawMode(true);  
    process.stdin.resume();  
    process.stdin.on('data', process.exit.bind(process, 0));  
};
```

4. Copy and paste the code below where you set the **exit** function to call the **createDatabase** and **readDatabase** functions.

```
createDatabase()  
.then(() => readDatabase())  
.then(() => { exit(`Completed successfully`); })  
.catch((error) => { exit(`Completed with error ${JSON.stringify(error)}`) });
```

At this point, your code in `app.js` should now look as following code:

```

const CosmosClient = require('@azure/cosmos').CosmosClient;

const config = require('./config');

const endpoint = config.endpoint;
const key = config.key;

const client = new CosmosClient({ endpoint, key });

const HttpStatusCode = { NOTFOUND: 404 };

const databaseId = config.database.id;
const containerId = config.container.id;
const partitionKey = { kind: "Hash", paths: ["/Country"] };

/**
 * Create the database if it does not exist
 */
async function createDatabase() {
    const { database } = await client.databases.createIfNotExists({ id: databaseId });
    console.log(`Created database:\n${database.id}\n`);
}

/**
 * Read the database definition
 */
async function readDatabase() {
    const { resource: databaseDefinition } = await client.database(databaseId).read();
    console.log(`Reading database:\n${databaseDefinition.id}\n`);
}

/**
 * Exit the app with a prompt
 * @param {message} message - The message to display
 */
function exit(message) {
    console.log(message);
    console.log('Press any key to exit');
    process.stdin.setRawMode(true);
    process.stdin.resume();
    process.stdin.on('data', process.exit.bind(process, 0));
}

createDatabase()
    .then(() => readDatabase())
    .then(() => { exit(`Completed successfully`); })
    .catch((error) => { exit(`Completed with error ${JSON.stringify(error)} `) });

```

5. In your terminal, locate your `app.js` file and run the command:

```
node app.js
```

## Create a container

Next create a container within the Azure Cosmos DB account, so that you can store and query the data.

### WARNING

Creating a container has pricing implications. Visit our [pricing page](#) so you know what to expect.

A container can be created by using either the `createIfNotExists` or `create` function from the **Containers** class. A

container consists of items (which in the case of the SQL API is JSON documents) and associated JavaScript application logic.

1. Copy and paste the **createContainer** and **readContainer** function underneath the **readDatabase** function in the app.js file. The **createContainer** function will create a new container with the `containerId` specified from the `config` object if it does not already exist. The **readContainer** function will read the container definition to verify the container exists.

```
/**  
 * Create the container if it does not exist  
 */  
  
async function createContainer() {  
  
    const { container } = await client.database(databaseId).containers.createIfNotExists({ id:  
        containerId, partitionKey }, { offerThroughput: 400 });  
    console.log(`Created container:\n${config.container.id}\n`);  
}  
  
/**  
 * Read the container definition  
 */  
async function readContainer() {  
    const { resource: containerDefinition } = await  
        client.database(databaseId).container(containerId).read();  
    console.log(`Reading container:\n${containerDefinition.id}\n`);  
}
```

2. Copy and paste the code underneath the call to **readDatabase** to execute the **createContainer** and **readContainer** functions.

```
createDatabase()  
.then(() => readDatabase())  
  
// ADD THIS PART TO YOUR CODE  
.then(() => createContainer())  
.then(() => readContainer())  
// ENDS HERE  
  
.then(() => { exit(`Completed successfully`); })  
.catch((error) => { exit(`Completed with error ${JSON.stringify(error)}` )});
```

At this point, your code in `app.js` should now look like this:

```
const CosmosClient = require('@azure/cosmos').CosmosClient;  
  
const config = require('./config');  
  
const endpoint = config.endpoint;  
const key = config.key;  
  
const client = new CosmosClient({ endpoint, key });  
  
const HttpStatusCodes = { NOTFOUND: 404 };  
  
const databaseId = config.database.id;  
const containerId = config.container.id;  
const partitionKey = { kind: "Hash", paths: ["/Country"] };  
  
/**  
 * Create the database if it does not exist  
 */  
async function createDatabase() {
```

```

async function createDatabase() {
    const { database } = await client.databases.createIfNotExists({ id: databaseId });
    console.log(`Created database:\n${database.id}\n`);
}

/**
 * Read the database definition
 */
async function readDatabase() {
    const { body: databaseDefinition } = await client.database(databaseId).read();
    console.log(`Reading database:\n${databaseDefinition.id}\n`);
}

/**
 * Create the container if it does not exist
 */
async function createContainer() {

    const { container } = await client.database(databaseId).containers.createIfNotExists({ id: containerId, partitionKey }, { offerThroughput: 400 });
    console.log(`Created container:\n${config.container.id}\n`);
}

/**
 * Read the container definition
 */
async function readContainer() {
    const { resource: containerDefinition } = await client.database(databaseId).container(containerId).read();
    console.log(`Reading container:\n${containerDefinition.id}\n`);
}

/**
 * Exit the app with a prompt
 * @param {message} message - The message to display
 */
function exit(message) {
    console.log(message);
    console.log('Press any key to exit');
    process.stdin.setRawMode(true);
    process.stdin.resume();
    process.stdin.on('data', process.exit.bind(process, 0));
}

createDatabase()
    .then(() => readDatabase())
    .then(() => createContainer())
    .then(() => readContainer())
    .then(() => { exit(`Completed successfully`); })
    .catch((error) => { exit(`Completed with error ${JSON.stringify(error)}`) });

```

3. In your terminal, locate your `app.js` file and run the command:

```
node app.js
```

## 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. You can now insert an item into Azure Cosmos DB.

1. Copy and paste the **createFamilyItem** function underneath the **readContainer** function. The **createFamilyItem** function creates the items containing the JSON data saved in the `config` object. We'll

check to make sure an item with the same ID does not already exist before creating it.

```
/**  
 * Create family item  
 */  
async function createFamilyItem(itemBody) {  
    const { item } = await client.database(databaseId).container(containerId).items.upsert(itemBody);  
    console.log(`Created family item with id:\n${itemBody.id}\n`);  
};
```

2. Copy and paste the code below the call to **readContainer** to execute the **createFamilyItem** function.

```
createDatabase()  
.then(() => readDatabase())  
.then(() => createContainer())  
.then(() => readContainer())  
  
// ADD THIS PART TO YOUR CODE  
.then(() => createFamilyItem(config.items.Andersen))  
.then(() => createFamilyItem(config.items.Wakefield))  
// ENDS HERE  
  
.then(() => { exit(`Completed successfully`); })  
.catch((error) => { exit(`Completed with error ${JSON.stringify(error)}`) });
```

3. In your terminal, locate your `app.js` file and run the command:

```
node app.js
```

## Query Azure Cosmos DB resources

Azure Cosmos DB supports rich queries against JSON documents stored in each container. The following sample code shows a query that you can run against the documents in your container.

1. Copy and paste the **queryContainer** function below the **createFamilyItem** function in the `app.js` file.

Azure Cosmos DB supports SQL-like queries as shown below.

```

/**
 * Query the container using SQL
 */
async function queryContainer() {
    console.log(`Querying container:\n${config.container.id}`);

    // query to return all children in a family
    const querySpec = {
        query: "SELECT VALUE r.children FROM root r WHERE r.lastName = @lastName",
        parameters: [
            {
                name: "@lastName",
                value: "Andersen"
            }
        ]
    };
}

const { resources } = await client.database(databaseId).container(containerId).items.query(querySpec,
{enableCrossPartitionQuery:true}).fetchAll();
for (var queryResult of resources) {
    let resultString = JSON.stringify(queryResult);
    console.log(`\tQuery returned ${resultString}\n`);
}
};


```

2. Copy and paste the code below the calls to **createFamilyItem** to execute the **queryContainer** function.

```

createDatabase()
.then(() => readDatabase())
.then(() => createContainer())
.then(() => readContainer())
.then(() => createFamilyItem(config.items.Andersen))
.then(() => createFamilyItem(config.items.Wakefield))

// ADD THIS PART TO YOUR CODE
.then(() => queryContainer())
// ENDS HERE

.then(() => { exit(`Completed successfully`); })
.catch((error) => { exit(`Completed with error ${JSON.stringify(error)}` ) });

```

3. In your terminal, locate your `app.js` file and run the command:

```
node app.js
```

## Replace an item

Azure Cosmos DB supports replacing the content of items.

1. Copy and paste the **replaceFamilyItem** function below the **queryContainer** function in the `app.js` file.  
Note we've changed the property 'grade' of a child to 6 from the previous value of 5.

```
// ADD THIS PART TO YOUR CODE
/**
 * Replace the item by ID.
 */
async function replaceFamilyItem(itemBody) {
  console.log(`Replacing item:\n${itemBody.id}\n`);
  // Change property 'grade'
  itemBody.children[0].grade = 6;
  const { item } = await client.database(databaseId).container(containerId).item(itemBody.id,
  itemBody.Country).replace(itemBody);
}
```

- Copy and paste the code below the call to **queryContainer** to execute the **replaceFamilyItem** function. Also, add the code to call **queryContainer** again to verify that item has successfully changed.

```
createDatabase()
  .then(() => readDatabase())
  .then(() => createContainer())
  .then(() => readContainer())
  .then(() => createFamilyItem(config.items.Andersen))
  .then(() => createFamilyItem(config.items.Wakefield))
  .then(() => queryContainer())

// ADD THIS PART TO YOUR CODE
  .then(() => replaceFamilyItem(config.items.Andersen))
  .then(() => queryContainer())
// ENDS HERE

  .then(() => { exit(`Completed successfully`); })
  .catch((error) => { exit(`Completed with error ${JSON.stringify(error)}`) });

```

- In your terminal, locate your `app.js` file and run the command:

```
node app.js
```

## Delete an item

Azure Cosmos DB supports deleting JSON items.

- Copy and paste the **deleteFamilyItem** function underneath the **replaceFamilyItem** function.

```
/**
 * Delete the item by ID.
 */
async function deleteFamilyItem(itemBody) {
  await client.database(databaseId).container(containerId).item(itemBody.id,
  itemBody.Country).delete(itemBody);
  console.log(`Deleted item:\n${itemBody.id}\n`);
}
```

- Copy and paste the code below the call to the second **queryContainer** to execute the **deleteFamilyItem** function.

```

createDatabase()
  .then(() => readDatabase())
  .then(() => createContainer())
  .then(() => readContainer())
  .then(() => createFamilyItem(config.items.Andersen))
  .then(() => createFamilyItem(config.items.Wakefield))
  .then(() => queryContainer()
  ())
  .then(() => replaceFamilyItem(config.items.Andersen))
  .then(() => queryContainer())

  // ADD THIS PART TO YOUR CODE
  .then(() => deleteFamilyItem(config.items.Andersen))
  // ENDS HERE

  .then(() => { exit(`Completed successfully`); })
  .catch((error) => { exit(`Completed with error ${JSON.stringify(error)}` ) });

```

- In your terminal, locate your `app.js` file and run the command:

```
node app.js
```

## Delete the database

Deleting the created database will remove the database and all children resources (containers, items, etc.).

- Copy and paste the **cleanup** function underneath the **deleteFamilyItem** function to remove the database and all its children resources.

```

/**
 * Cleanup the database and container on completion
 */
async function cleanup() {
  await client.database(databaseId).delete();
}

```

- Copy and paste the code below the call to **deleteFamilyItem** to execute the **cleanup** function.

```

createDatabase()
  .then(() => readDatabase())
  .then(() => createContainer())
  .then(() => readContainer())
  .then(() => createFamilyItem(config.items.Andersen))
  .then(() => createFamilyItem(config.items.Wakefield))
  .then(() => queryContainer())
  .then(() => replaceFamilyItem(config.items.Andersen))
  .then(() => queryContainer())
  .then(() => deleteFamilyItem(config.items.Andersen))

  // ADD THIS PART TO YOUR CODE
  .then(() => cleanup())
  // ENDS HERE

  .then(() => { exit(`Completed successfully`); })
  .catch((error) => { exit(`Completed with error ${JSON.stringify(error)}` ) });

```

## Run your Node.js application

Altogether, your code should look like this:

```

// @ts-check
const CosmosClient = require("@azure/cosmos").CosmosClient;
const config = require("./config");
const dbContext = require("./data/databaseContext");

const newItem = {
  id: "3",
  category: "fun",
  name: "Cosmos DB",
  description: "Complete Cosmos DB Node.js Quickstart 🎉",
  isComplete: false
};

async function main() {
  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);

  try {
    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();

    console.log(items);

    // Create a new item
    const { resource: createdItem } = await container.items.create(newItem);
    console.log(`Created item: %s`, createdItem);

    const { id, category } = createdItem;

    // Update the item
    createdItem.isComplete = true;
    const { resource: updatedItem } = await container
      .item(id, category)
      .replace(createdItem);
    console.log(`Updated item: %s`, updatedItem);

    // Delete the item
    const { resource: result } = await container.item(id, category).delete();
    console.log("Deleted item with id: %s", id);
  } 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:  
FamilyDatabase  
  
Reading database:  
FamilyDatabase  
  
Created container:  
FamilyContainer  
  
Reading container:  
FamilyContainer  
  
Created family item with id:  
Anderson.1  
  
Created family item with id:  
Wakefield.7  
  
Querying container:  
FamilyContainer  
    Query returned [{"firstName": "Henriette Thaulow", "gender": "female", "grade": 5, "pets": [{"givenName": "Fluffy"}]}]  
  
Replacing item:  
Anderson.1  
  
Querying container:  
FamilyContainer  
    Query returned [{"firstName": "Henriette Thaulow", "gender": "female", "grade": 6, "pets": [{"givenName": "Fluffy"}]}]  
  
Deleted item:  
Anderson.1  
  
Completed successfully  
Press any key to exit
```

## 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

2/28/2020 • 13 minutes to read • [Edit Online](#)

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:

## List of To-Do Items

Name	Description	Completed	
ASP.NET Core MVC Tutorial	Learn about Cosmos DB	<input type="checkbox"/>	<a href="#">Edit</a>   <a href="#">Details</a>   <a href="#">Delete</a>

[Create New](#)

© 2019 - CosmosWebSample

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, you can 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. Go to the [Azure portal](#) to create an Azure Cosmos DB account. Search for and select **Azure Cosmos DB**.

The screenshot shows the Azure portal search interface. A search bar at the top contains the text "Azure Cosmos DB". Below the search bar, there are two main sections: "Services" and "Resources". In the "Services" section, several items are listed with icons: Azure Cosmos DB (highlighted with a red box), Azure Database for MySQL servers, Azure Databricks, Azure DevOps, Azure Lighthouse, Azure Migrate, Azure Sentinel, Azure SQL, Azure Database for MariaDB servers, and Azure Active Directory. In the "Resources" section, it says "No results were found."

2. Select **Add**.
3. 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 <b>Create new</b> , then enter a unique name for the new resource group.
Account Name	A unique name	<p>Enter a name to identify your Azure Cosmos account. Because <i>documents.azure.com</i> is appended to the name that you provide to create your URI, use a unique name.</p> <p>The name can only contain lowercase letters, numbers, and the hyphen (-) character. It must be between 3-31 characters in length.</p>

Setting	Value	Description
API	The type of account to create	<p>Select <b>Core (SQL)</b> to create a document database and query by using SQL syntax.</p> <p>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.</p> <p><a href="#">Learn more about the SQL API.</a></p>
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.

Dashboard > New > Create Azure Cosmos DB Account

## Create Azure Cosmos DB Account

[Basics](#) [Network](#) [Tags](#) [Review + create](#)

Azure Cosmos DB is a fully managed globally distributed, multi-model database service, transparently replicating your data across any number of Azure regions. You can elastically scale throughput and storage, and take advantage of fast, single-digit-millisecond data access using the API of your choice backed by 99,999 SLA. [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: Contoso Subscription

\* Resource Group: (New) myResourceGroup [Create new](#)

**INSTANCE DETAILS**

\* Account Name: mysqlapicosmosdb [documents.azure.com](#)

\* API: Core (SQL)

\* Location: West US

Geo-Redundancy: [Enable](#) [Disable](#)

Multi-region Writes: [Enable](#) [Disable](#)

[Review + create](#) [Previous](#) [Next: Network](#)

4. Select **Review + create**. You can skip the **Network** and **Tags** sections.
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**.

The screenshot shows the 'Deployment' section of the Azure Cosmos DB overview. It displays a success message: 'Your deployment is complete'. Below this, it shows deployment details: name (Microsoft.Azure.CosmosDB-20190321000000), subscription (Contoso Subscription), and resource group (myResourceGroup). Deployment occurred at 3/21/2019, 5:00:03 PM, took 5 minutes 38 seconds, and had a correlation ID of 8e0be948-0c60-4da0-0000-000000000000. A table summarizes the deployment status:

RESOURCE	TYPE	STATUS	OPERATION DETAILS
mysqlapicosmosdb	Microsoft.DocumentDb/databaseAcc...	OK	<a href="#">Operation details</a>

6. Select **Go to resource** to go to the Azure Cosmos DB account page.

The screenshot shows the 'Quick start' section of the Azure Cosmos DB account page. It congratulates the user on account creation and provides instructions to connect using a sample app. It lists supported platforms: .NET, .NET Core, Xamarin, Java, Node.js, and Python. Step 1: 'Add a collection' is described as storing data in collections, with a link to 'Create 'Items' collection'. Step 2: 'Download and run your .NET app' is described as downloading a sample app, with a 'Download' button.

Go to the Azure Cosmos DB account page, and select **Keys**. Copy the values to use in the web application you create next.

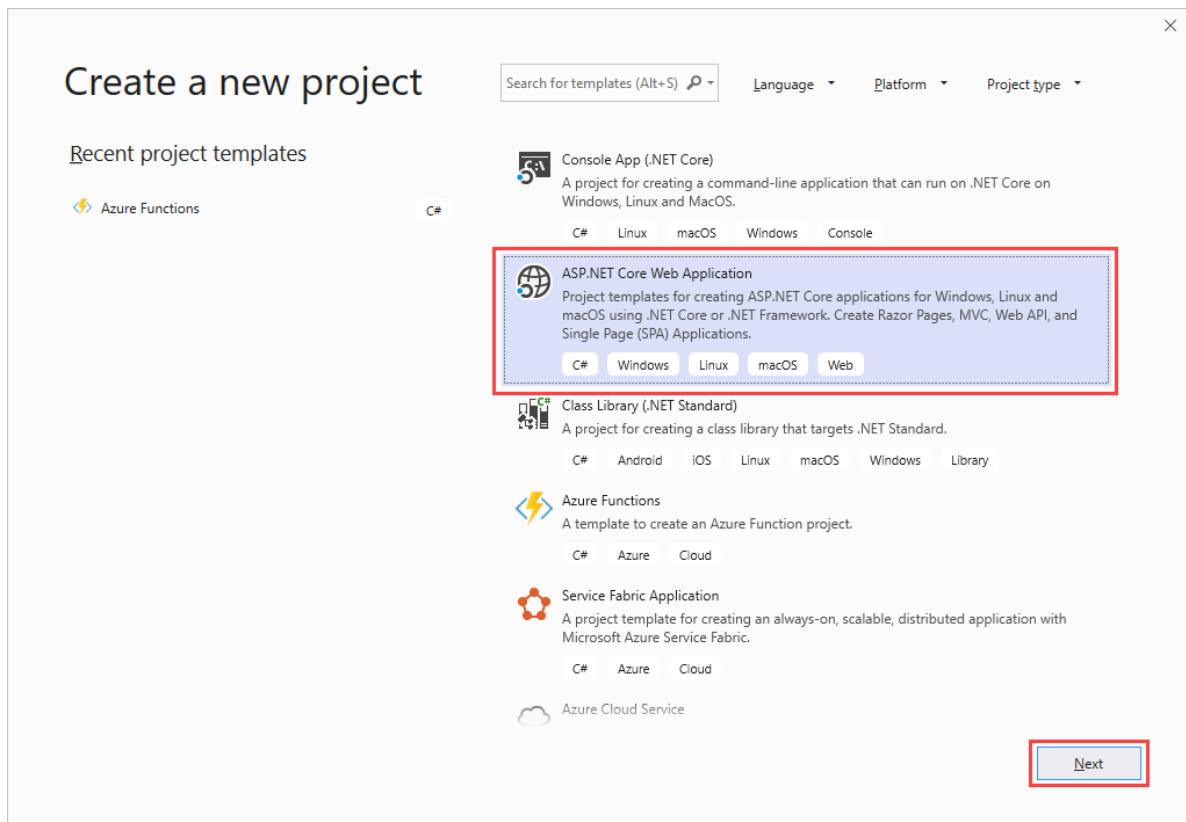
The screenshot shows the 'Keys' section of the Azure Cosmos DB account page. It displays two tabs: 'Read-write Keys' (selected) and 'Read-only Keys'. Under 'Read-write Keys', the 'URI' field contains https://contoso-demo.documents.azure.com:443/. The 'PRIMARY KEY' field contains a long base64 string: MECyqFOL6eGkv7d3YUuZoQY8ej4ZpQJ2fW30YgP2GNLlYaaNoHoqRYbOPNdRLQjOwNx5oxe3sPEFn0PQ2T0V7==. The 'SECONDARY KEY' field contains another base64 string: A1qltpGX6GyekSpljcYfNv0RAO91eCaMfNL1Ge60b6GtYszQYBU5pWZvXzufWvJB1X4T0e0sfXYNuWe95pv==. Under 'PRIMARY CONNECTION STRING', it shows AccountEndpoint=https://contoso-demo.documents.azure.com:443/AccountKey=WFa5AYX1OLxpDrWZJ0OrjOeJmWLhN4pVuXS5QjSQX7... Under 'SECONDARY CONNECTION STRING', it shows AccountEndpoint=https://contoso-demo.documents.azure.com:443/AccountKey=1ShVfjwnHLBVGBrkFTkeTS3AERH5cgDDjGLqDxD4Dkay... Both connection strings have copy icons.

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.

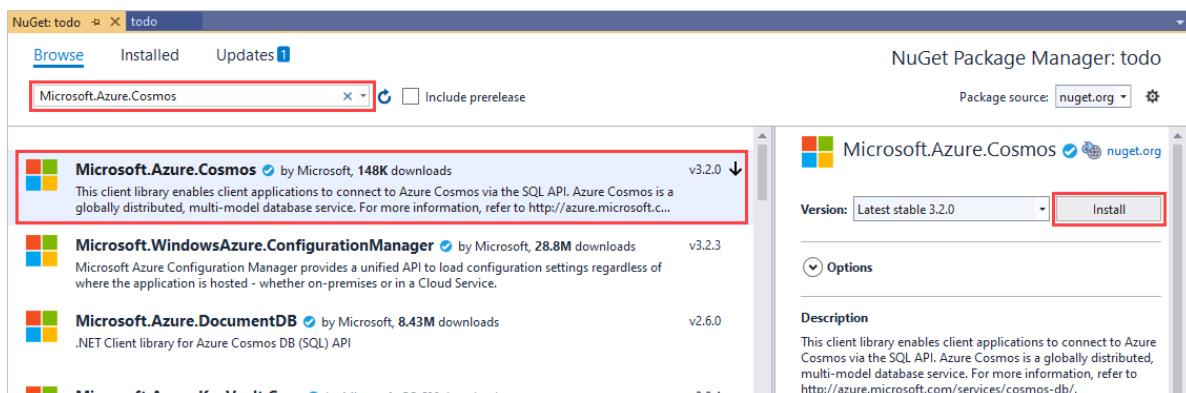
5. Select **Debug > Start Debugging** or F5 to run your ASP.NET application locally.

## 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`.

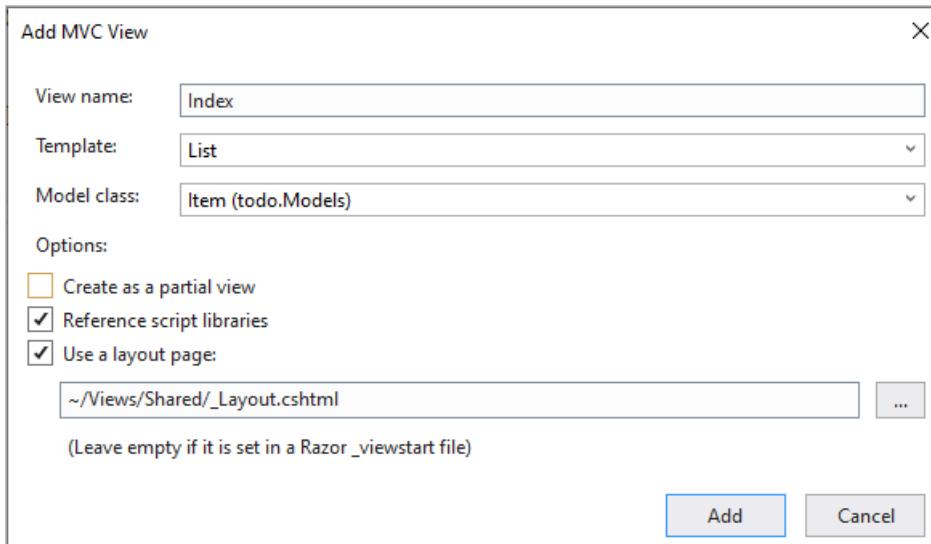
### Add views

Next, let's create the following three views.

- Add a list item view
- Add a new item view
- Add an edit item view

#### Add a list 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**, provide the following values:
  - In **View name**, enter *Index*.
  - In **Template**, select **List**.
  - In **Model class**, select **Item (todo.Models)**.
  - Select **Use a layout page** and enter `~/Views/Shared/_Layout.cshtml`.



4. After you add these values, select **Add** and let Visual Studio create a new template view.

Once done, Visual Studio opens the *cshtml* file that it creates. You can close that file in Visual Studio. We'll come back to it later.

#### Add a new item view

Similar to how you created a view to list items, create a new view to create items by using the following steps:

1. In **Solution Explorer**, right-click the **Item** folder again, select **Add > View**.
2. 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**.

#### Add an edit item view

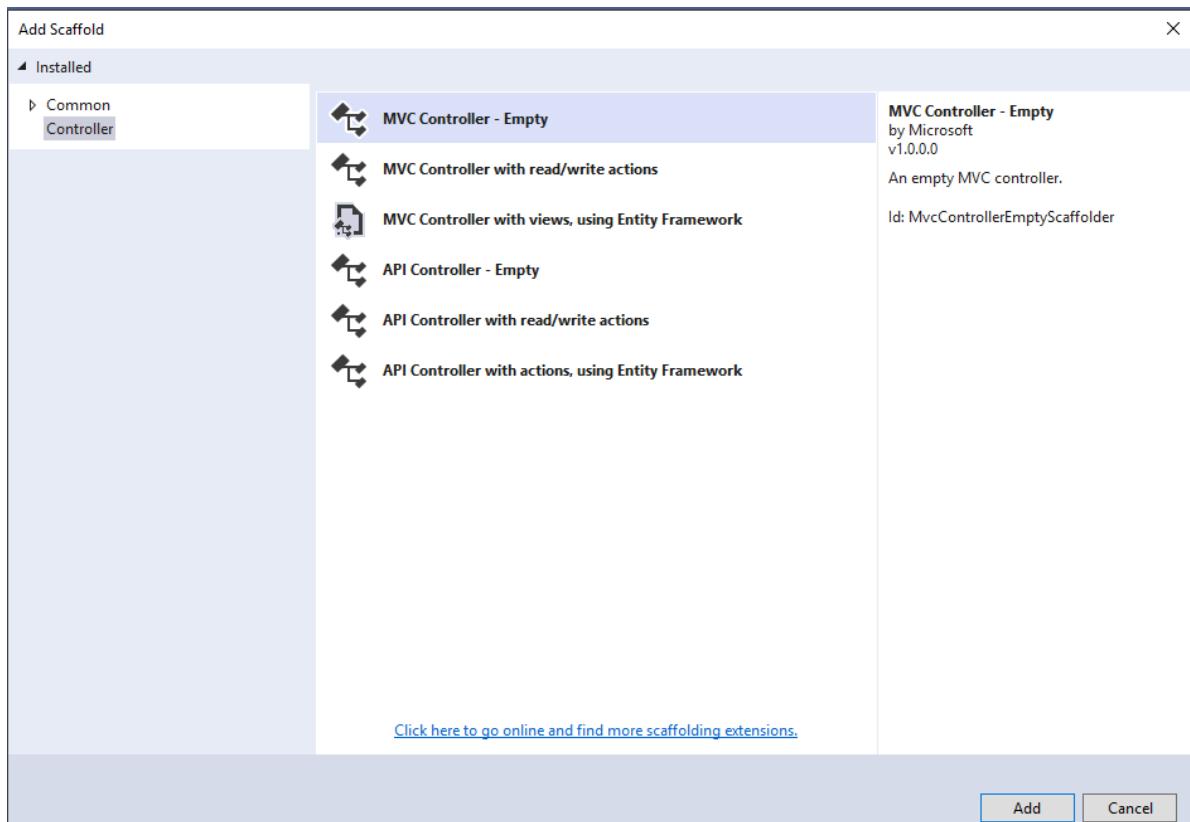
And finally, add a view to edit an item with the following steps:

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 *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**.

Once you complete these steps, close all the *cshtml* documents in Visual Studio as you return to these views later.

#### 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 todo.Services;
    using Microsoft.AspNetCore.Mvc;
    using 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)
            {
                await _cosmosDbService.CreateItemAsync(item);
                return RedirectToAction("Index");
            }
            return View(item);
        }
    }
}
```

```

        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: Connect to Azure Cosmos DB

Now that the standard MVC stuff is taken care of, let's turn to adding the code to connect to Azure Cosmos DB and do CRUD operations.

### Perform CRUD operations on the data

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.Services
{
    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. Repeat the previous two steps, but this time, use the name *ICosmosDBService*, and use the following code:

```

namespace todo.Services
{
    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);
    }
}

```

5. Register the **CosmosDBService** implementation with the container.

The code in the previous step receives a `CosmosClient` as part of the constructor. In order to tell ASP.NET Core pipeline how to create that object, edit the project's *Startup.cs* file and add the following line to the **ConfigureServices** handler:

```

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](#).

- Within the same file, 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;
    CosmosClientBuilder clientBuilder = new CosmosClientBuilder(account, key);
    CosmosClient client = clientBuilder
        .WithConnectionModeDirect()
        .Build();
    CosmosDbService cosmosDbService = new CosmosDbService(client, databaseName, containerName);
    DatabaseResponse database = await client.CreateDatabaseIfNotExistsAsync(databaseName);
    await database.Database.CreateContainerIfNotExistsAsync(containerName, "/id");

    return cosmosDbService;
}
```

- Define the configuration in the project's *appsettings.json* file. Open the file and add a section called **CosmosDb**:

```
"CosmosDb": {
    "Account": "<enter the URI from the Keys blade of the Azure Portal>",
    "Key": "<enter the PRIMARY KEY, or the SECONDARY KEY, from the Keys blade of the Azure Portal>",
    "DatabaseName": "Tasks",
    "ContainerName": "Items"
}
```

If you run the application, ASP.NET Core's pipeline instantiates **CosmosDbService** and maintain a single instance as singleton. When **ItemController** processes client-side requests, it receives this single instance and can use it for CRUD operations.

If you build and run this project now, you should now see something that looks like this:

List of To-Do Items		
Name	Description	Completed
<a href="#">Create New</a>		
© 2019 - CosmosWebSample		

If instead you see the home page of the application, append `/Item` to the url.

## Step 6: Run the application locally

To test the application on your local computer, use the following steps:

- Select 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:

## List of To-Do Items

Name	Description	Completed
<a href="#">Create New</a>		
© 2019 - CosmosWebSample		

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.

## List of To-Do Items

Name	Description	Completed	
Cosmos DB	Enjoy learning about Cosmos DB	<input type="checkbox"/>	<a href="#">Edit</a>   <a href="#">Details</a>   <a href="#">Delete</a>
<a href="#">Create New</a>			
© 2019 - CosmosWebSample			

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.

## Edit a To-Do Item

Item

Name

Description

Completed

[Save](#)

[Back to List](#)

© 2019 - CosmosWebSample

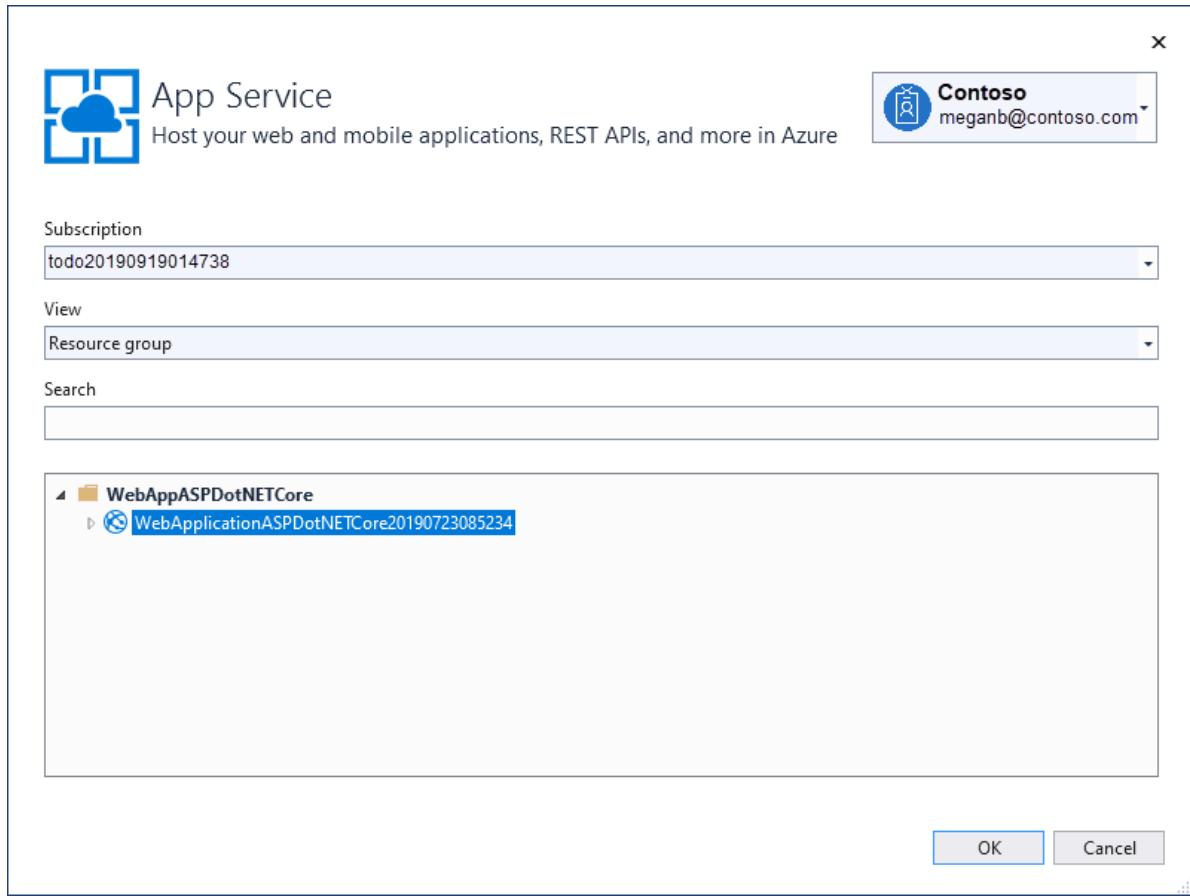
5. Verify the state of the data in the Azure Cosmos DB service using [Cosmos Explorer](#) or the Azure Cosmos DB Emulator's Data Explorer.
6. Once you've tested the app, select Ctrl+F5 to stop debugging the app. You're ready to deploy!

## Step 7: 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**.

The screenshot shows the 'Create new' dialog for an Azure App Service. On the left, there's a 'Name' field containing 'todo20190919011502'. Below it are 'Subscription' ('Contoso Subscription'), 'Resource group' ('ContosoRG32 (West US 2)'), 'Hosting Plan' ('ContosoWebAppPlan (West US 2, S1)'), and 'Application Insights' ('None'). On the right, there's a 'User' section for 'Contoso meganb@contoso.com'. Below that, 'Explore additional Azure services' links are shown for 'Create a storage account' and 'Create a SQL Database'. A note says 'Clicking the Create button will create the following Azure resources' followed by 'App Service - todo20190919011502'. At the bottom are 'Export...', 'Create' (highlighted in blue), and 'Cancel' buttons.

Name  
todo20190919011502

Subscription  
Contoso Subscription

Resource group  
ContosoRG32 (West US 2) New...

Hosting Plan  
ContosoWebAppPlan (West US 2, S1) New...

Application Insights i  
None

Contoso meganb@contoso.com

Explore additional Azure services

Create a storage account

Create a SQL Database

Clicking the Create button will create the following Azure resources

App Service - todo20190919011502

Export... Create Cancel

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

11/6/2019 • 18 minutes to read • [Edit Online](#)

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.

Name	Category	Complete
Build a Java application using the SQL API	Cosmos DB	<input checked="" type="checkbox"/>

## 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, you can use the [Azure Cosmos DB Emulator](#) with a URI of `https://localhost:8081`. For the key to use with the emulator, see [Authenticating requests](#).

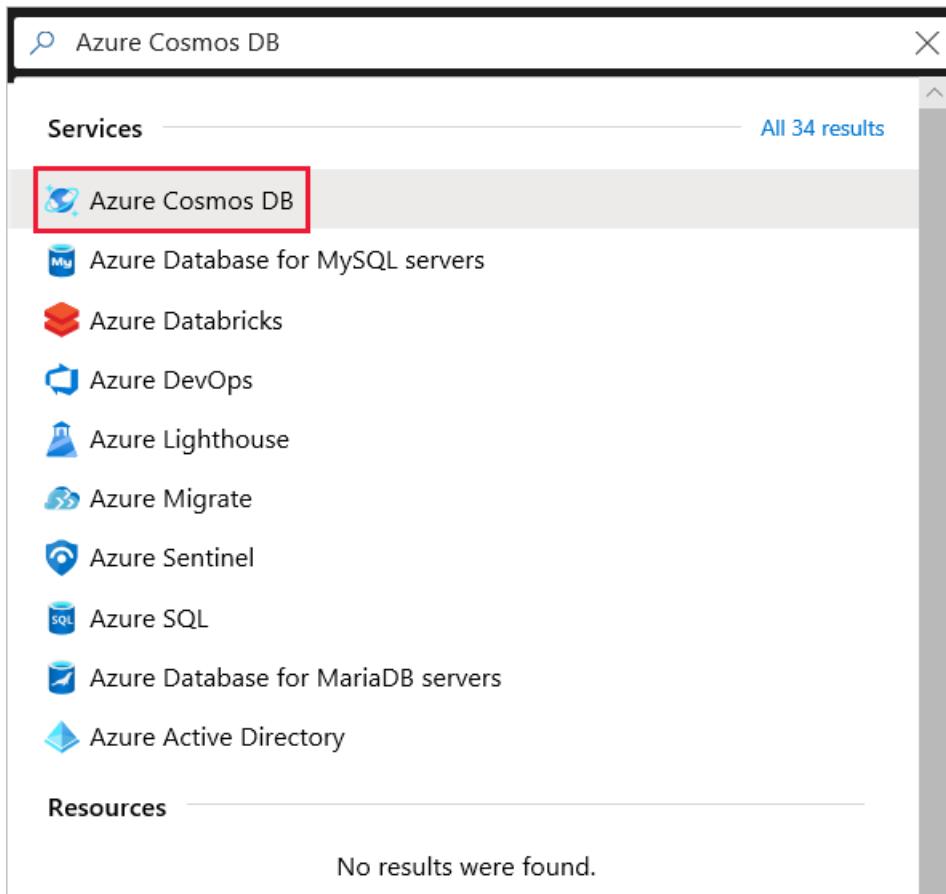
- [Java Development Kit \(JDK\) 7+](#).
- [Eclipse IDE for Java EE Developers](#).
- [An Azure Web Site with a Java runtime environment \(e.g. Tomcat or Jetty\) enabled](#).

If you're installing these tools for the first time, coreservlets.com provides a walk-through of the installation process in the Quick Start section of their [Tutorial: Installing TomCat7 and Using it with Eclipse](#) article.

## Step 1: 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. Go to the [Azure portal](#) to create an Azure Cosmos DB account. Search for and select **Azure Cosmos DB**.



2. Select **Add**.
3. 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.

Setting	Value	Description
Resource Group	Resource group name	Select a resource group, or select <b>Create new</b> , then enter a unique name for the new resource group.
Account Name	A unique name	<p>Enter a name to identify your Azure Cosmos account. Because <i>documents.azure.com</i> is appended to the name that you provide to create your URL, use a unique name.</p> <p>The name can only contain lowercase letters, numbers, and the hyphen (-) character. It must be between 3-31 characters in length.</p>
API	The type of account to create	<p>Select <b>Core (SQL)</b> to create a document database and query by using SQL syntax.</p> <p>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.</p> <p><a href="#">Learn more about the SQL API.</a></p>
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.

## Create Azure Cosmos DB Account

[Basics](#) [Network](#) [Tags](#) [Review + create](#)

Azure Cosmos DB is a fully managed globally distributed, multi-model database service, transparently replicating your data across any number of Azure regions. You can elastically scale throughput and storage, and take advantage of fast, single-digit-millisecond data access using the API of your choice backed by 99.999 SLA. [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  * Resource Group	Contoso Subscription  (New) myResourceGroup <a href="#">Create new</a>
--	---

### INSTANCE DETAILS

* Account Name 	mysqlapicosmosdb <span style="float: right;">documents.azure.com</span>
* API 	Core (SQL)
* Location 	West US
Geo-Redundancy 	<input type="button" value="Enable"/> <input type="button" value="Disable"/>
Multi-region Writes 	<input type="button" value="Enable"/> <input type="button" value="Disable"/>

[Review + create](#)

[Previous](#)

[Next: Network](#)

4. Select **Review + create**. You can skip the **Network** and **Tags** sections.

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**.

Dashboard > Microsoft.Azure.CosmosDB-2019032100000 - Overview

**Microsoft.Azure.CosmosDB-2019032100000 - Overview**

Deployment

 Delete  Cancel  Redeploy  Refresh

 Overview  Inputs  Outputs  Template

**✓ Your deployment is complete**

[Go to resource](#)

 Deployment name: Microsoft.Azure.CosmosDB-2019032100000  
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	<a href="#">Operation details</a>

6. Select **Go to resource** to go to the Azure Cosmos DB account page.

Congratulations! Your Azure Cosmos DB account was created.

Now, let's connect to it using a sample app:

**Choose a platform**

- .NET
- .NET Core
- Xamarin
- Java
- Node.js
- Python

- 1 Add a collection**  
In Azure Cosmos DB, data is stored in collections.  
**Create 'Items' collection**
- 2 Download and run your .NET app**  
Once collection is created, download a sample .NET app connected to it, extract, build and run.  
**Download**

Go to the Azure Cosmos DB account page, and select **Keys**. Copy the values to use in the web application you create next.

URI
https://contoso-demo.documents.azure.com:443/

PRIMARY KEY
MECyqFOL6eGkv73YUuZoQY8ej4ZpQJ2frW30YgP2GNLtyaaNoHoqRYbOPNdRLQjOwNx5oxe3sPEfn0PQ2T0V7==

SECONDARY KEY
A1qltpGX6GyekSpIjcYfNV0RAO91eCaMfNLi1Ge60b6GtYszQYBU5pWZvXzuWvJB1X4T0e0sfXYNuWe95pv==

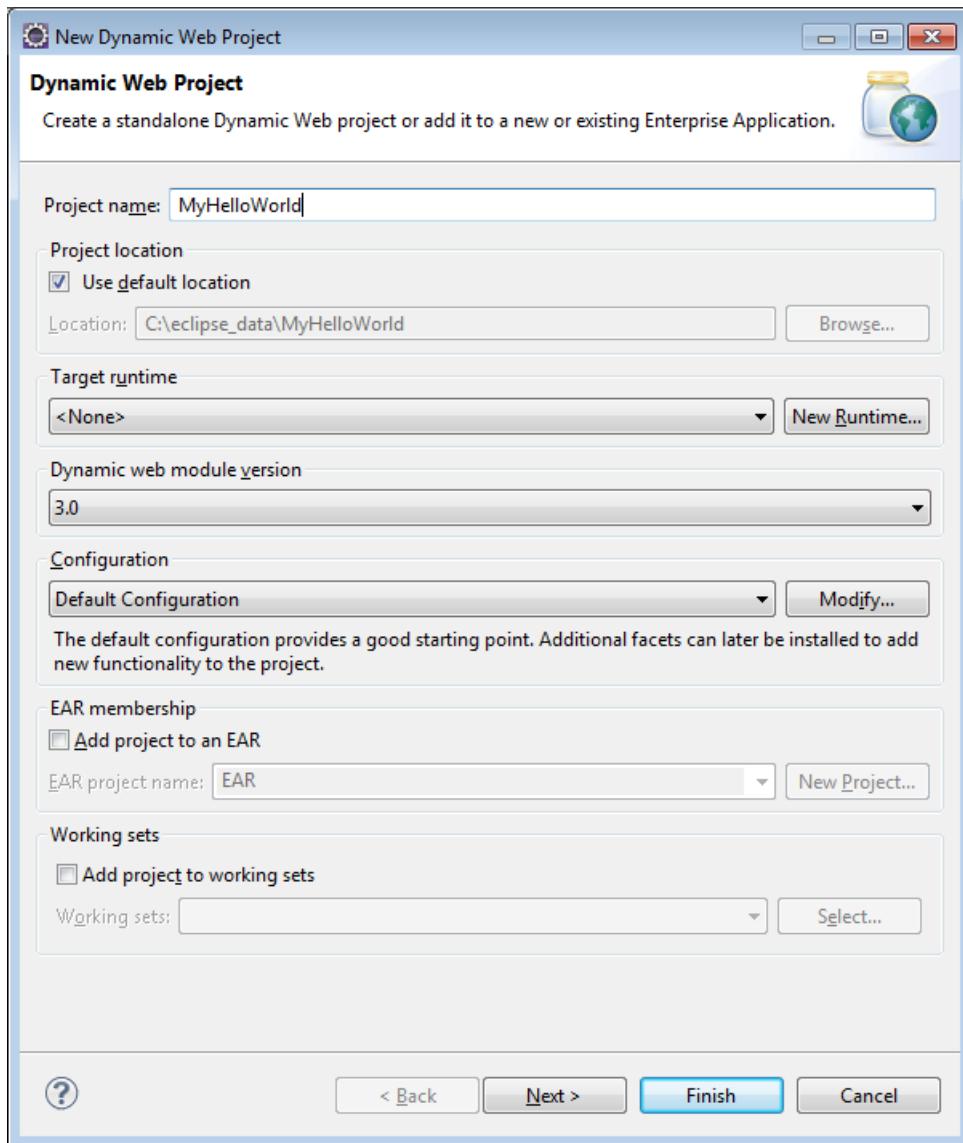
PRIMARY CONNECTION STRING
AccountEndpoint=https://contoso-demo.documents.azure.com:443/;AccountKey=WFA5AYX1OLxpDrWZJ0OrjOeJmWLhN4pVuXS5QlJSQX7...

SECONDARY CONNECTION STRING
AccountEndpoint=https://contoso-demo.documents.azure.com:443/;AccountKey=1ShVfjwnHL8VGBrIkFTkeTS3AERH5cgDDjGLqDxD4Dkay...

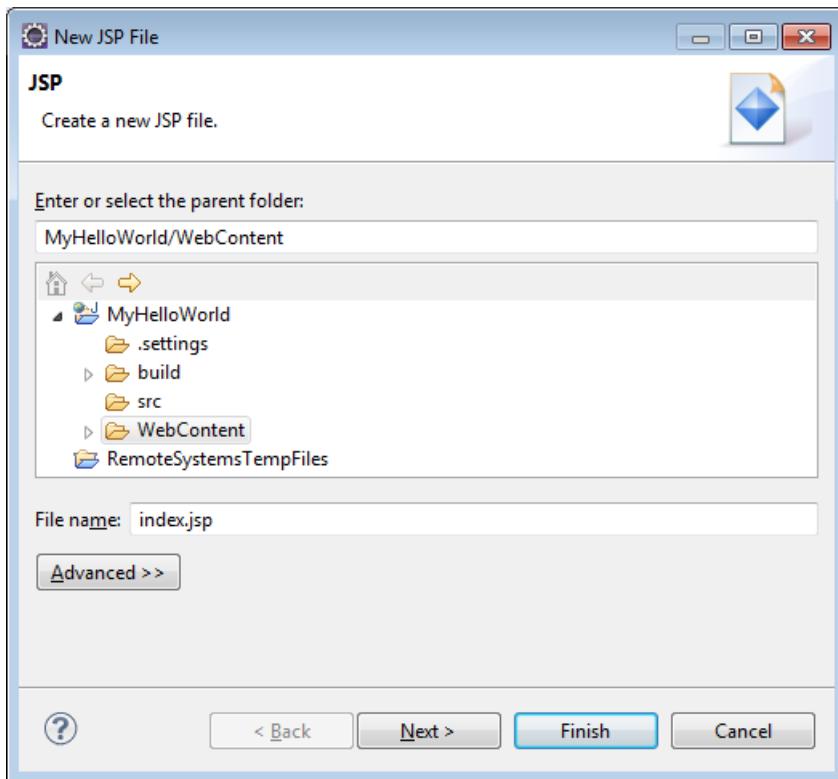
## Step 2: Create the Java JSP application

To create the JSP application:

- 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**.



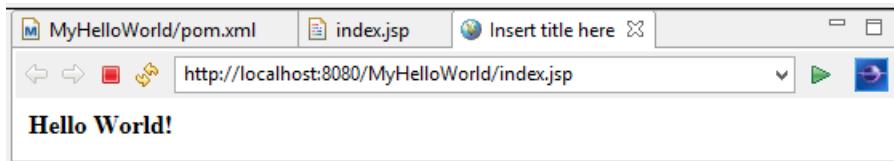
2. Enter a project name in the **Project name** box, and in the **Target Runtime** drop-down menu, optionally select a value (e.g. Apache Tomcat v7.0), and then click **Finish**. Selecting a target runtime enables you to run your project locally through Eclipse.
3. In Eclipse, in the Project Explorer 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:



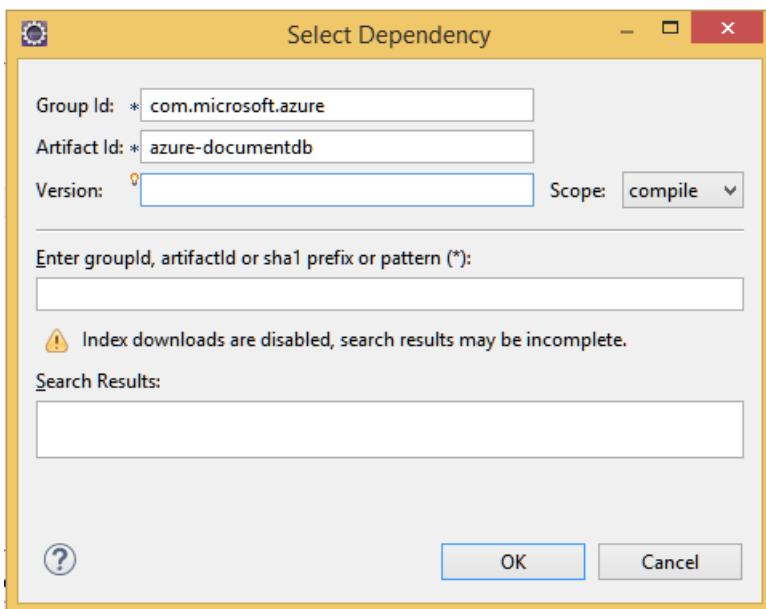
## Step 3: Install the SQL Java SDK

The easiest way to pull in the SQL Java SDK and its dependencies is through [Apache Maven](#).

To do this, you will need to convert your project to a maven project by completing 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.microsoft.azure.

- In the **Artifact Id** box, enter azure-documentdb.
- In the **Version** box, enter 1.5.1.



- Or add the dependency XML for Group Id and Artifact Id directly to the pom.xml via a text editor:

```
<dependency>
    <groupId>com.microsoft.azure</groupId>
    <artifactId>azure-documentdb</artifactId>
    <version>1.9.1</version>
</dependency>
```

6. Click **OK** and Maven will install the SQL Java SDK.

7. Save the pom.xml file.

## Step 4: Using the Azure Cosmos DB service in a Java application

1. First, let's define the TodoItem object in TodoItem.java:

```
@Data
@Builder
public class TodoItem {
    private String category;
    private boolean complete;
    private String id;
    private String name;
}
```

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.

2. To invoke the Azure Cosmos DB service, you must instantiate a new **DocumentClient**. In general, it is best to reuse the **DocumentClient** - rather than construct a new client for each subsequent request. We can reuse the client by wrapping the client in a **DocumentClientFactory**. In DocumentClientFactory.java, you need to paste the URI and PRIMARY KEY value you saved to your clipboard in [step 1](#). Replace [YOUR\_ENDPOINT\_HERE] with your URI and replace [YOUR\_KEY\_HERE] with your PRIMARY KEY.

```

private static final String HOST = "[YOUR_ENDPOINT_HERE]";
private static final String MASTER_KEY = "[YOUR_KEY_HERE]";

private static DocumentClient documentClient = new DocumentClient(HOST, MASTER_KEY,
    ConnectionPolicy.GetDefault(), ConsistencyLevel.Session);

public static DocumentClient getDocumentClient() {
    return documentClient;
}

```

3. Now let's create a Data Access Object (DAO) to abstract persisting our 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.

The following code illustrates how to retrieve our database and collection, if it exists, or create a new one if it doesn't exist:

```

public class DocDbDao implements TodoDao {
    // The name of our database.
    private static final String DATABASE_ID = "TodoDB";

    // The name of our collection.
    private static final String COLLECTION_ID = "TodoCollection";

    // The Azure Cosmos DB Client
    private static DocumentClient documentClient = DocumentClientFactory
        .getDocumentClient();

    // Cache for the database object, so we don't have to query for it to
    // retrieve self links.
    private static Database databaseCache;

    // Cache for the collection object, so we don't have to query for it to
    // retrieve self links.
    private static DocumentCollection collectionCache;

    private Database getTodoDatabase() {
        if (databaseCache == null) {
            // Get the database if it exists
            List<Database> databaseList = documentClient
                .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 {
                    Database databaseDefinition = new Database();
                    databaseDefinition.setId(DATABASE_ID);

                    databaseCache = documentClient.createDatabase(
                        databaseDefinition, null).getResource();
                } catch (DocumentClientException 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 DocumentCollection getTodoCollection() {
    if (collectionCache == null) {
        // Get the collection if it exists.
        List<DocumentCollection> collectionList = documentClient
            .queryCollections(
                getTodoDatabase().getSelfLink(),
                "SELECT * FROM root r WHERE r.id='" + COLLECTION_ID
                + "'", null).getQueryIterable().toList();

        if (collectionList.size() > 0) {
            // Cache the collection object so we won't have to query for it
            // later to retrieve the selfLink.
            collectionCache = collectionList.get(0);
        } else {
            // Create the collection if it doesn't exist.
            try {
                DocumentCollection collectionDefinition = new DocumentCollection();
                collectionDefinition.setId(COLLECTION_ID);

                collectionCache = documentClient.createCollection(
                    getTodoDatabase().getSelfLink(),
                    collectionDefinition, null).getResource();
            } catch (DocumentClientException 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 collectionCache;
}
}

```

4. The next step is to write some code to persist the TodoItems into the collection. In this example, we will use [Gson](#) to serialize and de-serialize TodoItem Plain Old Java Objects (POJOs) to JSON documents.

```

// We'll use Gson for POJO <=> JSON serialization for this example.
private static Gson gson = new Gson();

@Override
public TodoItem createTodoItem(TodoItem todoItem) {
    // Serialize the TodoItem as a JSON Document.
    Document todoItemDocument = new Document(gson.toJson(todoItem));

    // Annotate the document as a TodoItem for retrieval (so that we can
    // store multiple entity types in the collection).
    todoItemDocument.set("entityType", "todoItem");

    try {
        // Persist the document using the DocumentClient.
        todoItemDocument = documentClient.createDocument(
            getTodoCollection().getSelfLink(), todoItemDocument, null,
            false).getResource();
    } catch (DocumentClientException e) {
        e.printStackTrace();
        return null;
    }

    return gson.fromJson(todoItemDocument.toString(), TodoItem.class);
}

```

- Like Azure Cosmos databases and collections, documents are also referenced by self-links. The following helper function lets us retrieve documents by another attribute (e.g. "id") rather than self-link:

```

private Document getDocumentById(String id) {
    // Retrieve the document using the DocumentClient.
    List<Document> documentList = documentClient
        .queryDocuments(getTodoCollection().getSelfLink(),
            "SELECT * FROM root r WHERE r.id='"
                + id + "'", null)
        .getQueryIterable().toList();

    if (documentList.size() > 0) {
        return documentList.get(0);
    } else {
        return null;
    }
}

```

- We can use the helper method in step 5 to retrieve a TodoItem JSON document by id and then deserialize it to a POJO:

```

@Override
public TodoItem readTodoItem(String id) {
    // Retrieve the document by id using our helper method.
    Document todoItemDocument = getDocumentById(id);

    if (todoItemDocument != null) {
        // De-serialize the document in to a TodoItem.
        return gson.fromJson(todoItemDocument.toString(), TodoItem.class);
    } else {
        return null;
    }
}

```

- We can also use the DocumentClient to get a collection or list of TodoItems using SQL:

```

@Override
public List<TodoItem> readTodoItems() {
    List<TodoItem> todoItems = new ArrayList<TodoItem>();

    // Retrieve the TodoItem documents
    List<Document> documentList = documentClient
        .queryDocuments(getTodoCollection().getSelfLink(),
            "SELECT * FROM root r WHERE r.entityType = 'todoItem'",
            null).getQueryIterable().toList();

    // De-serialize the documents in to TodoItems.
    for (Document todoItemDocument : documentList) {
        todoItems.add(gson.fromJson(todoItemDocument.toString(),
            TodoItem.class));
    }

    return todoItems;
}

```

8. There are many ways to update a document with the DocumentClient. In our Todo list application, we want to be able to toggle whether a TodoItem is complete. This can be achieved by updating the "complete" attribute within the document:

```

@Override
public TodoItem updateTodoItem(String id, boolean isComplete) {
    // Retrieve the document from the database
    Document todoItemDocument = 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.
    todoItemDocument.set("complete", isComplete);

    try {
        // Persist/replace the updated document.
        todoItemDocument = documentClient.replaceDocument(todoItemDocument,
            null).getResource();
    } catch (DocumentClientException e) {
        e.printStackTrace();
        return null;
    }

    return gson.fromJson(todoItemDocument.toString(), TodoItem.class);
}

```

9. Finally, we want the ability to delete a TodoItem from our list. To do this, we can use the helper method we wrote earlier to retrieve the self-link and then tell the client to delete it:

```
@Override
public boolean deleteTodoItem(String id) {
    // Azure Cosmos DB refers to documents by self link rather than id.

    // Query for the document to retrieve the self link.
    Document todoItemDocument = getDocumentById(id);

    try {
        // Delete the document by self link.
        documentClient.deleteDocument(todoItemDocument.getSelfLink(), null);
    } catch (DocumentClientException e) {
        e.printStackTrace();
        return false;
    }

    return true;
}
```

## Step 5: Wiring the rest of the Java application development project 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 our DAO.

1. First, let's start with building a controller to call our DAO:

```

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 = TodoItem.builder().name(name).category(category)
            .complete(isComplete).build();
        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);
    }
}

```

In a more complex application, the controller may house complicated business logic on top of the DAO.

2. Next, we'll create a servlet to route HTTP requests to the controller:

```

public class TodoServlet 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.getWriter().println(apiResponse);
    }

    @Override
    protected void doPost(HttpServletRequest request,
                         HttpServletResponse response) throws ServletException, IOException {
        doGet(request, response);
    }
}

```

3. We'll need a web user interface to display to the user. Let's re-write the index.jsp we created earlier:

```

<html>
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">

```

```

<meta http-equiv="X-UA-Compatible" content="IE=edge;" />
<title>Azure Cosmos DB 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>
        </div>

        <!-- Update Button -->
        <div class="todoUpdatePanel">
            <form class="form-horizontal" role="form">
                <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>
            </form>
        </div>
    </div>

```

```

        </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 --&gt;
&lt;script src="//ajax.aspnetcdn.com/ajax/jquery/jquery-2.1.1.min.js"&gt;&lt;/script&gt;
&lt;script src="//ajax.aspnetcdn.com/ajax/bootstrap/3.2.0/bootstrap.min.js"&gt;&lt;/script&gt;
&lt;script src="assets/todo.js"&gt;&lt;/script&gt;
&lt;/body&gt;
&lt;/html&gt;
</pre>

```

4. And finally, write some client-side JavaScript to tie the web user interface and the servlet together:

```

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"
},
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.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');

        // Toggle 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();
});

```

5. Awesome! 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**.
6. Once the item appears, you can update whether it's complete by toggling the checkbox and clicking **Update Tasks**.

## Step 6: 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.

- To export your application as a WAR file, right-click on your project in **Project Explorer**, click **Export**, and then click **WAR File**.
- 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**.
- 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](#).
- Once the WAR file is uploaded to the webapps directory, the runtime environment will detect that you've added it and will automatically load it.
- 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:

- 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.
- If Eclipse is open, close it and restart it to load Lombok.
- In Eclipse, on the **File** menu, click **Import**.
- In the **Import** window, click **Git**, click **Projects from Git**, and then click **Next**.
- On the **Select Repository Source** screen, click **Clone URI**.
- 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**.
- On the **Branch Selection** screen, ensure that **master** is selected, and then click **Next**.
- On the **Local Destination** screen, click **Browse** to select a folder where the repository can be copied, and then click **Next**.
- On the **Select a wizard to use for importing projects** screen, ensure that **Import existing projects** is selected, and then click **Next**.
- 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.
- 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](#).
- In **Project Explorer**, right-click the **azure-documentdb-java-sample**, click **Build Path**, and then click **Configure Build Path**.
- 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**.
- Use step 12 to open the **Properties** window again, and then in the left pane click **Targeted Runtimes**.
- On the **Targeted Runtimes** screen, click **New**, select **Apache Tomcat v7.0**, and then click **OK**.
- Use step 12 to open the **Properties** window again, and then in the left pane click **Project Facets**.
- On the **Project Facets** screen, select **Dynamic Web Module** and **Java**, and then click **OK**.
- 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](#).

# Tutorial: Build a Node.js web app using the JavaScript SDK to manage a SQL API account in Azure Cosmos DB

12/13/2019 • 12 minutes to read • [Edit Online](#)

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, you can 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. Go to the [Azure portal](#) to create an Azure Cosmos DB account. Search for and select **Azure Cosmos DB**.

Azure Cosmos DB

Services

- Azure Cosmos DB**
- Azure Database for MySQL servers
- Azure Databricks
- Azure DevOps
- Azure Lighthouse
- Azure Migrate
- Azure Sentinel
- Azure SQL
- Azure Database for MariaDB servers
- Azure Active Directory

Resources

No results were found.

2. Select **Add**.
3. 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 <b>Create new</b> , then enter a unique name for the new resource group.
Account Name	A unique name	<p>Enter a name to identify your Azure Cosmos account. Because <i>documents.azure.com</i> is appended to the name that you provide to create your URI, use a unique name.</p> <p>The name can only contain lowercase letters, numbers, and the hyphen (-) character. It must be between 3-31 characters in length.</p>

SETTING	VALUE	DESCRIPTION
API	The type of account to create	<p>Select <b>Core (SQL)</b> to create a document database and query by using SQL syntax.</p> <p>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.</p> <p><a href="#">Learn more about the SQL API.</a></p>
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.

Dashboard > New > Create Azure Cosmos DB Account

## Create Azure Cosmos DB Account

[Basics](#) [Network](#) [Tags](#) [Review + create](#)

Azure Cosmos DB is a fully managed globally distributed, multi-model database service, transparently replicating your data across any number of Azure regions. You can elastically scale throughput and storage, and take advantage of fast, single-digit-millisecond data access using the API of your choice backed by 99.999 SLA. [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	Contoso Subscription
└─ * Resource Group	(New) myResourceGroup
	<a href="#">Create new</a>

**INSTANCE DETAILS**

* Account Name	mysqlapicosmosdb
	documents.azure.com
* API ⓘ	Core (SQL)
* Location	West US
Geo-Redundancy ⓘ	<a href="#">Enable</a> <a href="#">Disable</a>
Multi-region Writes ⓘ	<a href="#">Enable</a> <a href="#">Disable</a>

[Review + create](#) [Previous](#) [Next: Network](#)

4. Select **Review + create**. You can skip the **Network** and **Tags** sections.
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**.

The screenshot shows the 'Deployment' section of the Azure Cosmos DB overview. It displays a success message: 'Your deployment is complete'. Below this, it shows deployment details: name (Microsoft.Azure.CosmosDB-20190321000000), subscription (Contoso Subscription), and resource group (myResourceGroup). Deployment details include start time (3/21/2019, 5:00:03 PM), duration (5 minutes 38 seconds), and correlation ID (8e0be948-0c60-4da0-0000-000000000000). A table lists the deployed resource ('mysqlapicosmosdb') as Microsoft.DocumentDb/databaseAcc... with status OK and operation details.

6. Select **Go to resource** to go to the Azure Cosmos DB account page.

The screenshot shows the 'Quick start' section of the Azure Cosmos DB account page. It congratulates the user on creating the account and provides instructions to connect using a sample app. It offers to choose a platform (.NET, .NET Core, Xamarin, Java, Node.js, Python) and provides steps to add a collection and download/run a .NET app. The 'Quick start' option is highlighted in the left sidebar.

Go to the Azure Cosmos DB account page, and select **Keys**. Copy the values to use in the web application you create next.

The screenshot shows the 'Keys' section of the Azure Cosmos DB account page. It displays two tabs: 'Read-write Keys' and 'Read-only Keys'. The 'Read-write Keys' tab is selected, showing the primary key and secondary key. Both fields are highlighted with red boxes. Below these are the primary connection string and secondary connection string, each also highlighted with red boxes. The 'Keys' option is highlighted in the left sidebar.

## 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
```

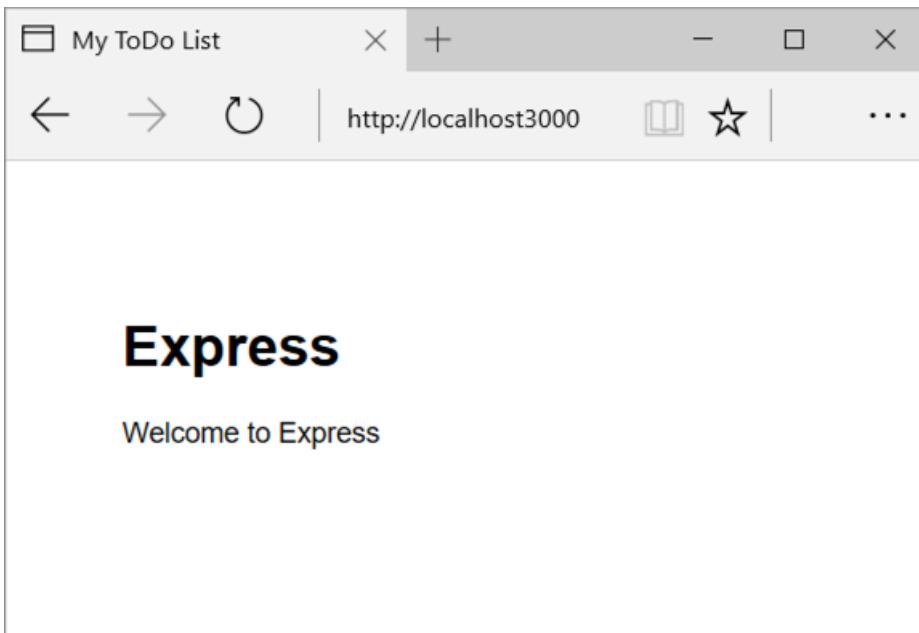
4. Open the **todo** directory and install dependencies.

```
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
      .replaceItem(itemId, doc)
    return replaced
  }
}
```

```
        .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 setting 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.

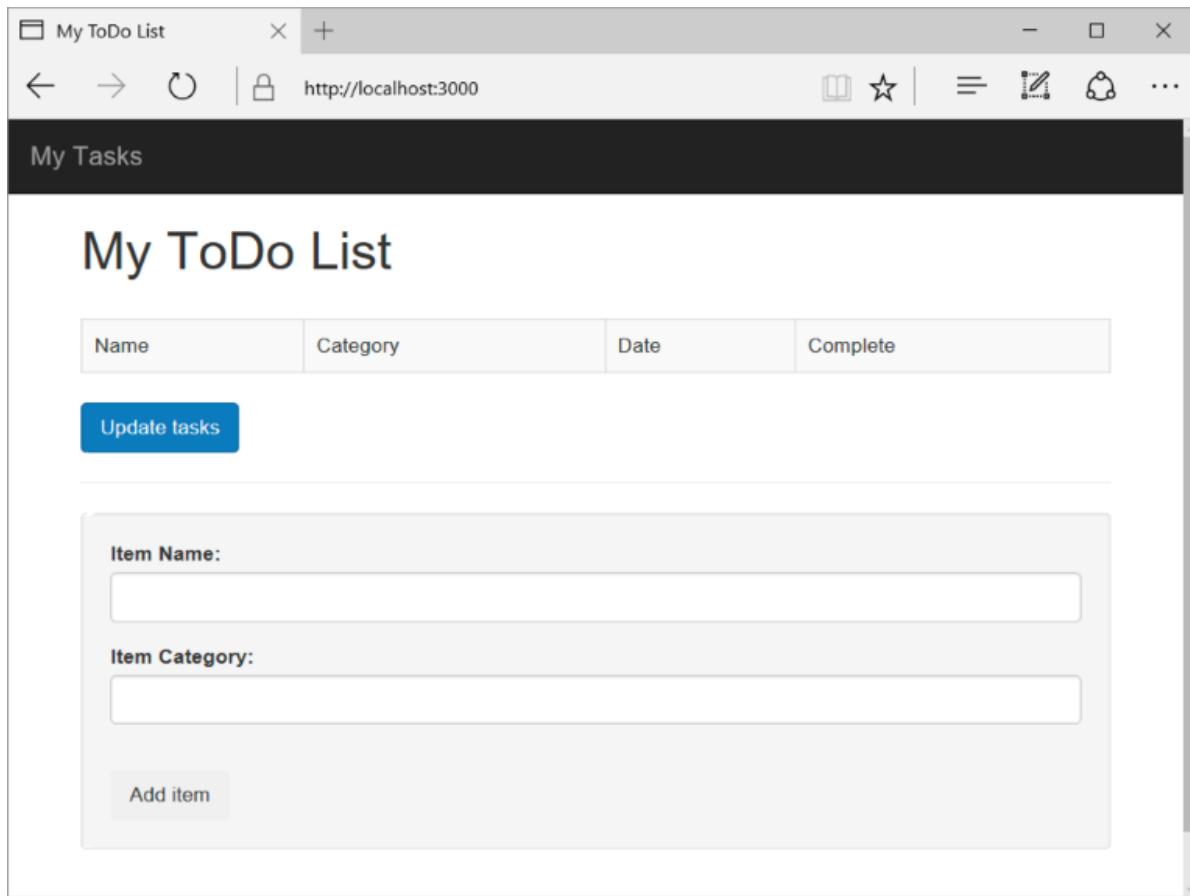
The first form contains a table for your data and a button that allows you to update items by posting to **/completeTask** method of the controller.

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.

Name	Category	Date	Complete
A Thing	An Item	6/28/2017	<input type="checkbox"/>

**Update tasks**

**Item Name:**

**Item Category:**

Add item

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

11/6/2019 • 5 minutes to read • [Edit Online](#)

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 needs. 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:



Low latency



Rich query



Limitless scale



Replicate globally



Geospatial



Binary attachments



- 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.

The screenshot shows the Azure portal interface for a 'my-xamarin-app' Azure Cosmos DB account. The left sidebar contains navigation links like Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Quick start (which is selected and highlighted in blue), and Data Explorer (Preview). The main content area has a heading 'Congratulations! Your Azure Cosmos DB account was created.' Below it, a section titled 'Choose a platform' shows icons for .NET, .NET Core, Xamarin (selected), Java, Node.js, and Python. Three large numbered steps are displayed: 1. Add a collection, which describes collections as storage units and includes a 'Create 'Items' collection' button; 2. Download and run your Xamarin app, which includes a 'Download' button; 3. Learn More, which lists links for Code Samples, Documentation, Pricing, Capacity Planner, and Forum.

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>());
}
```

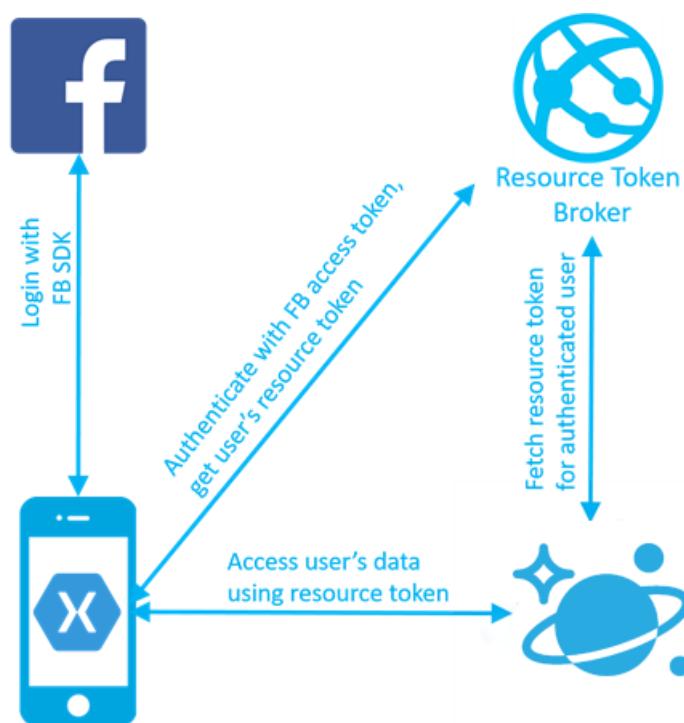
## Add users

Like many get started samples, the Azure Cosmos DB sample you downloaded authenticates to the service by using a master 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 master 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 1 RU. You can also add alerts to the **Throughput** metric to monitor the traffic growth and programmatically change the throughput as alerts fire.

my-xamarin-app - Scale  
Azure Cosmos DB account

Search (Ctrl+ /)

Overview

Activity log

Access control (IAM)

Tags

Diagnose and solve problems

Quick start

Data Explorer (Preview)

Replicate data globally

Default consistency

Firewall

Keys

Add Azure Search

Locks

Automation script

Items

PRICING TIER ⓘ

Standard

\* THROUGHPUT (RU/s) ⓘ

10000

Between 400 and 10000 RU/s

RU/m ⓘ

ON OFF

DEFAULT STORAGE CAPACITY ⓘ

10 GB

ESTIMATED HOURLY COST

THROUGHPUT

0.80 USD

STORAGE

0 USD

Save Discard

## 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.

my-xamarin-app - Replicate data globally  
Azure Cosmos DB account

Search (Ctrl+ /)

Save Discard Manual Failover Automatic Failover

Overview Activity log Access control (IAM) Tags Diagnose and solve problems Quick start Data Explorer (Preview)

SETTINGS

Replicate data globally (selected)

Default consistency Firewall Keys Add Azure Search Locks Automation script

COLLECTIONS

Click on a location to add or remove regions from your Azure Cosmos DB account.  
\* Each region is billable based on the throughput and storage for the account. [Learn more](#)



WRITER REGION

West US

READ REGIONS

The account has no read regions.

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

11/6/2019 • 24 minutes to read • [Edit Online](#)

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.

Which API are you going to use with Azure Cosmos DB?

- [SQL API](#) - You can use any of the source options provided in the Data Migration tool to import data.
- [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 currently 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.
- [Gremlin API](#) - The Data Migration tool isn't a supported import tool for Gremlin API accounts at this time.

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 4.5.1](#) 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.

## 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.

## 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

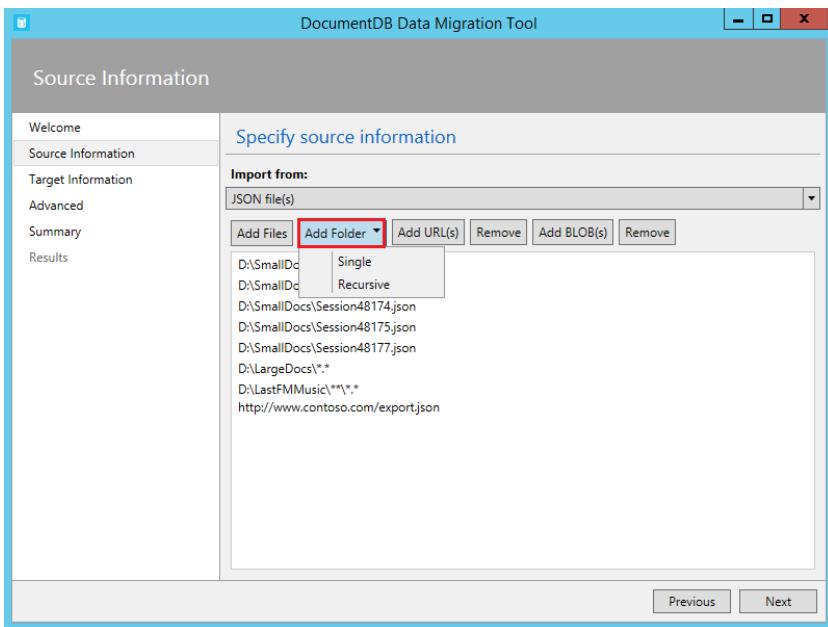
## Select data source

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=wJmFRYna6ttQ79ATmrTMq18vPr184QBiHTt6oinFkZRvoe7Vv81x9sn6z1V1BY10bEPMgGM982wfYXpWXWB9w==;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.json;C:\TESessions\Session48177.json /t:DocumentDBBulk /t.ConnectionString:"AccountEndpoint=<CosmosDB Endpoint>;AccountKey=<CosmosDB Key>;Database=<CosmosDB Database>;" /t.Collection:subs /t.CollectionThroughput:2500

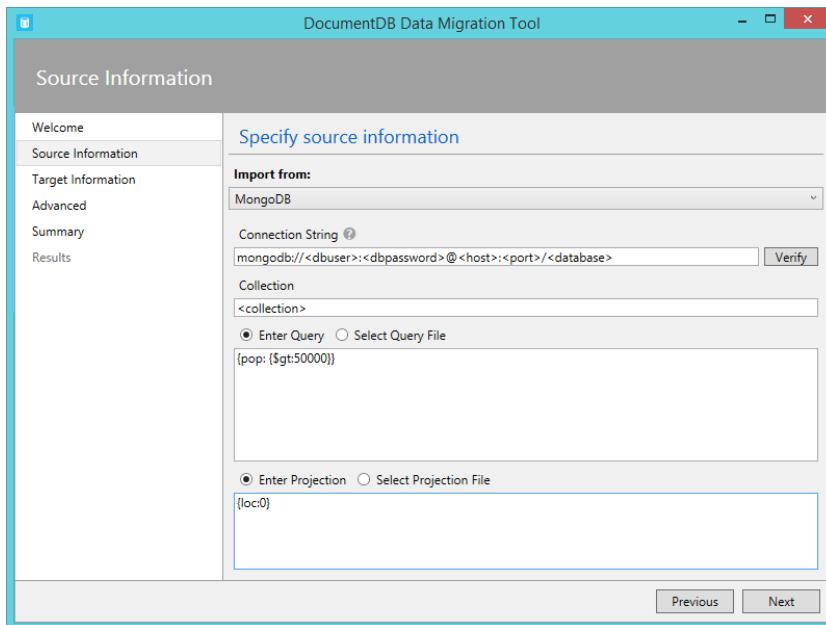
#Import a single JSON file and partition the data across 4 collections
dt.exe /s:JsonFile /s:Files:D:\CompanyData\Companies.json /t:DocumentDBBulk /t.ConnectionString:"AccountEndpoint=<CosmosDB Endpoint>;AccountKey=<CosmosDB Key>;Database=<CosmosDB Database>;" /t.Collection:comp[1-4] /t.PartitionKey:name /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

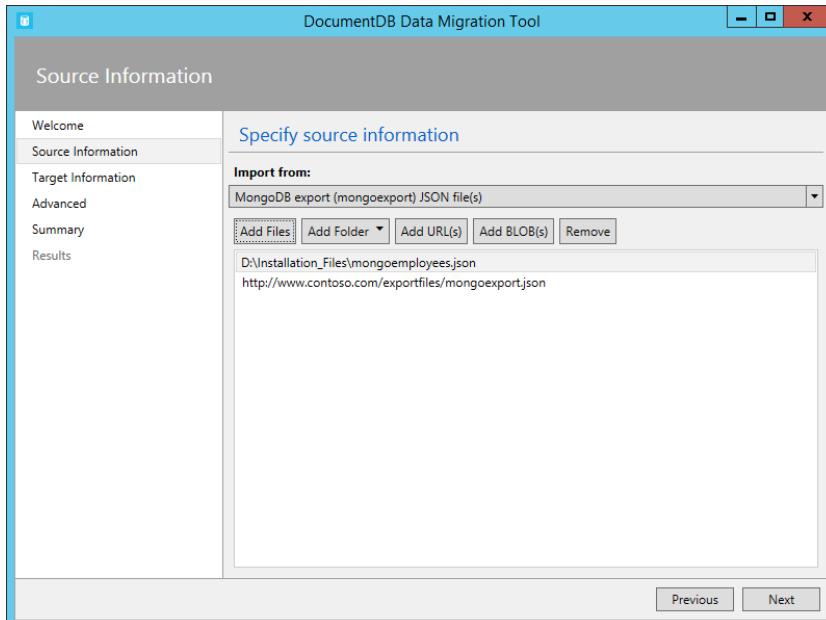
#Import documents from a MongoDB collection which match the query and exclude the loc field
dt.exe /s:MongoDB /s.ConnectionString:mongodb://<dbuser>:<dbpassword>@<host>:<port>/<database> /s.Collection:zips /s.Query:{pop:{$gt:5000}} /s.Projection:{loc:0}
/t:DocumentDBBulk /t.ConnectionString:"AccountEndpoint=<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.



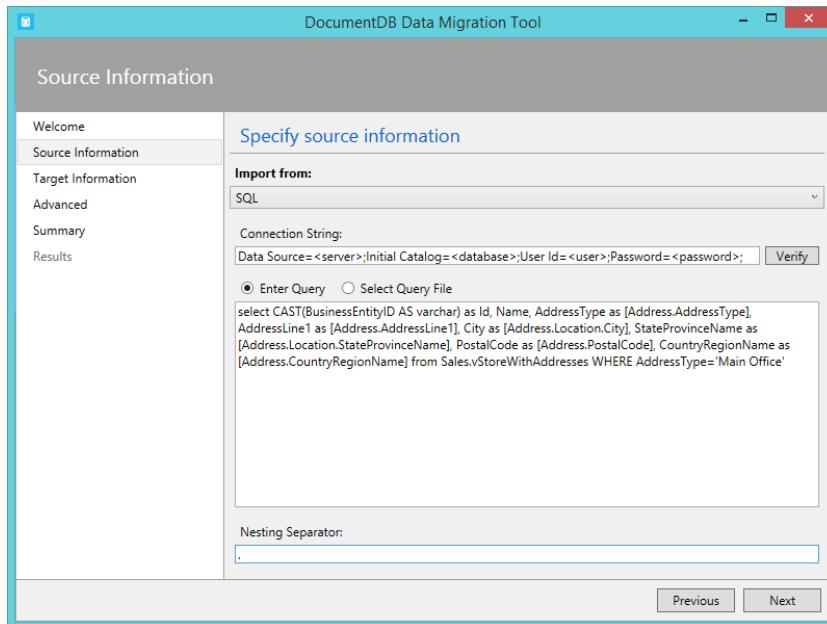
When adding folders that have MongoDB export JSON files for import, you have the option of recursively searching for files in subfolders.

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	Closetout 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		Venieres 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" }}
```

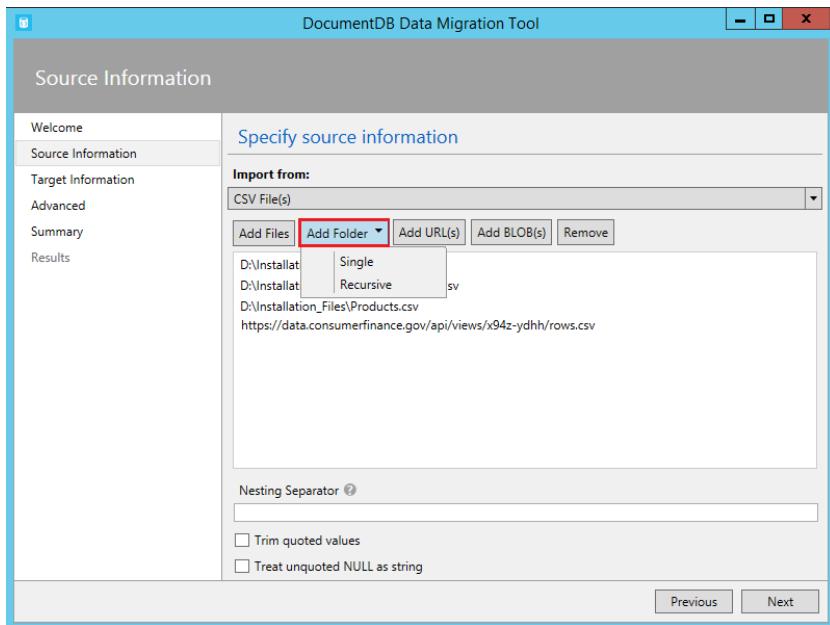
Here are some command-line samples to import from SQL Server:

```
#Import records from SQL which match a query
dt.exe /s:SQL /s.ConnectionString:"Data Source=<server>;Initial Catalog=AdventureWorks;User Id=adwworks;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=adwworks;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.

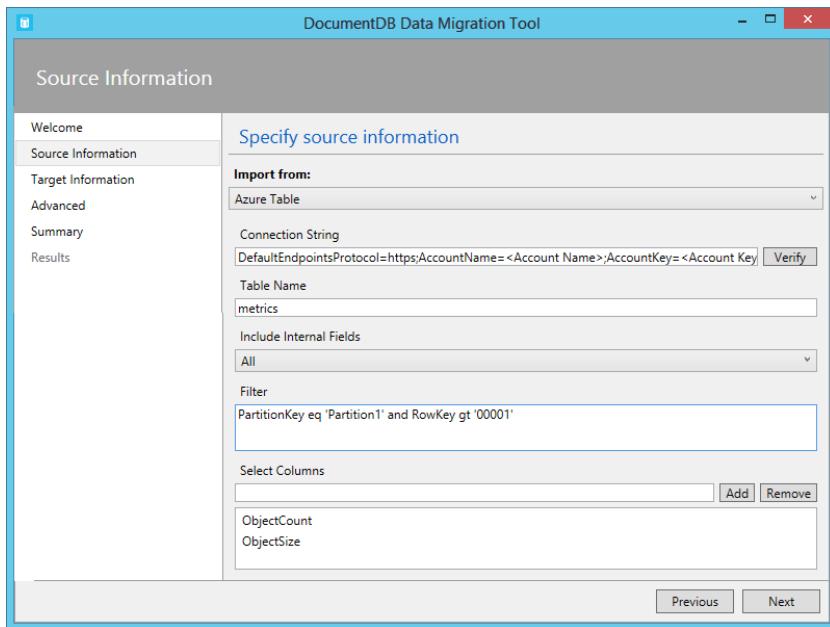
Here is a command-line sample for CSV import:

```
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:

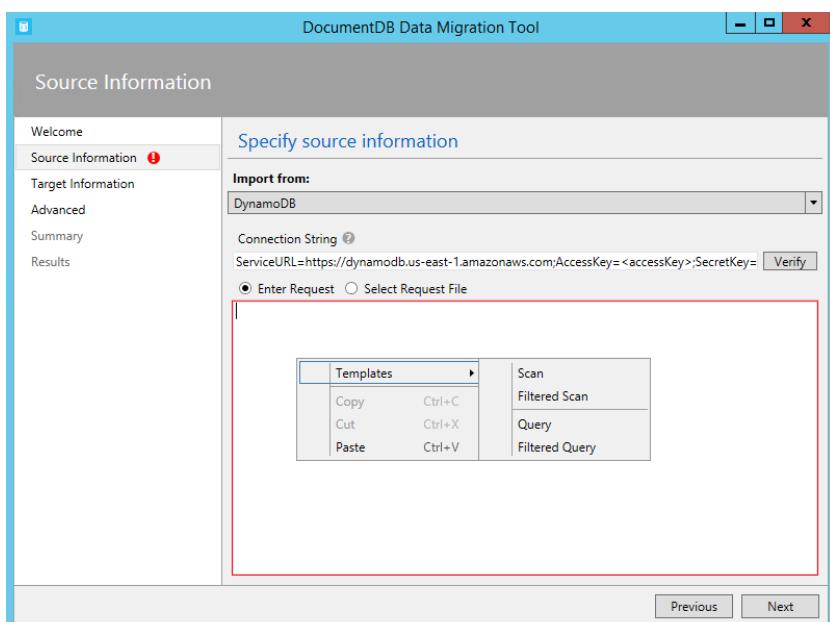
1. Include Internal Fields
  - a. All - Include all internal fields (PartitionKey, RowKey, and Timestamp)
  - b. None - Exclude all internal fields
  - c. RowKey - Only include the RowKey field
2. Select Columns
  - a. Azure Table storage filters don't support projections. If you want to only import specific Azure Table entity properties, add them to the Select Columns list. All other entity properties are ignored.

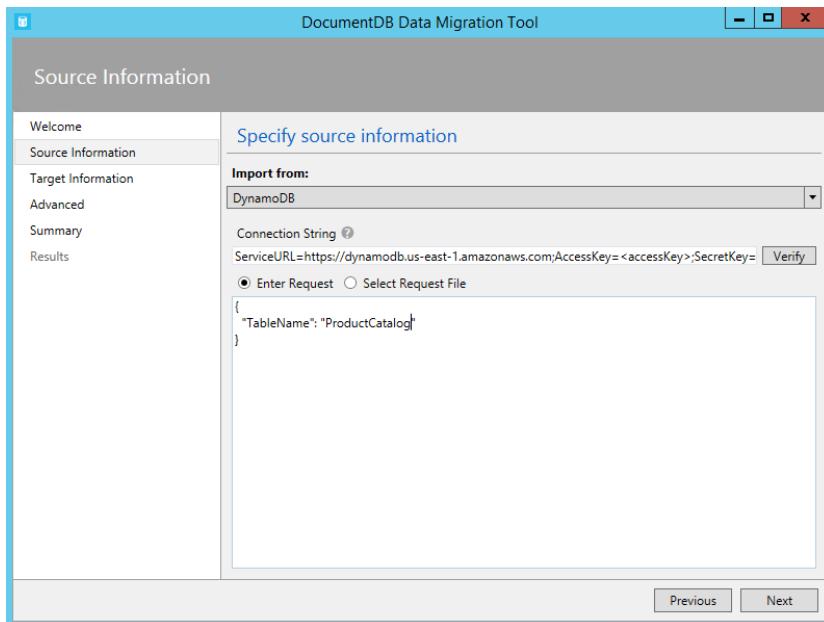
Here is a command-line sample to import from Azure Table storage:

```
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

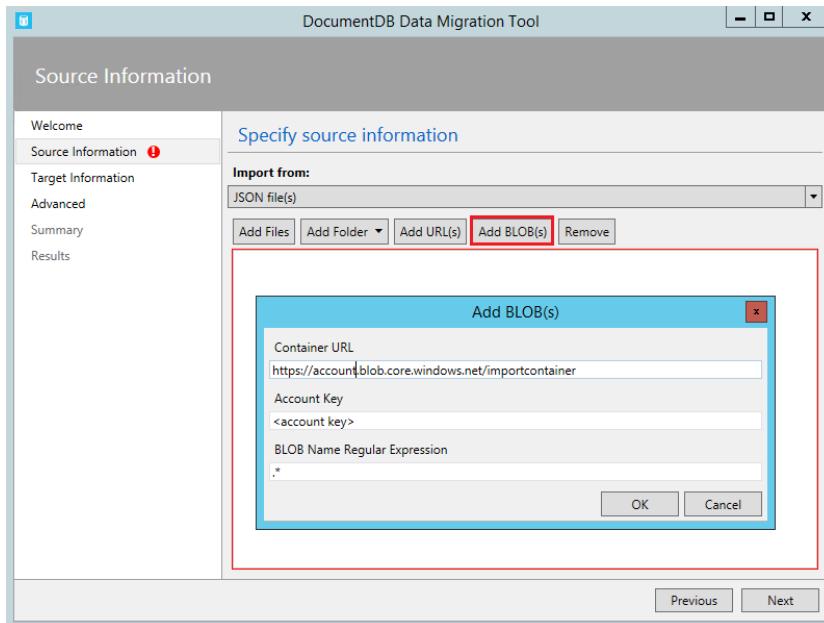
Use the Verify command to ensure that the Amazon DynamoDB instance specified in the connection string field can be accessed.

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.

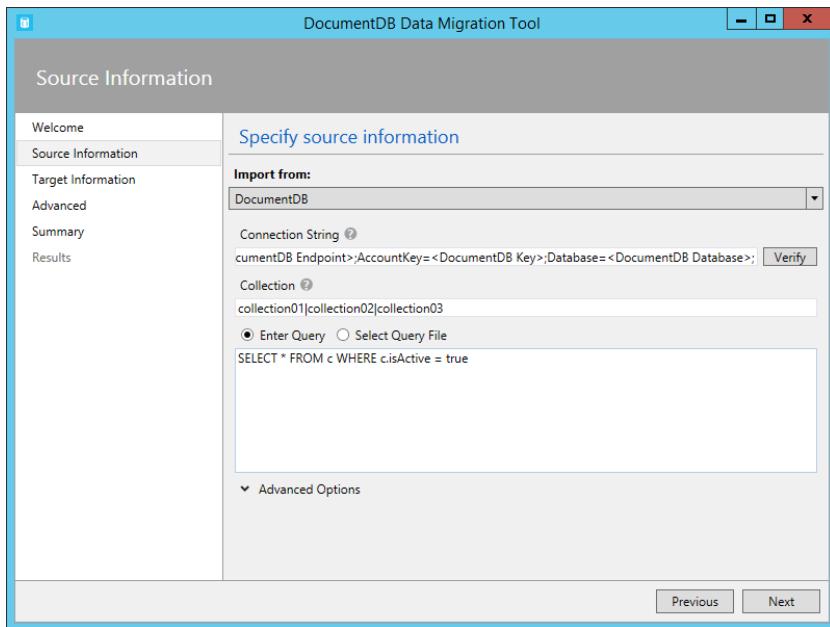


Here is command-line sample to import JSON files from Azure Blob storage:

```
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.

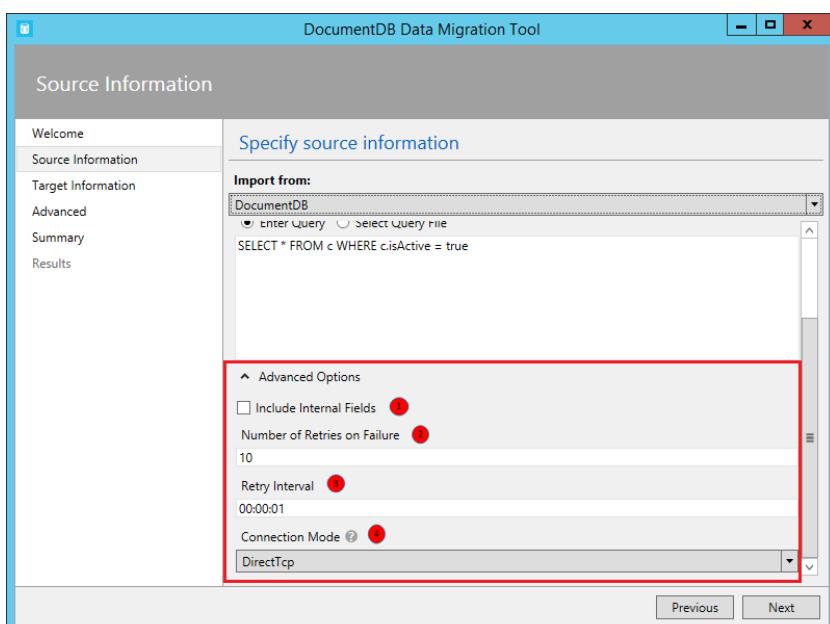
To import from a single Azure Cosmos container, enter the name of the collection to import data from. To import from more than one Azure Cosmos container, provide a regular expression to match one or more collection names (for example, collection01 | collection02 | collection03). You may optionally specify, 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

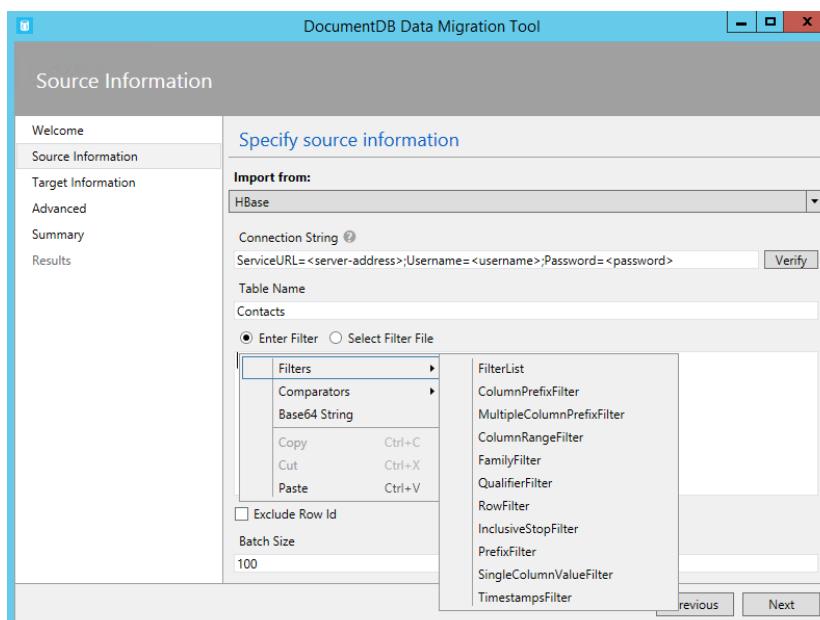
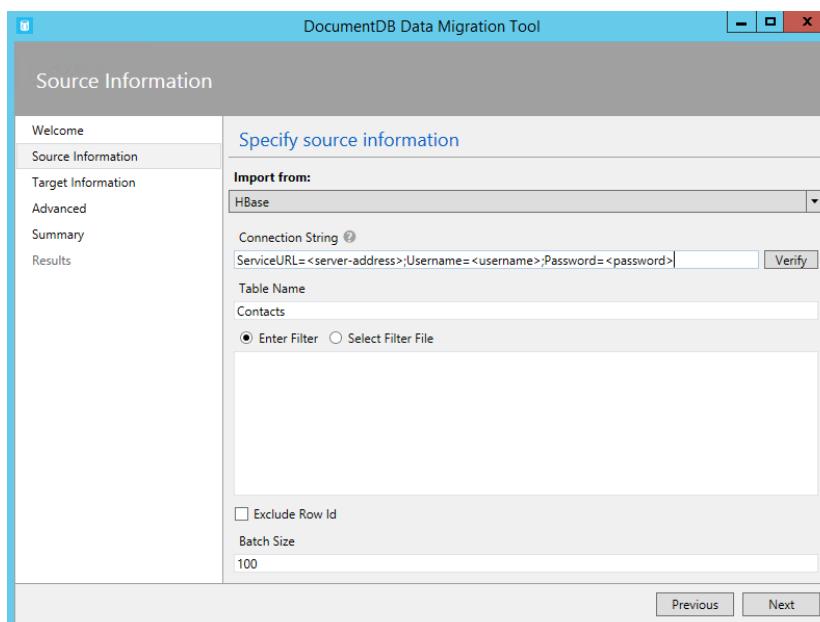
#Export an Azure Cosmos container to a JSON file
dt.exe /s:DocumentDB /s.ConnectionString:"AccountEndpoint=<CosmosDB Endpoint>;AccountKey=<CosmosDB Key>;Database=<CosmosDB Database>;" /s.Collection:StoresSub
/t:JsonFile /t.File:StoresExport.json /t.Overwrite /t.CollectionThroughput:2500
```

**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**

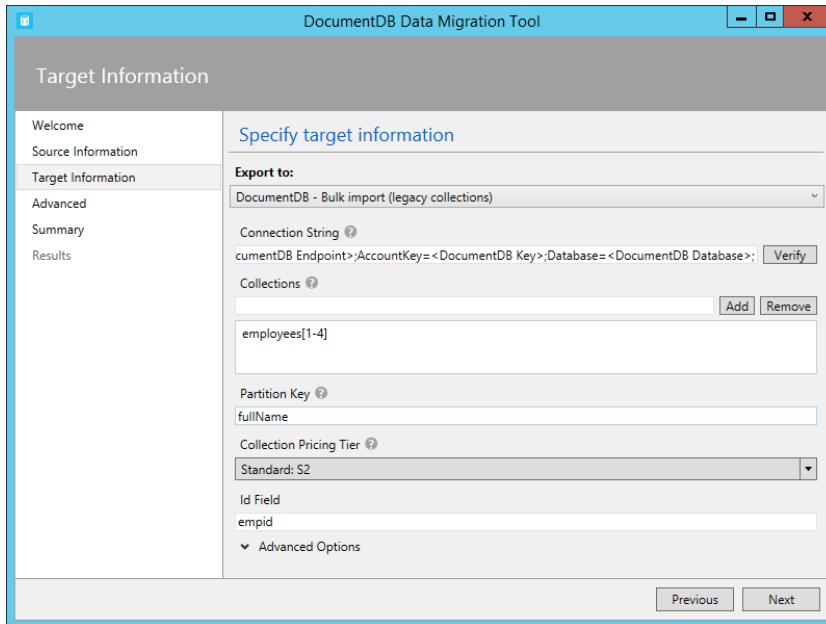
Use the Verify command to ensure that the HBase instance specified in the connection string field can be accessed.

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 (BulkInsertjs), you may choose to specify your own import stored procedure:



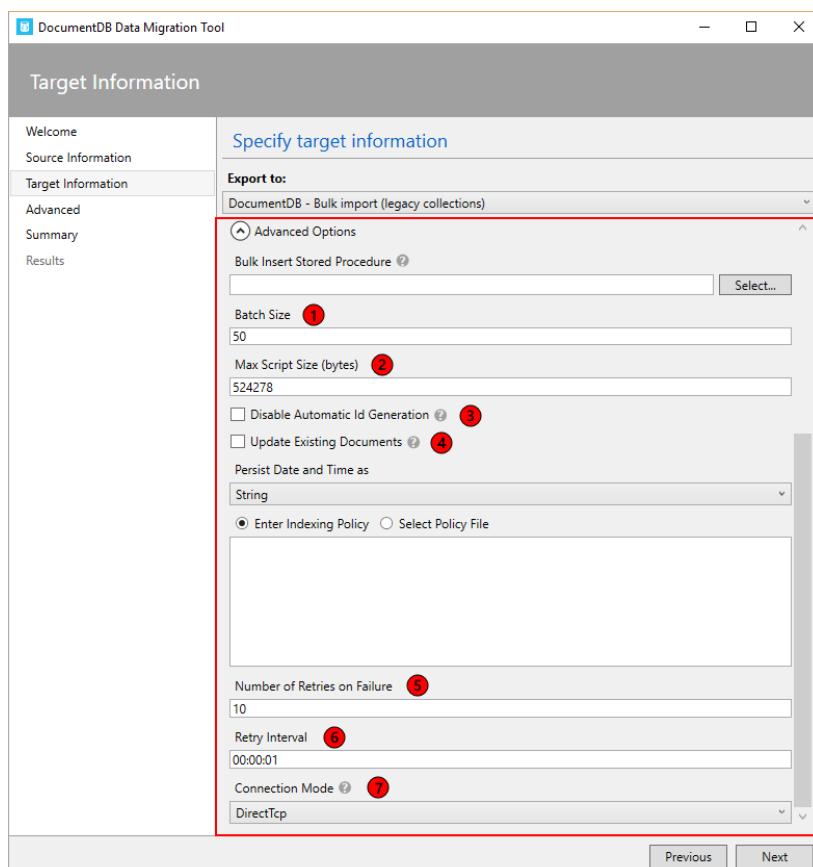
Additionally, when importing date types (for example, from SQL Server or MongoDB), you can choose between three import options:

Persist Date and Time as:  
String  
String  
Epoch  
Both

- 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": "2013-10-21T21:17:25.241000Z", "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.

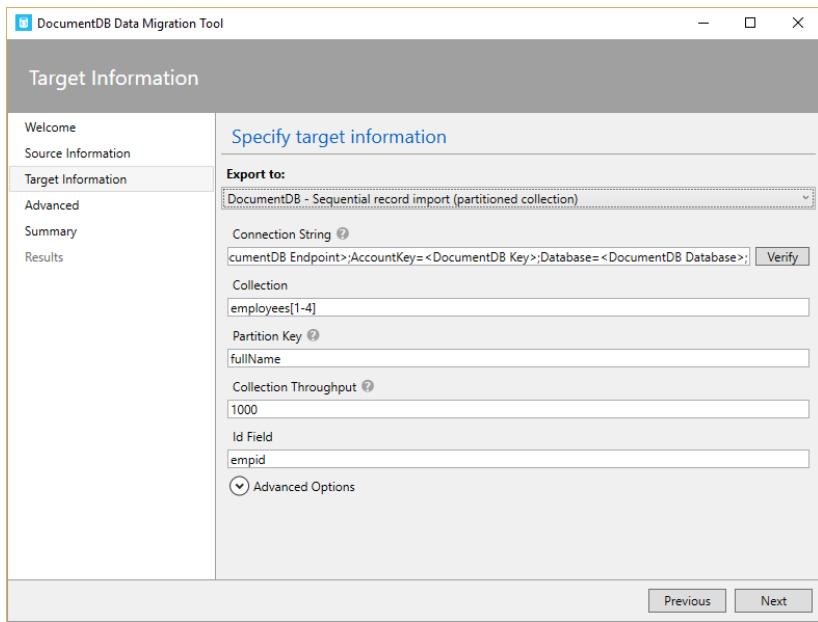


**TIP**

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:

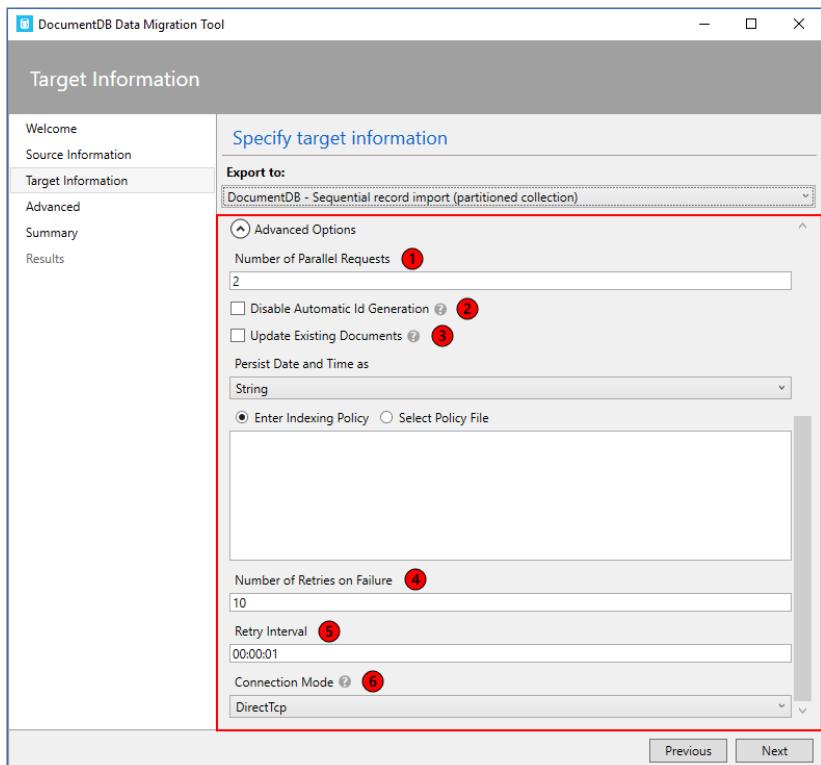
Persist Date and Time as:
<input type="button" value="String"/>
<input checked="" type="button" value="Epoch"/>
<input type="button" value="Both"/>

- 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": "2013-10-21T21:17:25.241000Z", "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.

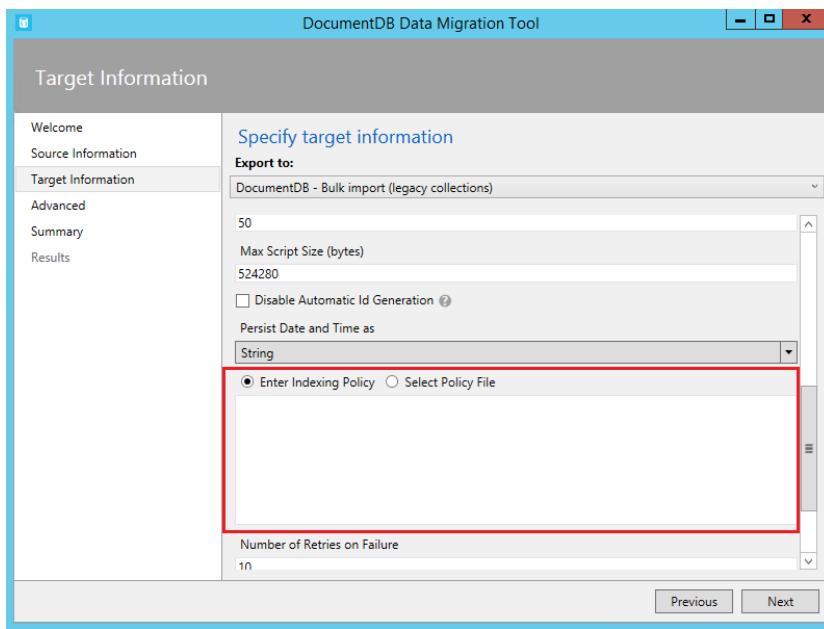


#### TIP

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

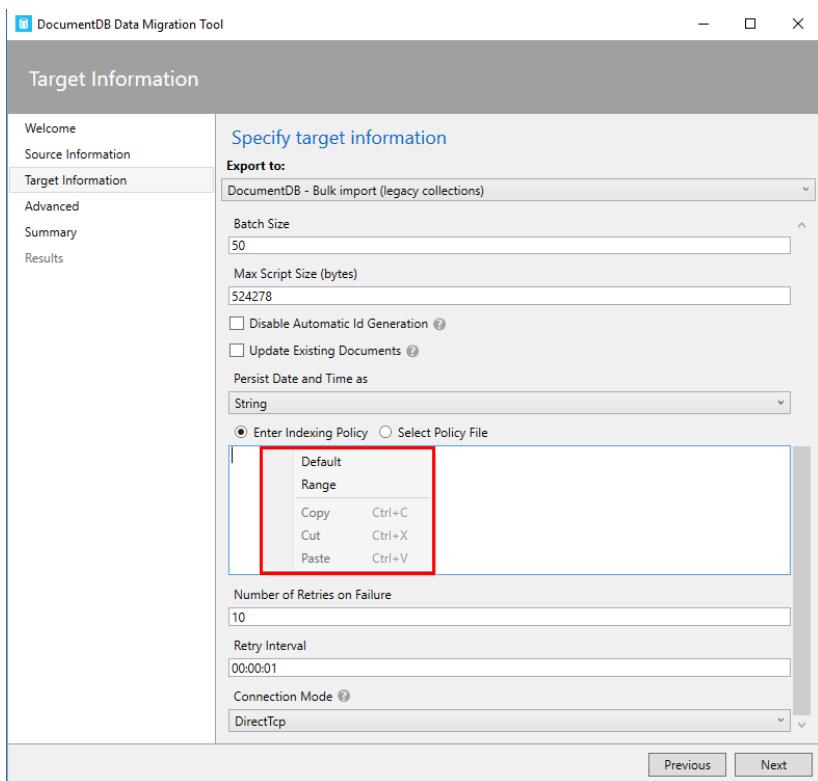
When you allow the migration tool to create Azure Cosmos DB SQL API collections during import, you can specify the indexing policy of the collections. In the advanced options section of the Azure Cosmos DB Bulk import and Azure Cosmos DB Sequential record options, navigate to the Indexing Policy section.



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.

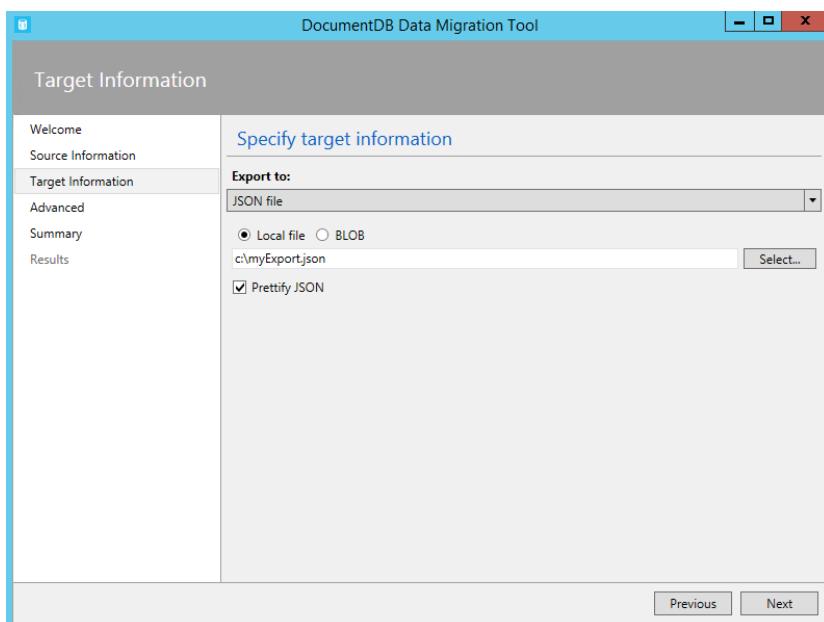


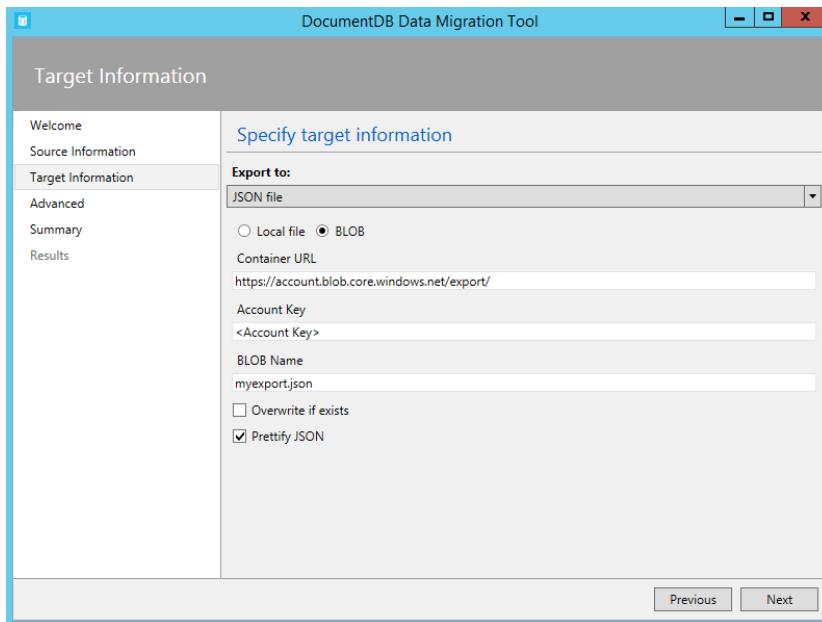
#### NOTE

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"
      }
    ]
  ]
}]
```

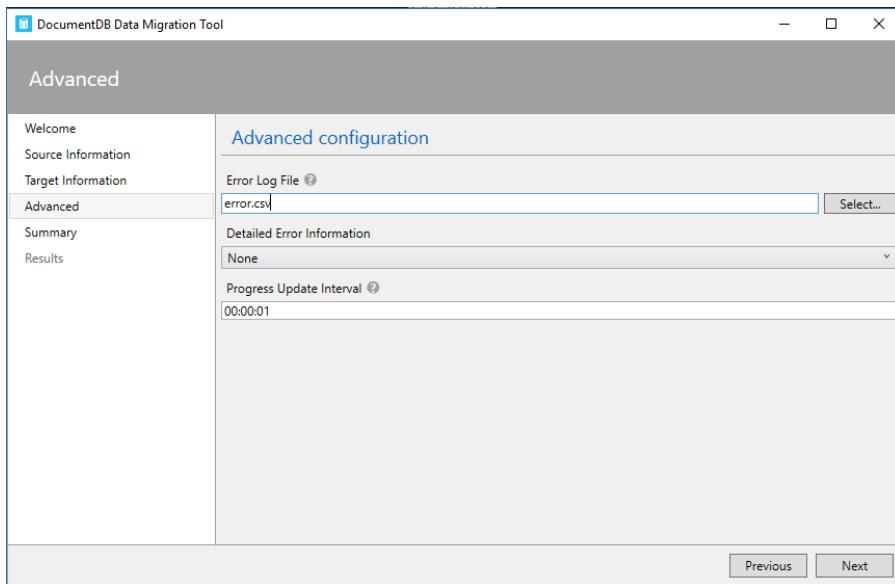
Here is a command-line sample to export the JSON file to Azure Blob storage:

```
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

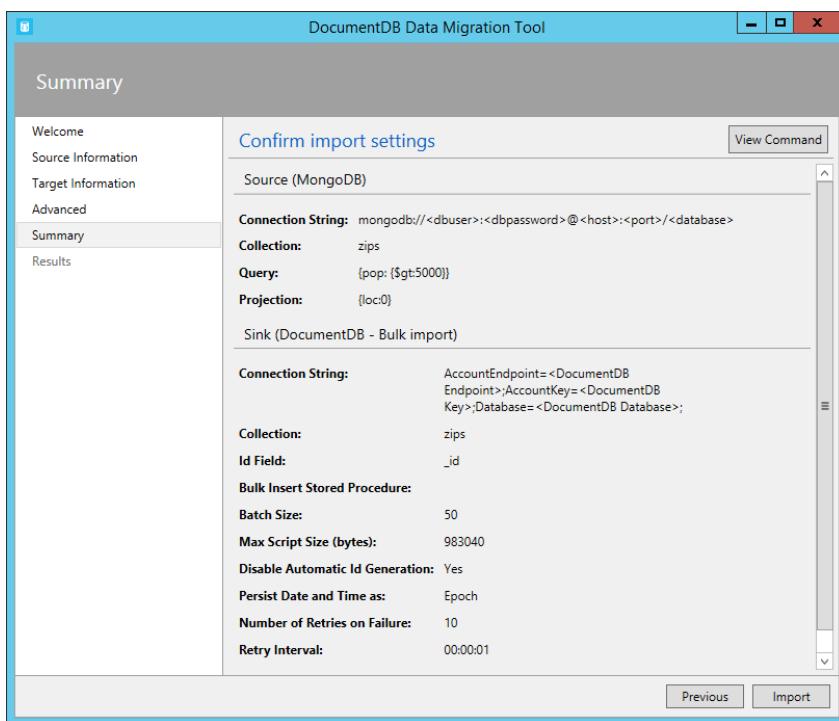
In the Advanced configuration screen, specify the location of the log file to which you would like any errors written. The following rules apply to this page:

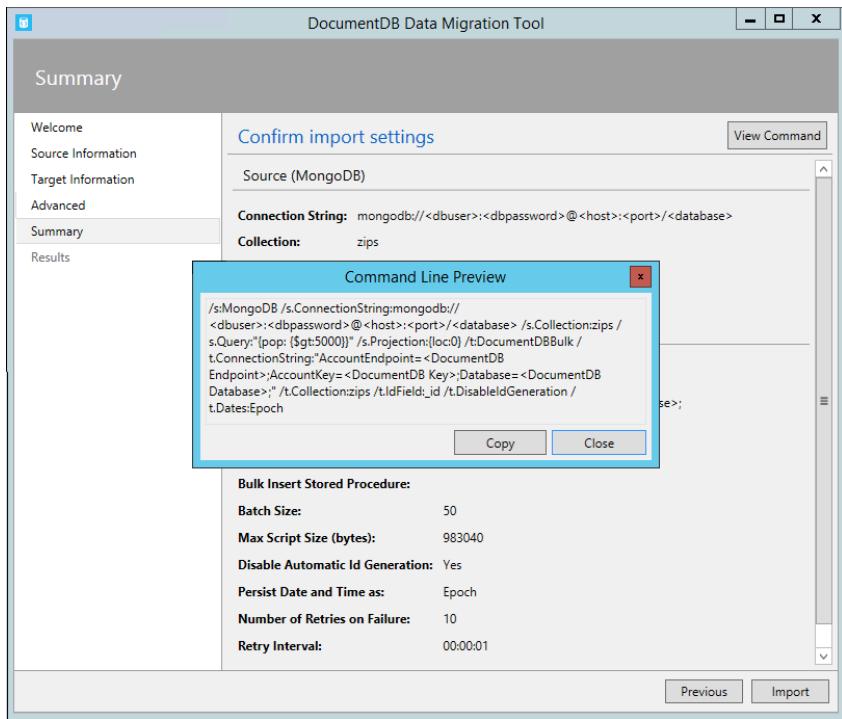
1. If a file name isn't provided, then all errors are returned on the Results page.
2. If a file name is provided without a directory, then the file is created (or overwritten) in the current environment directory.
3. If you select an existing file, then the file is overwritten, there's no append option.
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.



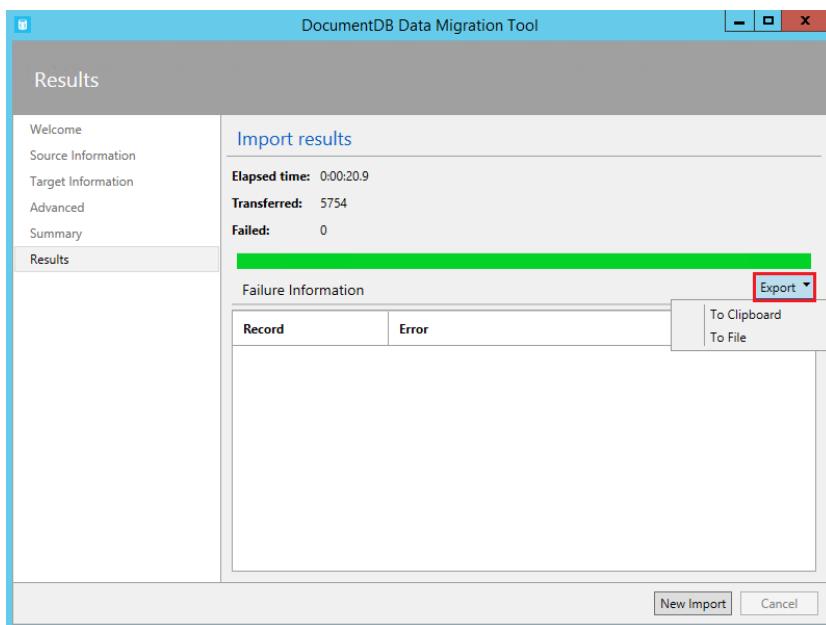
## Confirm import settings and view command line

- 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.)

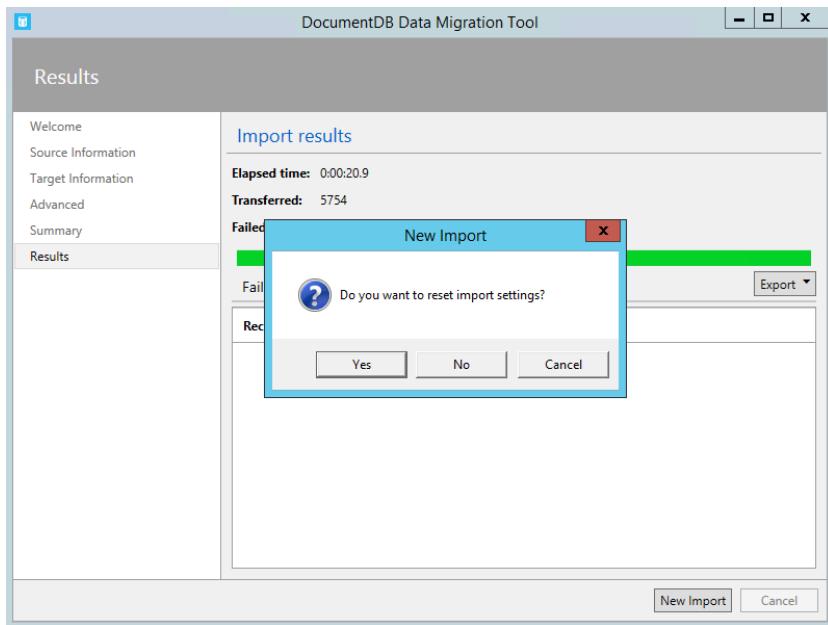




2. Once you're satisfied with your source and target options, click **Import**. The elapsed time, transferred count, and failure information (if you didn't provide a file name in the Advanced configuration) update as the import is in process. Once complete, you can export the results (for example, to 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

2/24/2020 • 6 minutes to read • [Edit Online](#)

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, you can use the [Azure Cosmos DB Emulator](#) with a URL 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 `authorizationKey` 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}) status code
for operation {response.RequestMessage.RequestUri.ToString()}.");
            }
        }
    }));
}
// 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

12/5/2019 • 2 minutes to read • [Edit Online](#)

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? 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` ordered by their grade.

### 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:

- 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

11/6/2019 • 6 minutes to read • [Edit Online](#)

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.

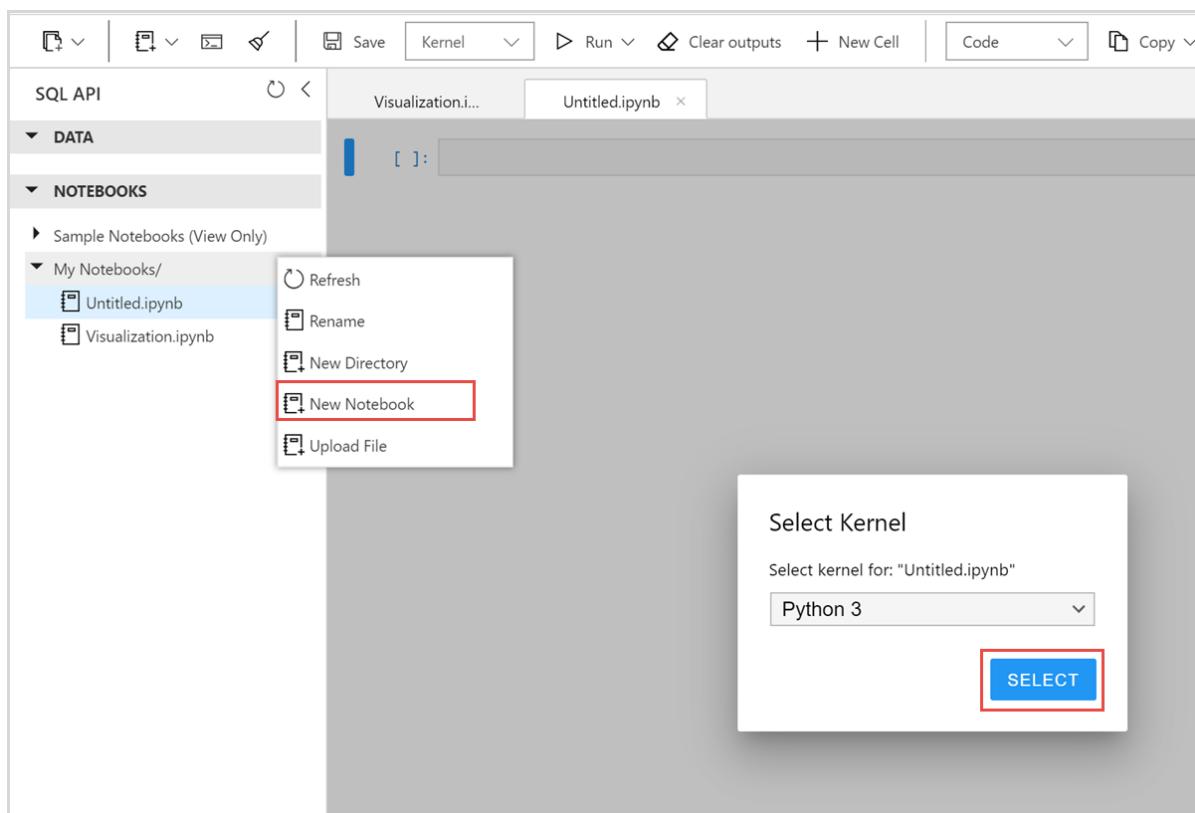
## Prerequisites

- [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 /CardID 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 screenshot shows the Azure Cosmos DB Notebook interface. At the top, there's a toolbar with various icons and dropdown menus. In the center, there's a navigation bar with tabs like 'VisualizeRetailD...', 'Visualizations w...', and 'tebooks'. Below the toolbar, there's a section titled 'Visualizations w...' with a 'Restart Kernel' button. The main area contains a code cell labeled [16] with the following Python code:

```

[16]: 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')

```

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.

The screenshot shows the output of the code execution. It displays two lines of text: 'Database RetailDemo created' and 'Container WebsiteData created'.

You can also refresh the **Data** tab and see the newly created resources:

The screenshot shows the Azure Cosmos DB Data Explorer interface. On the left, there's a sidebar with a tree view. The 'DATA' node is expanded, showing 'RetailDemo' and 'WebsiteData'. There are three numbered callouts: 1) 'Select cell' points to the code cell in the notebook; 2) 'Select Run' points to the 'Run' button in the toolbar; 3) 'Refresh pane' points to the 'Refresh pane' button in the sidebar. The main content area is titled 'Getting started with Cosmos notebooks' and contains instructions for creating a database and container. A code cell [1] is shown with the same Python code as before, and its output is displayed below it: 'Database RetailDemo created' and 'Container WebsiteData created'.

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.CosmosHttpServletResponseError 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 \'{}\' contains \'{}\'' .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 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=False).head(5), columns=['Count'])
```

[21]:

Count	
Item	
Puffy Jacket	25
Athletic Shoes	20
Athletic Shorts	19
Crewneck Sweater	14
Light Jeans	13

## Visualize your data

- 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
```

- Next prepare to plot the data on a map. Join the data in Azure Cosmos DB with country 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 information for mapping
countries =
gpd.read_file("https://cosmosnotebooksdata.blob.core.windows.net/notebookdata/countries.json")

# Merge the countries dataframe with our data in Azure Cosmos DB, joining on country 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)
```

- Visualize the sales revenue of different countries 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)

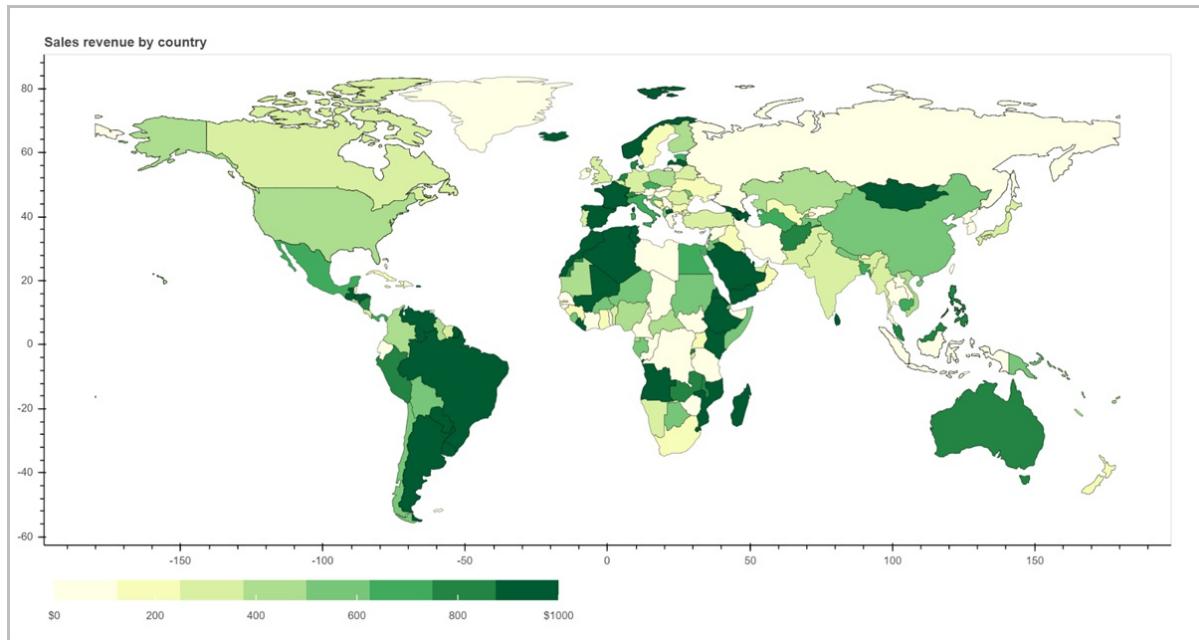
#Specify figure layout.
p.add_layout(color_bar, 'below')

#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 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")])

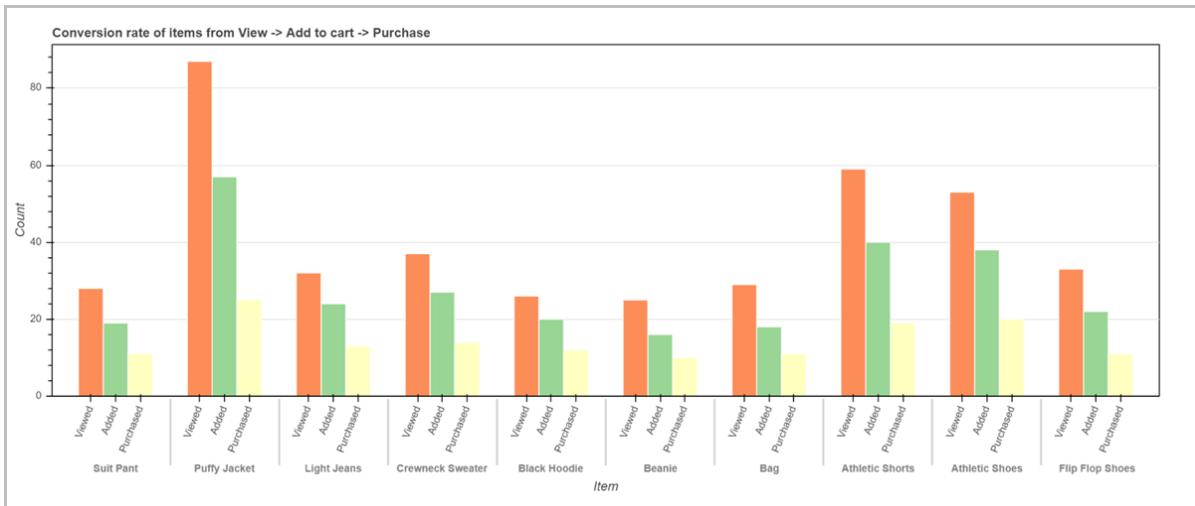
p.vbar(x='Item_Action', top='ItemRevenue_max', width=1, source=group,
line_color="white", fill_color=index_cmap, )

#Configure how the plot looks
p.y_range.start = 0
p.x_range.range_padding = 0.05
p.xaxis.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 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

12/5/2019 • 7 minutes to read • [Edit Online](#)

In this article, we show how to use the Azure portal to setup 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.

andrl - Replicate data globally  
Azure Cosmos DB account

Search (Ctrl+ /)

- Overview
- Activity log
- Access control (IAM)
- Tags
- Diagnose and solve problems
- Quick start
- Data Explorer

**SETTINGS**

- Keys
- Replicate data globally**
- Default consistency
- Firewall
- Add Azure Search
- Properties
- Locks
- Automation script

**MONITORING**

- Metrics
- Alert rules

Save Discard Manual Failover Failover Priorities

Click on a location to add or remove regions from your Azure Cosmos DB account  
\* Each region is billable based on the throughput and storage for the account. Learn more

**WRITE REGION**

- Central US

**READ REGIONS**

- Australia East
- Australia Southeast
- Brazil South
- Canada Central

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. This can be done by setting the connection policy. 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 PreferredLocations list. If the request fails, the client will fail down the list to the next region, and so on.

The SDKs will only attempt to read from the regions specified in PreferredLocations. So, for example, if the Database Account is available in four regions, but the client only specifies two read(non-write) regions for 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 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.

## .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.

```
// 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);
```

## 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 V2 SDK

The following code shows how to set preferred locations by using the Java SDK:

```
ConnectionPolicy policy = new ConnectionPolicy();  
policy.setUsingMultipleWriteLocations(true);  
policy.setPreferredLocations(Arrays.asList("East US", "West US", "Canada Central"));  
AsyncDocumentClient client =  
    new AsyncDocumentClient.Builder()  
        .withMasterKeyOrResourceToken(this.accountKey)  
        .withServiceEndpoint(this.accountEndpoint)  
        .withConnectionPolicy(policy)  
        .build();
```

## REST

Once a database account has been made available in multiple regions, clients can query its availability by performing a GET request on the following 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

9/27/2019 • 4 minutes to read • [Edit Online](#)

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, you can 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	<a href="#">DocumentClient.CreateDatabaseIfNotExistsAsync</a>
Read a database by ID	<a href="#">DocumentClient.ReadDatabaseAsync</a>
Read all the databases	<a href="#">DocumentClient.ReadDatabaseFeedAsync</a>

TASK	API REFERENCE
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	<a href="#">DocumentClient.AccessCondition</a> <a href="#">Documents.Client.AccessConditionType</a>

## 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	<a href="#">IndexingDirective.Exclude</a>
Use Lazy indexing	<a href="#">IndexingPolicy.IndexingMode</a>
Exclude specified document paths from the index	<a href="#">IndexingPolicy.ExcludedPaths</a>
Force a range scan operation on a hash indexed path	<a href="#">FeedOptions.EnableScanInQuery</a>
Use range indexes on strings	<a href="#">IndexingPolicy.IncludedPaths</a> <a href="#">RangeIndex</a>
Perform an index transform	<a href="#">ReplaceDocumentCollectionAsync</a>

## 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	<a href="#">IndexingPolicy</a> <a href="#">IndexKind.Spatial</a> <a href="#">DataType.Point</a>
Insert documents with GeoJSON points	<a href="#">DocumentClient.CreateDocumentAsync</a> <a href="#">DataType.Point</a>
Find points within a specified distance	<a href="#">ST_DISTANCE</a> <a href="#">GeometryOperationExtensions.Distance</a>
Find points within a polygon	<a href="#">ST_WITHIN</a> <a href="#">GeometryOperationExtensions.Within</a> <a href="#">Polygon</a>
Enable geospatial indexing on an existing collection	<a href="#">DocumentClient.ReplaceDocumentCollectionAsync</a> <a href="#">DocumentCollection.IndexingPolicy</a>

TASK	API REFERENCE
Validate point and polygon data	<code>ST_ISVALID</code> <code>ST_ISVALIDDETAILED</code> <code>GeometryOperationExtensions.IsValid</code> <code>GeometryOperationExtensions.IsValidDetailed</code>

## 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	<code>DocumentQueryable.CreateDocumentQuery</code>
Query for equality using ==	<code>DocumentQueryable.CreateDocumentQuery</code>
Query for inequality using != and NOT	<code>DocumentQueryable.CreateDocumentQuery</code>
Query using range operators like >, <, >=, <=	<code>DocumentQueryable.CreateDocumentQuery</code>
Query using range operators against strings	<code>DocumentQueryable.CreateDocumentQuery</code>
Query with ORDER BY	<code>DocumentQueryable.CreateDocumentQuery</code>
Query with aggregate functions	<code>DocumentQueryable.CreateDocumentQuery</code>
Work with subdocuments	<code>DocumentQueryable.CreateDocumentQuery</code>
Query with intra-document Joins	<code>DocumentQueryable.CreateDocumentQuery</code>
Query with string, math, and array operators	<code>DocumentQueryable.CreateDocumentQuery</code>
Query with parameterized SQL using <code>SqlQuerySpec</code>	<code>DocumentQueryable.CreateDocumentQuery</code> <code>SqlQuerySpec</code>
Query with explicit paging	<code>DocumentQueryable.CreateDocumentQuery</code>
Query partitioned collections in parallel	<code>DocumentQueryable.CreateDocumentQuery</code>
Query with ORDER BY for partitioned collections	<code>DocumentQueryable.CreateDocumentQuery</code>

## 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	<a href="#">DocumentClient.CreateDocumentChangeFeedQuery</a>
Read partition key ranges	<a href="#">DocumentClient.ReadPartitionKeyRangeFeedAsync</a>

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	<a href="#">DocumentClient.CreateStoredProcedureAsync</a>
Execute a stored procedure	<a href="#">DocumentClient.ExecuteStoredProcedureAsync</a>
Read a document feed for a stored procedure	<a href="#">DocumentClient.ReadDocumentFeedAsync</a>
Create a stored procedure with ORDER BY	<a href="#">DocumentClient.CreateStoredProcedureAsync</a>
Create a pre-trigger	<a href="#">DocumentClient.CreateTriggerAsync</a>
Create a post-trigger	<a href="#">DocumentClient.CreateTriggerAsync</a>
Create a user-defined function (UDF)	<a href="#">DocumentClient.CreateUserDefinedFunctionAsync</a>

## User management examples

The [RunDemoAsync](#) method of the sample *UserManagement* project shows how to do the following tasks:

TASK	API REFERENCE
Create a user	<a href="#">DocumentClient.CreateUserAsync</a>
Set permissions on a collection or document	<a href="#">DocumentClient.CreatePermissionAsync</a>
Get a list of a user's permissions	<a href="#">DocumentClient.ReadUserAsync</a> <a href="#">DocumentClient.ReadPermissionFeedAsync</a>

# Azure Cosmos DB.NET V3 SDK (Microsoft.Azure.Cosmos) examples for the SQL API

11/8/2019 • 3 minutes to read • [Edit Online](#)

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, you can 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
<a href="#">Create a database</a>	<a href="#">CosmosClient.CreateDatabaseIfNotExistsAsync</a>
<a href="#">Read a database by ID</a>	<a href="#">Database.ReadAsync</a>

Task	API Reference
Read all the databases for an account	<a href="#">CosmosClient.GetDatabaseQueryIterator</a>
Delete a database	<a href="#">Database.DeleteAsync</a>

## 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	<a href="#">Database.CreateContainerIfNotExistsAsync</a>
Create a container with custom index policy	<a href="#">Database.CreateContainerIfNotExistsAsync</a>
Change configured performance of a container	<a href="#">Container.ReplaceThroughputAsync</a>
Get a container by ID	<a href="#">Container.ReadContainerAsync</a>
Read all the containers in a database	<a href="#">Database.GetContainerQueryIterator</a>
Delete a container	<a href="#">Container.DeleteContainerAsync</a>

## 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	<a href="#">Container.CreateItemAsync</a>
Read an item by ID	<a href="#">container.ReadItemAsync</a>
Query for items	<a href="#">container.GetItemQueryIterator</a>
Replace an item	<a href="#">container.ReplaceItemAsync</a>
Upsert an item	<a href="#">container.UpsertItemAsync</a>
Delete an item	<a href="#">container.DeleteItemAsync</a>
Replace an item with conditional ETag check	<a href="#">RequestOptions.IfMatchEtag</a>

## 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	<a href="#">IndexingDirective.Exclude</a>
Use Lazy indexing	<a href="#">IndexingPolicy.IndexingMode</a>
Exclude specified item paths from the index	<a href="#">IndexingPolicy.ExcludedPaths</a>

## 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	<a href="#">container.GetItemQueryIterator</a>
Query items from multiple partitions	<a href="#">container.GetItemQueryIterator</a>
Query using a SQL statement	<a href="#">container.GetItemQueryIterator</a>

## 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	<a href="#">Container.GetChangeFeedProcessorBuilder</a>
Read change feed from a specific time	<a href="#">Container.GetChangeFeedProcessorBuilder</a>
Read change feed from the beginning	<a href="#">ChangeFeedProcessorBuilder.WithStartTime(DateTime)</a>
Migrate from change feed processor to change feed in V3 SDK	<a href="#">Container.GetChangeFeedProcessorBuilder</a>

## 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	<a href="#">Scripts.CreateStoredProcedureAsync</a>
Execute a stored procedure	<a href="#">Scripts.ExecuteStoredProcedureAsync</a>

TASK	API REFERENCE
Delete a stored procedure	<a href="#">Scripts.Delete.StoredProcedureAsync</a>

# Azure Cosmos DB: Java examples for the SQL API

10/8/2019 • 4 minutes to read • [Edit Online](#)

The latest sample applications that perform CRUD operations and other common operations on Azure Cosmos DB resources are included in the [azure-documentdb-java](#) 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

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, you can 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 need the following to run this sample application:

- Java Development Kit 7
- Microsoft Azure DocumentDB Java SDK

You can optionally use Maven to get the latest Microsoft Azure DocumentDB Java SDK 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.microsoft.azure</groupId>
  <artifactId>azure-documentdb</artifactId>
  <version>LATEST</version>
</dependency>
```

## Running the sample applications

Clone the sample repo:

```
$ git clone https://github.com/Azure/azure-documentdb-java.git
$ cd azure-documentdb-java
```

You can run the samples either using Eclipse or from the command line using Maven.

To run from Eclipse:

- Load the main parent project pom.xml file in Eclipse; it should automatically load documentdb-examples.
- To run the samples, you need a valid Azure Cosmos DB Endpoint. The endpoints are read from `src/test/java/com/microsoft/azure/documentdb/examples/AccountCredentials.java`.
- You can pass your endpoint credentials as VM Arguments in Eclipse JUnit Run Config or you can put your endpoint credentials in AccountCredentials.java.

```
-DACOUNT_HOST="https://REPLACE_THIS.documents.azure.com:443/" -DACOUNT_KEY="REPLACE_THIS"
```

- Now you can run the samples as JUnit tests in Eclipse.

To run from the command line:

- The other way for running samples is to use maven:
- Run Maven and pass your Azure Cosmos DB Endpoint credentials:

```
mvn test -DACOUNT_HOST="https://REPLACE_THIS_WITH_YOURS.documents.azure.com:443/" -  
DACOUNT_KEY="REPLACE_THIS_WITH_YOURS"
```

#### NOTE

Each sample is self-contained; it sets itself up and cleans up after itself. The samples issue multiple calls to [DocumentClient.createCollection](#). 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 [DatabaseCrudSamples](#) 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 and read a database	<a href="#">DocumentClient.createDatabase</a> <a href="#">DocumentClient.readDatabase</a> <a href="#">Resource.setId</a>
Create and delete a database	<a href="#">DocumentClient.deleteDatabase</a>
Create and query a database	<a href="#">DocumentClient.queryDatabases</a>

## Collection examples

The [CollectionCrudSamples](#) 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 single partition collection	<a href="#">DocumentClient.createCollection</a>
Create a custom multi-partition collection	<a href="#">DocumentCollection</a> <a href="#">PartitionKeyDefinition</a> <a href="#">RequestOptions</a>
Delete a collection	<a href="#">DocumentClient.deleteCollection</a>

## Document examples

The [DocumentCrudSamples](#) 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, read, and delete a document	<a href="#">DocumentClient.createDocument</a> <a href="#">DocumentClient.readDocument</a> <a href="#">DocumentClient.deleteDocument</a>
Create a document with a programmable document definition	<a href="#">Document</a> <a href="#">Resource.setId</a>

## Indexing examples

The [CollectionCrudSamples](#) 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
Create an index and set indexing policy	<a href="#">Index</a> <a href="#">IndexingPolicy</a>

For more information about indexing, see [Azure Cosmos DB indexing policies](#).

## Query examples

The [DocumentQuerySamples](#) file shows how to perform the following tasks. To learn about the SQL query reference in Azure Cosmos DB before running the following samples, see [SQL query examples](#) conceptual article.

TASK	API REFERENCE
Perform a simple cross-partition document query	<a href="#">DocumentClient.queryDocuments</a> <a href="#">FeedOptions.setEnableCrossPartitionQuery</a>
Order by query	<a href="#">FeedResponse&lt;T&gt;.getQueryIterator</a>

For more information about writing queries, see [SQL query within Azure Cosmos DB](#).

## Offer examples

The [OfferCrudSamples](#) file shows how to perform the following tasks:

TASK	API REFERENCE
Create a collection and set throughput	<a href="#">DocumentClient.createCollection</a> <a href="#">RequestOptions.setOfferThroughput</a>
Read a collection to find the associated offer	<a href="#">Offer.getContent</a> <a href="#">DocumentClient.replaceOffer</a> <a href="#">DocumentClient.readCollection</a> <a href="#">DocumentClient.queryOffers</a>

## Partition key examples

The [SinglePartitionCollectionDocumentCrudSample](#) file shows how to perform the following tasks. To learn about partitioning and partition keys in Azure Cosmos DB before running the following samples, see [Partitioning](#) conceptual article.

TASK	API REFERENCE
Create a single partition collection	<a href="#">DocumentClient.createCollection</a>
Change the throughput offer for a single partition collection	<a href="#">DocumentClient.replaceOffer</a>

## Stored procedure examples

The [StoredProcedureSamples](#) file 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	<a href="#">StoredProcedure</a> <a href="#">DocumentClient.create.StoredProcedure</a>
Run a stored procedure with arguments	<a href="#">DocumentClient.execute.StoredProcedure</a>
Run a stored procedure with an object argument	<a href="#">DocumentClient.execute.StoredProcedure</a>

# Azure Cosmos DB: Async Java examples for the SQL API

10/8/2019 • 4 minutes to read • [Edit Online](#)

The latest sample applications that perform CRUD operations and other common operations on Azure Cosmos DB resources are included in the [azure-cosmosdb-java](#) GitHub repository. This article provides:

- Links to the tasks in each of the example Async Java 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, you can 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 need the following to run this sample application:

- Java Development Kit 8
- Microsoft Azure Cosmos DB Java SDK

You can optionally use Maven to get the latest Microsoft Azure Cosmos DB Java SDK binaries for use in your project. You can choose the latest version from [here](#). 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.microsoft.azure</groupId>
  <artifactId>azure-cosmosdb</artifactId>
  <version>${LATEST}</version>
</dependency>
```

## Running the sample applications

Clone the sample repo:

```
$ git clone https://github.com/Azure/azure-cosmosdb-java.git
$ cd azure-cosmosdb-java
```

You can run the samples either using Eclipse or from the command line using Maven.

To run from Eclipse:

- Load the main parent project pom.xml file in Eclipse; it should automatically load azure-cosmosdb-examples.
- To run the samples, you need a valid Azure Cosmos DB Endpoint. The endpoints are read from `src/test/java/com/microsoft/azure/cosmosdb/rx/examples/TestConfigurations.java`.

- You can pass your endpoint credentials as VM Arguments in Eclipse JUnit Run Config or you can put your endpoint credentials in TestConfigurations.java.

```
-DACOUNT_HOST=<Fill your Azure Cosmos DB account name> -DACOUNT_KEY=<Fill your Azure Cosmos DB primary key>
```

- Now you can run the samples as JUnit tests in Eclipse.

To run from the command line:

- The other way for running samples is to use maven:
- Run Maven and pass your Azure Cosmos DB Endpoint credentials:

```
mvn test -DACOUNT_HOST=<Fill your Azure Cosmos DB account name> -DACOUNT_KEY=<Fill your Azure Cosmos DB primary key>
```

#### **NOTE**

Each sample is self-contained; it sets itself up and cleans up after itself. The samples issue multiple calls to [DocumentClient.createCollection](#). 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 [DatabaseCrudAsyncAPITest](#) 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	<a href="#">AsyncDocumentClient.createDatabase</a>
Fail to create a database that already exists	
Create and read a database	<a href="#">AsyncDocumentClient.readDatabase</a>
Create and delete a database	<a href="#">AsyncDocumentClient.deleteDatabase</a>
Create and query a database	<a href="#">AsyncDocumentClient.queryDatabases</a>

## Collection examples

The [CollectionCrudAsyncAPITest](#) 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 single partition collection	<a href="#">AsyncDocumentClient.createCollection</a>

TASK	API REFERENCE
Create a custom multi-partition collection	DocumentCollection PartitionKeyDefinition RequestOptions
Fail to create a collection that already exists	
Create and read a collection	AsyncDocumentClient.readCollection
Create and delete a collection	AsyncDocumentClient.deleteCollection
Create and query a collection	AsyncDocumentClient.queryCollection

## Document examples

The [DocumentCrudAsyncAPITest](#) 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	AsyncDocumentClient.createDocument
Create a document with a programmable document definition	Document Resource.setId
Create documents and find total RU cost	ResourceResponse.getRequestCharge
Fail to create a document that already exists	
Create and replace a document	AsyncDocumentClient.replaceDocument
Create and upsert a document	AsyncDocumentClient.upsertDocument
Create and delete a document	AsyncDocumentClient.deleteDocument
Create and read a document	AsyncDocumentClient.readDocument

## Indexing examples

The [CollectionCrudAsyncAPITest](#) 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
Create an index and set indexing policy	Index IndexingPolicy

For more information about indexing, see [Azure Cosmos DB indexing policies](#).

## Query examples

The [DocumentQuerySamples](#) file shows how to perform the following tasks. To learn about the SQL query reference in Azure Cosmos DB before running the following samples, see [SQL query examples](#) conceptual article.

TASK	API REFERENCE
Perform a simple document query	<a href="#">AsyncDocumentClient.queryDocuments</a>
Perform a simple document query and find Total RU cost	<a href="#">FeedResponse.getRequestCharge</a>
Perform a simple document query, read one page and unsubscribe from the returned observable	
Perform a simple document query and filter the results	
Perform an order-by cross-partition document query	<a href="#">FeedOptions.setEnableCrossPartitionQuery</a>

For more information about writing queries, see [SQL query within Azure Cosmos DB](#).

## Offer examples

The [OfferCRUDAsyncAPITest](#) file shows how to perform the following tasks:

TASK	API REFERENCE
Create a collection and set throughput	<a href="#">AsyncDocumentClient.createCollection</a> <a href="#">RequestOptions.setOfferThroughput</a>
Read a collection to find the associated offer	<a href="#">Offer.getContent</a> <a href="#">DocumentClient.queryOffers</a>
Update a collection's throughput by replacing its offer	<a href="#">AsyncDocumentClient.replaceOffer</a>

## Stored procedure examples

The [StoredProcedureAsyncAPITest](#) file 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 and run a stored procedure	<a href="#">StoredProcedure</a> <a href="#">DocumentClient.createStoredProcedure</a> <a href="#">AsyncDocumentClient.executeStoredProcedure</a>
Create and run a stored procedure with arguments	
Create and run a stored procedure with an object argument	

## Unique Key

The [UniqueIndexAsyncAPITest](#) file shows how to perform the following tasks. To learn about the unique keys in Azure Cosmos DB before running the following samples, see [Unique key constraints](#) conceptual article.

TASK	API REFERENCE
Create a collection with a unique key	<a href="#">UniqueKey</a> <a href="#">UniqueKeyPolicy</a> <a href="#">DocumentCollection.setUniqueKeyPolicy</a>

# Node.js examples to manage data in Azure Cosmos DB

1/24/2020 • 2 minutes to read • [Edit Online](#)

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, you can 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	<a href="#">Databases.createIfNotExists</a>
List databases for an account	<a href="#">Databases.readAll</a>
Read a database by ID	<a href="#">Database.read</a>
Delete a database	<a href="#">Database.delete</a>

## 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
Create a range index on a string path	IndexKind.Range, IndexingPolicy, Items.query

TASK	API REFERENCE
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

10/8/2019 • 2 minutes to read • [Edit Online](#)

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

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, you can 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 [Python SDK](#).

### NOTE

Each sample is self-contained; it sets itself up and cleans up after itself. The samples issue multiple calls to `CosmosClient.CreateContainer`. Each time this is done, your subscription is billed for one hour of usage. For more information about Azure Cosmos DB billing, see [Azure Cosmos DB Pricing](#).

## Database examples

The [Program.py](#) file of the [DatabaseManagement](#) project 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	<a href="#">CosmosClient.CreateDatabase</a>
Read a database by ID	<a href="#">CosmosClient.ReadDatabase</a>
List databases for an account	<a href="#">CosmosClient.ReadDatabases</a>
Delete a database	<a href="#">CosmosClient.DeleteDatabase</a>

## Collection examples

The [Program.py](#) file of the [CollectionManagement](#) project 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
Create a collection	<a href="#">CosmosClient.CreateContainer</a>
Read a list of all collections in a database	<a href="#">CosmosClient.ReadContainers</a>
Get a collection by ID	<a href="#">CosmosClient.ReadContainer</a>
Change the throughput of a collection	<a href="#">CosmosClient.ReplaceOffer</a>
Delete a collection	<a href="#">CosmosClient.DeleteContainer</a>

## Document examples

The [Program.py](#) file of the [DocumentManagement](#) project 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 a document	<a href="#">CosmosClient.CreateItem</a>
Create a collection of documents	<a href="#">CosmosClient.CreateItem</a>
Read a document by ID	<a href="#">CosmosClient.ReadItem</a>
Read all the documents in a collection	<a href="#">CosmosClient.ReadItems</a>
Replace document with conditional ETag check	<a href="#">CosmosClient.ReplaceItem</a>

## Indexing examples

The [Program.py](#) file of the [IndexManagement](#) project 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
Use manual (instead of automatic) indexing	Automatic indexing policy
Exclude specified document paths from the index	Indexing policy with excluded paths
Exclude a document from the index	<a href="#">IndexingDirective.Exclude</a>
Set indexing mode	<a href="#">IndexingMode</a>
Use range indexes on strings	Indexing policy with included paths
Perform an index transformation	<a href="#">CosmosClient.ReplaceContainer</a>

## Query examples

The sample projects also show how to do the following query tasks. To learn about the SQL query reference in Azure Cosmos DB before running the following samples, see [SQL query examples](#) conceptual article. To learn about the SQL query reference in Azure Cosmos DB before running the following samples, see [SQL query examples](#) conceptual article.

TASK	API REFERENCE
Query an account for a database	<a href="#">CosmosClient.QueryDatabases</a>
Query for documents	<a href="#">CosmosClient.QueryItems</a>
Force a range scan operation on a hash indexed path	<a href="#">HttpHeaders.EnableScanInQuery</a>

# Azure PowerShell samples for Azure Cosmos DB - SQL (Core) API

12/5/2019 • 2 minutes to read • [Edit Online](#)

The following table includes links to commonly used Azure PowerShell scripts for Azure Cosmos DB for SQL (Core) API. If you'd like to fork these PowerShell samples for Cosmos DB from our GitHub repository visit, [Cosmos DB PowerShell Samples on GitHub](#).

For additional Cosmos DB PowerShell samples for SQL (Core) API and documentation see, [Manage Azure Cosmos DB SQL API resources using PowerShell](#). For Cosmos DB PowerShell samples for other APIs see, [Cassandra API](#), [MongoDB API](#), [Gremlin API](#), and [Table API](#).

<a href="#">Create an account, database and container</a>	Create an Azure Cosmos account, database and container.
<a href="#">Create a container with a large partition key</a>	Create a container with a large partition key.
<a href="#">List or get databases or containers</a>	List or get database or containers.
<a href="#">Get RU/s</a>	Get RU/s for a database or container.
<a href="#">Update RU/s</a>	Update RU/s for a database or container.
<a href="#">Create a container with no index policy</a>	Create an Azure Cosmos container with index policy turned off.
<a href="#">Update an account or add a region</a>	Add a region to a Cosmos account. Can also be used to modify other account properties but these must be separate from changes to regions.
<a href="#">Change failover priority or trigger failover</a>	Change the regional failover priority of an Azure Cosmos account or trigger a manual failover.
<a href="#">Account keys or connection strings</a>	Get primary and secondary keys, connection strings or regenerate an account key of an Azure Cosmos account.
<a href="#">Create a Cosmos Account with IP Firewall</a>	Create an Azure Cosmos account with IP Firewall enabled.

# Azure CLI samples for Azure Cosmos DB SQL (Core) API

9/25/2019 • 2 minutes to read • [Edit Online](#)

The following table includes links to sample Azure CLI scripts for Azure Cosmos DB SQL (Core) API. Reference pages for all Azure Cosmos DB CLI commands are available in the [Azure CLI Reference](#). For Azure CLI samples for other Azure Cosmos DB APIs see, [Cassandra API](#), [MongoDB API](#), [Gremlin API](#), and [Table API](#). All Azure Cosmos DB CLI script samples can be found in the [Azure Cosmos DB CLI GitHub Repository](#).

<a href="#">Create an Azure Cosmos account, database and container</a>	Creates an Azure Cosmos DB account, database, and container for SQL (Core) API.
<a href="#">Change throughput</a>	Update RU/s on a database and container.
<a href="#">Add or failover regions</a>	Add a region, change failover priority, trigger a manual failover.
<a href="#">Account keys and connection strings</a>	List account keys, read-only keys, regenerate keys and list connection strings.
<a href="#">Secure with IP firewall</a>	Create a Cosmos account with IP firewall configured.
<a href="#">Secure new account with service endpoints</a>	Create a Cosmos account and secure with service-endpoints.
<a href="#">Secure existing account with service endpoints</a>	Update a Cosmos account to secure with service-endpoints when the subnet is eventually configured.

# Azure Resource Manager templates for Azure Cosmos DB

11/12/2019 • 2 minutes to read • [Edit Online](#)

The following tables include links to Azure Resource Manager templates for Azure Cosmos DB:

## SQL (Core) API

TEMPLATE	DESCRIPTION
<a href="#">Create an Azure Cosmos account, database, container</a>	This template creates a SQL (Core) API account in two regions with two containers with shared database throughput and a container with dedicated throughput. Throughput can be updated by resubmitting the template with updated throughput property value.
<a href="#">Create an Azure Cosmos account, database and container with a stored procedure, trigger and UDF</a>	This template creates a SQL (Core) API account in two regions with a stored procedure, trigger and UDF for a container.

## MongoDB API

TEMPLATE	DESCRIPTION
<a href="#">Create an Azure Cosmos account, database, collection</a>	This template creates an account using Azure Cosmos DB API for MongoDB in two regions with multi-master enabled. The Azure Cosmos account will have two containers that share database-level throughput.

## Cassandra API

TEMPLATE	DESCRIPTION
<a href="#">Create an Azure Cosmos account, keyspace, table</a>	This template creates a Cassandra API account in two regions with multi-master enabled. The Azure Cosmos account will have two tables that share keyspace-level throughput.

## Gremlin API

TEMPLATE	DESCRIPTION
<a href="#">Create an Azure Cosmos account, database, graph</a>	This template creates a Gremlin API account in two regions with multi-master enabled. The Azure Cosmos account will have two graphs that share database-level throughput.

## Table API

TEMPLATE	DESCRIPTION
<a href="#">Create an Azure Cosmos account, table</a>	This template creates a Table API account in two regions with multi-master enabled. The Azure Cosmos account will have a single table.

**TIP**

To enable shared throughput when using Table API, enable account-level throughput in the Azure Portal.

See [Azure Resource Manager reference for Azure Cosmos DB](#) page for the reference documentation.

# Understanding the differences between NoSQL and relational databases

1/10/2020 • 8 minutes to read • [Edit Online](#)

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](#).

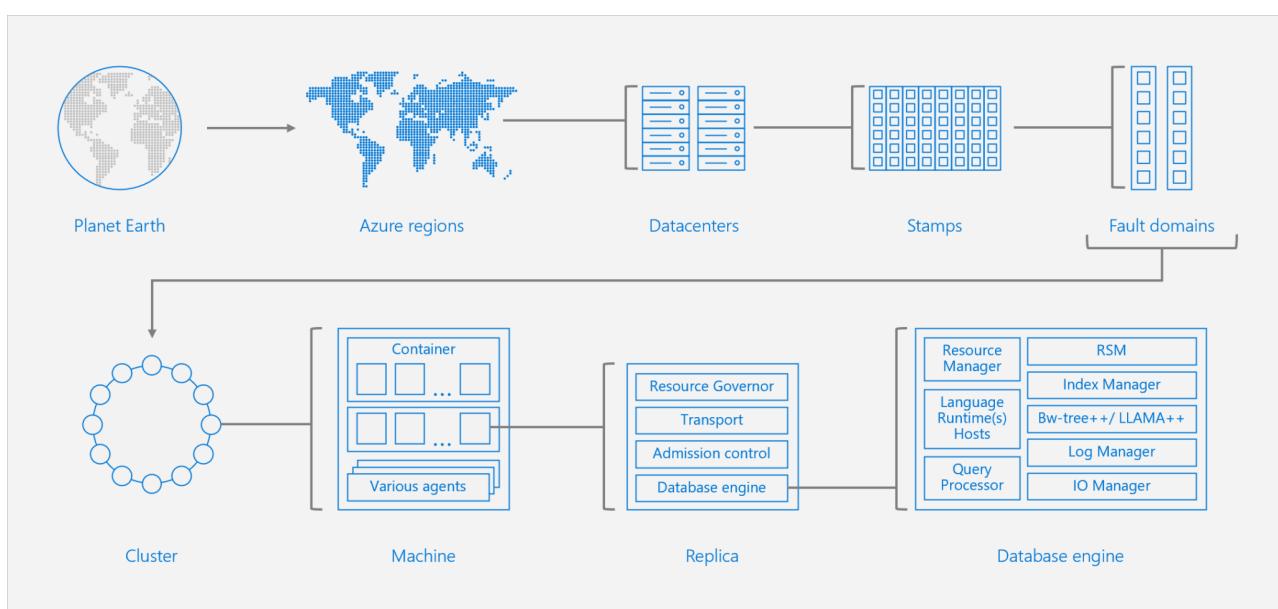
## High throughput

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.



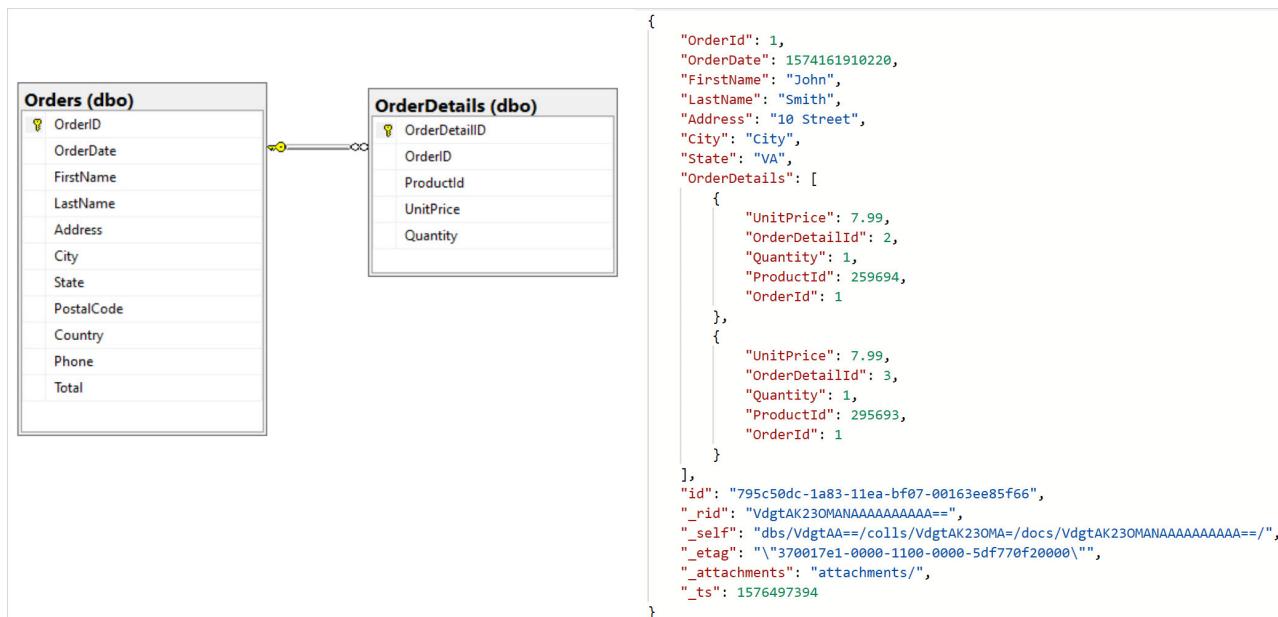
## Hierarchical data

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](#).

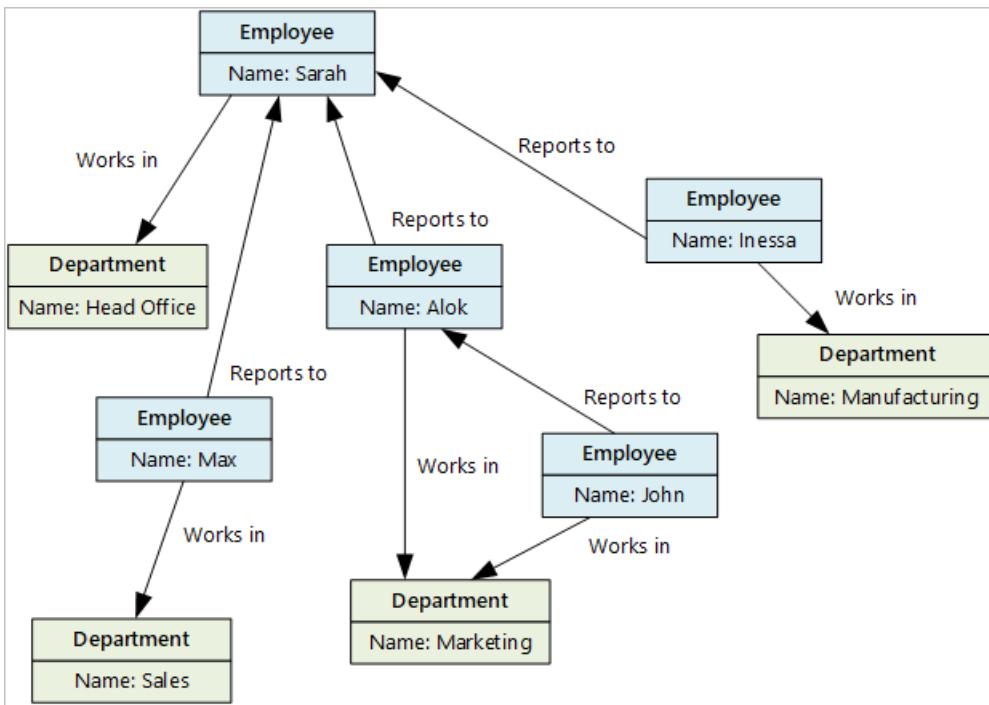


## Complex networks and relationships

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:

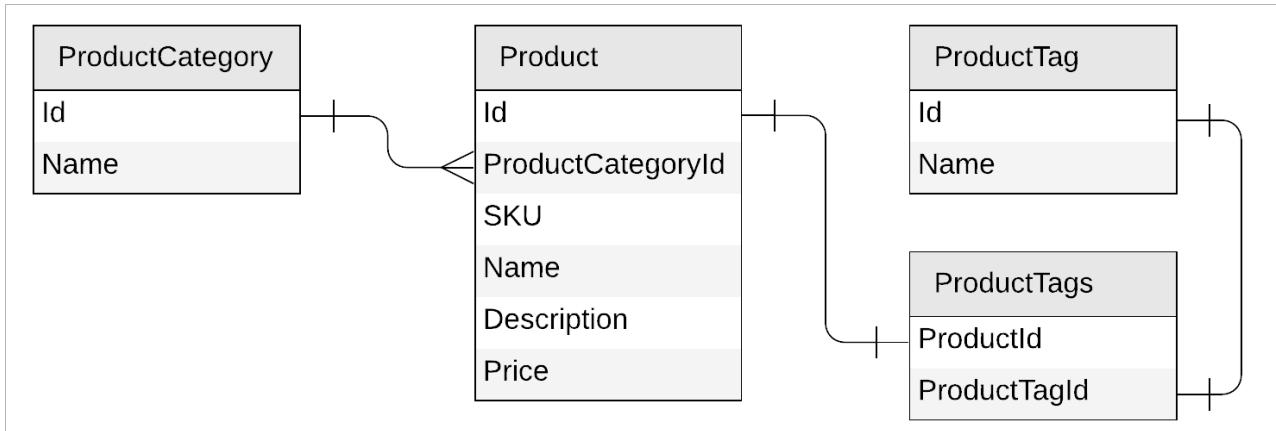
- a choice of pure JSON data types
- a JavaScript engine and [query API](#) built into the database.
- a state-of-the-art [change feed](#) which clients can subscribe to in order to get notified of modifications to a container.

# 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 denormalized 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](#)

# Global data distribution with Azure Cosmos DB – overview

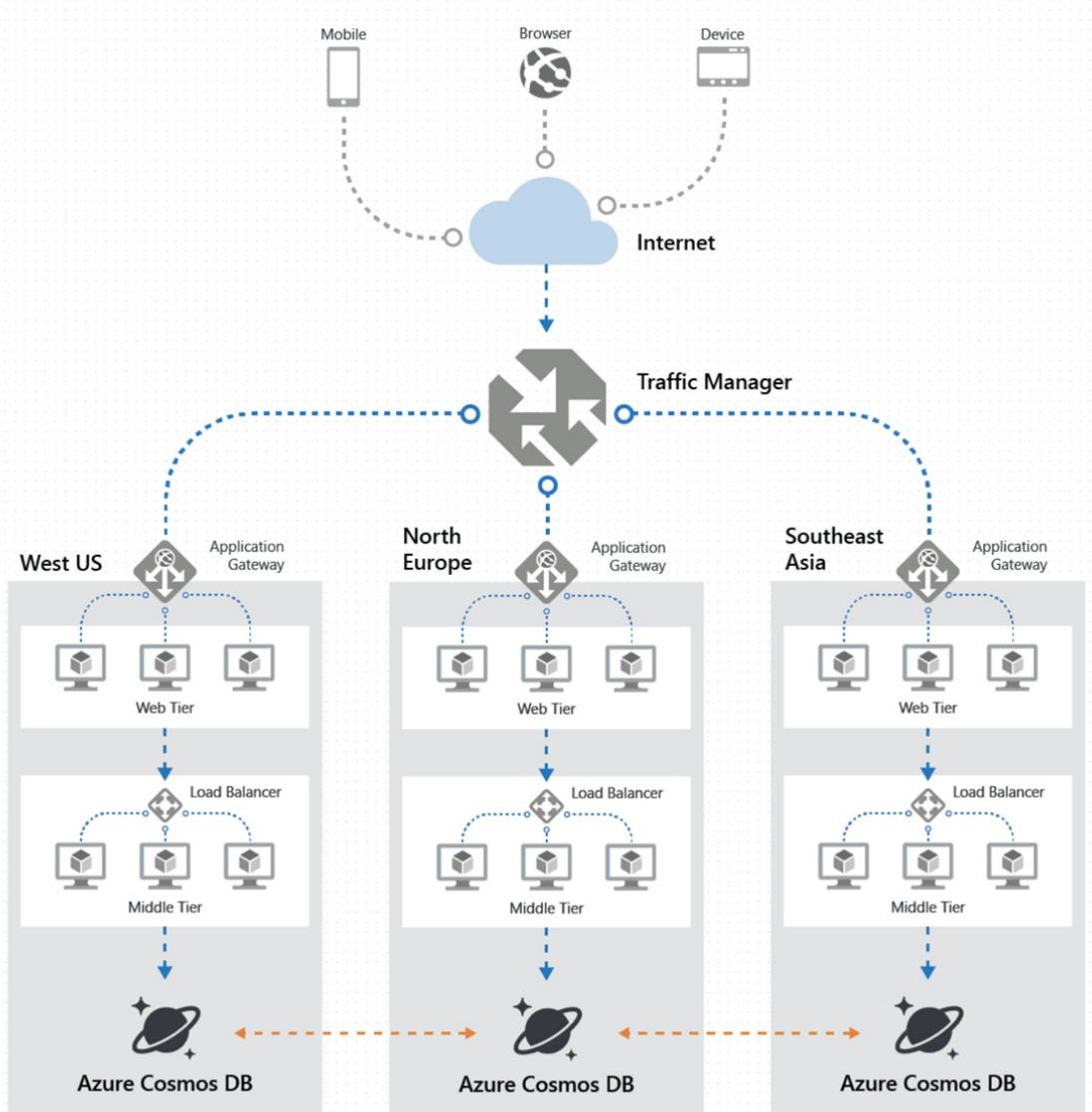
1/17/2020 • 3 minutes to read • [Edit Online](#)

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 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 guaranteed 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-master replication protocol, every region supports both writes and reads. The multi-master 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 guaranteed to be delivered 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-master 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

7/22/2019 • 5 minutes to read • [Edit Online](#)

Distributed databases that rely on replication for high availability, low latency, or both, make the fundamental tradeoff between the read consistency vs. availability, latency, and throughput. Most commercially available distributed databases ask developers to choose between the two extreme consistency models: *strong* consistency and *eventual* consistency. The linearizability or the strong consistency model is the gold standard of data programmability. But it adds a price of higher latency (in steady state) and reduced availability (during failures). On the other hand, eventual consistency offers higher availability and better performance, but makes it hard to program applications.

Azure Cosmos DB approaches data consistency as a spectrum of choices instead of two extremes. Strong consistency and eventual consistency are at the ends of the spectrum, but there are many consistency choices along the spectrum. Developers can use these options to make precise choices and granular tradeoffs with respect to high availability and performance.

With Azure Cosmos DB, developers can choose from five well-defined consistency models on the consistency spectrum. From strongest to more relaxed, the models include *strong*, *bounded staleness*, *session*, *consistent prefix*, and *eventual* consistency. The models are well-defined and intuitive and can be used for specific real-world scenarios. Each model provides [availability and performance tradeoffs](#) and is backed by the SLAs. 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.

## Scope of the read consistency

Read consistency applies to a single read operation scoped within a partition-key range or 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](#).

## Guarantees associated with consistency levels

The comprehensive SLAs provided by Azure Cosmos DB guarantee that 100 percent of read requests meet the consistency guarantee for any consistency level you choose. A read request meets the consistency SLA if all the consistency guarantees associated with the consistency level are satisfied. 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.
- **Bounded staleness:** The reads are guaranteed to honor the consistent-prefix guarantee. The reads might lag behind writes by at most "K" versions (i.e., "updates") of an item or by "T" time interval. 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 ( $T$ ) by which the reads might lag behind the writesBounded staleness offers total global order except within the "staleness window." The monotonic read guarantees exist within a region both inside and outside the staleness window. Strong consistency has the same semantics as the one offered by bounded staleness. The staleness window is equal to zero. Bounded staleness is also referred to as time-delayed linearizability. 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.
- **Session:** Within a single client session reads are guaranteed to honor the consistent-prefix (assuming a single "writer" session), monotonic reads, monotonic writes, read-your-writes, and write-follows-reads guarantees. Clients outside of the session performing writes will see eventual consistency.
- **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.
- **Eventual:** There's no ordering guarantee for reads. In the absence of any further writes, the replicas eventually converge.

## Consistency levels explained through baseball

Let's take a baseball game scenario as an example. Imagine a sequence of writes that represent the score from a baseball game. The inning-by-inning line score is described in the [Replicated data consistency through baseball](#) paper. This hypothetical baseball game is currently in the middle of the seventh inning. It's the seventh-inning stretch. The visitors are behind with a score of 2 to 5 as shown below:

	1	2	3	4	5	6	7	8	9	RUNS
Visitors	0	0	1	0	1	0	0			2
Home	1	0	1	1	0	2				5

An Azure Cosmos container holds the run totals for the visitors and home teams. While the game is in progress, different read guarantees might result in clients reading different scores. The following table lists the complete set of scores that might be returned by reading the visitors' and home scores with each of the five consistency guarantees. The visitors' score is listed first. Different possible return values are separated by commas.

CONSISTENCY LEVEL	SCORES (VISITORS, HOME)
<b>Strong</b>	2-5
<b>Bounded staleness</b>	Scores that are at most one inning out of date: 2-3, 2-4, 2-5
<b>Session</b>	<ul style="list-style-type: none"> <li>For the writer: 2-5</li> <li>For anyone other than the writer: 0-0, 0-1, 0-2, 0-3, 0-4, 0-5, 1-0, 1-1, 1-2, 1-3, 1-4, 1-5, 2-0, 2-1, 2-2, 2-3, 2-4, 2-5</li> <li>After reading 1-3: 1-3, 1-4, 1-5, 2-3, 2-4, 2-5</li> </ul>
<b>Consistent prefix</b>	0-0, 0-1, 1-1, 1-2, 1-3, 2-3, 2-4, 2-5
<b>Eventual</b>	0-0, 0-1, 0-2, 0-3, 0-4, 0-5, 1-0, 1-1, 1-2, 1-3, 1-4, 1-5, 2-0, 2-1, 2-2, 2-3, 2-4, 2-5

## 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:

- [Choose the right consistency level for your application](#)
- [Consistency levels across Azure Cosmos DB APIs](#)
- [Availability and performance tradeoffs for various consistency levels](#)
- [Configure the default consistency level](#)
- [Override the default consistency level](#)

# Choose the right consistency level

12/13/2019 • 3 minutes to read • [Edit Online](#)

Distributed databases relying on replication for high availability, low latency or both, make the fundamental tradeoff between the read consistency vs. availability, latency, and throughput. Most commercially available distributed databases ask developers to choose between the two extreme consistency models: *strong* consistency and *eventual* consistency. Azure Cosmos DB allows developers to choose among the five well-defined consistency models: *strong*, *bounded staleness*, *session*, *consistent prefix* and *eventual*. Each of these consistency models is well-defined, intuitive and can be used for specific real-world scenarios. Each of the five consistency models provide precise [availability and performance tradeoffs](#) and are backed by comprehensive SLAs. The following simple considerations will help you make the right choice in many common scenarios.

## SQL API and Table API

Consider the following points if your application is built using SQL API or Table API:

- For many real-world scenarios, session consistency is optimal and it's the recommended option. For more information, see, [How-to manage session token for your application](#).
- If your application requires strong consistency, it is recommended that you use bounded staleness consistency level.
- If you need stricter consistency guarantees than the ones provided by session consistency and single-digit-millisecond latency for writes, it is recommended that you use bounded staleness consistency level.
- If your application requires eventual consistency, it is recommended that you use consistent prefix consistency level.
- If you need less strict consistency guarantees than the ones provided by session consistency, it is recommended that you use consistent prefix consistency level.
- If you need the highest availability and the lowest latency, then use eventual consistency level.
- If you need even higher data durability without sacrificing performance, you can create a custom consistency level at the application layer. For more information see, [How-to implement custom synchronization in your applications](#).

## Cassandra, MongoDB, and Gremlin APIs

- For details on mapping between "Read Consistency Level" offered in Apache Cassandra and Cosmos DB consistency levels, see [Consistency levels and Cosmos DB APIs](#).
- For details on mapping between "Read Concern" of MongoDB and Azure Cosmos DB consistency levels, see [Consistency levels and Cosmos DB APIs](#).

## 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.

- When the consistency level is set to **bounded staleness**, Cosmos DB guarantees that the clients always read the value of a previous write, with a lag bounded by the staleness window.

- When the consistency level is set to **strong**, the staleness window is equivalent to zero, and the clients are guaranteed to read the latest committed value of the write operation.
- For the remaining three consistency levels, the staleness window is largely dependent on your workload. For example, 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.

## Next steps

Read more about the consistency levels in the following articles:

- [Consistency level mapping across Cosmos DB APIs](#)
- [Availability and performance tradeoffs for various consistency levels](#)
- [How to manage the session token for your application](#)
- [Monitor Probabilistically Bounded Staleness \(PBS\) metric](#)

# Consistency levels and Azure Cosmos DB APIs

12/13/2019 • 2 minutes to read • [Edit Online](#)

Azure Cosmos DB provides native support for wire protocol-compatible APIs for popular databases. These include MongoDB, Apache Cassandra, Gremlin, and Azure Table storage. These databases do not offer precisely defined consistency models or SLA-backed guarantees for the consistency levels. They typically provide only a subset of the five consistency models offered by Azure Cosmos DB.

When using SQL API, Gremlin API, and Table API, the default consistency level configured on the Azure Cosmos account is used.

When using Cassandra API or Azure Cosmos DB's API for MongoDB, applications get a full set of consistency levels offered by Apache Cassandra and MongoDB, respectively, with even stronger consistency and durability guarantees. This document shows the corresponding Azure Cosmos DB consistency levels for Apache Cassandra and MongoDB consistency levels.

## Mapping between Apache Cassandra and Azure Cosmos DB consistency levels

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.
- 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.

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	Linearizable 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	Linearizable 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	Linearizable 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	Linearizable 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	Linearizable 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	Linearizable 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

## Mapping between MongoDB and Azure Cosmos DB consistency levels

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.

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 "Master" by making the region as the first writable region.

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	<i>isMaster</i> = 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 ( <i>time-bounded</i> linearizability) in all other regions
			LINEARIZABILITY	Strong	Linearizability	Linearizability
			SNAPSHOT	Strong	Atomic reads across documents	Linearizability
		<i>isMaster</i> = 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	<i>isMaster</i> = 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 ( <i>time-bounded</i> 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 ( <i>time-bounded</i> linearizability) in all other regions
			LINEARIZABILITY	Bounded Staleness	Linearizability	Linearizable reads in the region where the write is performed Bounded staleness ( <i>time-bounded</i> linearizability) in all other regions
			SNAPSHOT	Bounded Staleness	Linearizability	Linearizable reads in the region where the write is performed Bounded staleness ( <i>time-bounded</i> linearizability) in all other regions
		<i>isMaster</i> = 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	<i>isMaster</i> = true/false	ANY	Consistent Prefix	Stale Reads; Dirty Reads	Consistent Prefix

## Next steps

Read more about consistency levels and compatibility between Azure Cosmos DB APIs with the open-source APIs. See the following articles:

- [Availability and performance tradeoffs for various consistency levels](#)
- [MongoDB features supported by the Azure Cosmos DB's API for MongoDB](#)
- [Apache Cassandra features supported by the Azure Cosmos DB Cassandra API](#)

# Consistency, availability, and performance tradeoffs

12/13/2019 • 3 minutes to read • [Edit Online](#)

Distributed databases that rely on replication for high availability, low latency, or both must make tradeoffs. The tradeoffs are between read consistency vs. availability, latency, and throughput.

Azure Cosmos DB approaches data consistency as a spectrum of choices. This approach includes more options than the two extremes of strong and eventual consistency. You can choose from five well-defined models on the consistency spectrum. From strongest to weakest, the models are:

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

Each model provides availability and performance tradeoffs and is backed by comprehensive SLAs.

## Consistency levels and latency

The read latency for all consistency levels is always guaranteed to be less than 10 milliseconds at the 99th percentile. This read latency is backed by the SLA. The average read latency, at the 50th percentile, is typically 2 milliseconds or less. Azure Cosmos accounts that span several regions and are configured with strong consistency are an exception to this guarantee.

The write latency for all consistency levels is always guaranteed to be less than 10 milliseconds at the 99th percentile. This write latency is backed by the SLA. The average write latency, at the 50th percentile, is usually 5 milliseconds or less.

For Azure Cosmos accounts configured with strong consistency with more than one region, the write latency is guaranteed to be less than two times round-trip time (RTT) between any of the two farthest regions, plus 10 milliseconds at the 99th percentile.

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. For your Azure Cosmos account, replication latencies are displayed in the Azure portal. You can use the Azure portal (go to the Metrics blade) to monitor the replication latencies between various regions that are associated with your Azure Cosmos account.

## Consistency levels and throughput

- For the same number of request units, the session, consistent prefix, and eventual consistency levels provide about two times the read throughput when compared with strong and bounded staleness.
- 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 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 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 the CAP Theorem. To learn more about why, see [Consistency levels in Azure Cosmos DB](#).

REGION(S)	REPLICATION MODE	CONSISTENCY LEVEL	RPO	RTO
1	Single or Multi-Master	Any Consistency Level	< 240 Minutes	< 1 Week
>1	Single Master	Session, Consistent Prefix, Eventual	< 15 minutes	< 15 minutes
>1	Single Master	Bounded Staleness	$K \& T$	< 15 minutes
>1	Single Master	Strong	0	< 15 minutes
>1	Multi-Master	Session, Consistent Prefix, Eventual	< 15 minutes	0
>1	Multi-Master	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.

## Strong consistency and multi-master

Cosmos accounts configured for multi-master 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 for using strong consistency with multi-master as any write into any region must be replicated and committed to all configured regions within the account. This results in the same write latency as a single master account.

## Next steps

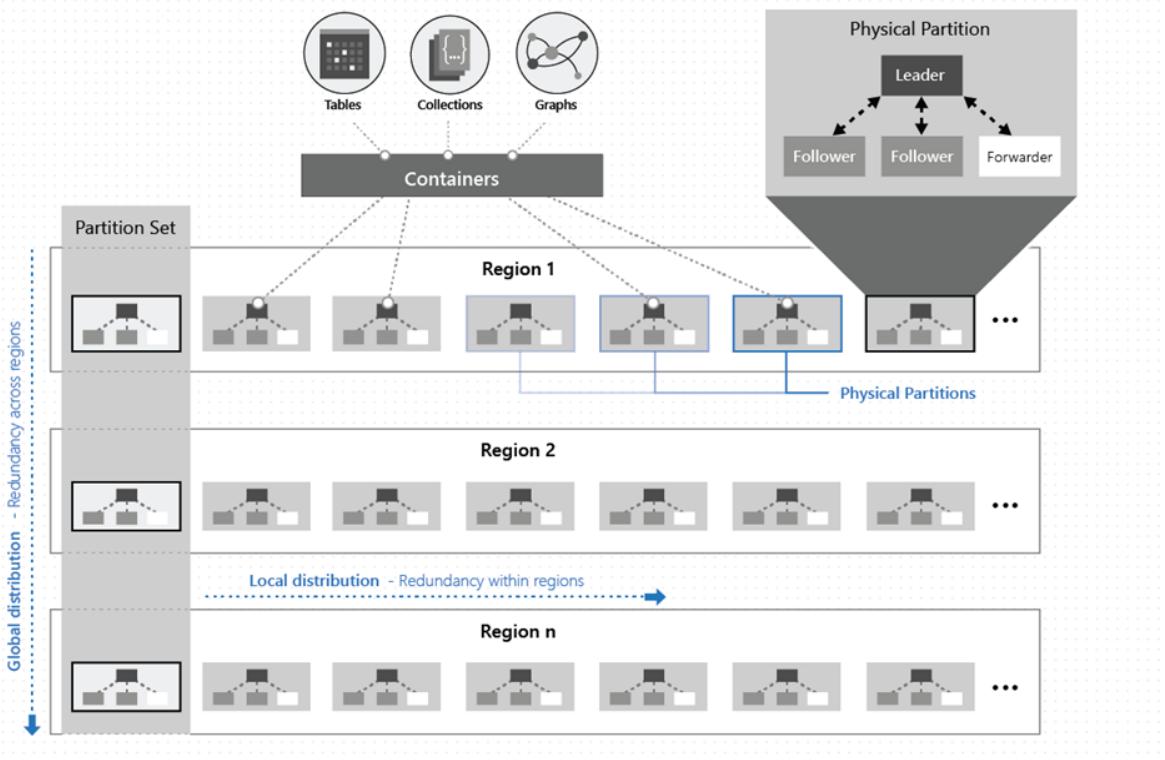
Learn more about global distribution and general consistency tradeoffs in distributed systems. See the following articles:

- [Consistency tradeoffs in modern distributed database systems design](#)
- [High availability](#)
- [Azure Cosmos DB SLA](#)

# High availability with Azure Cosmos DB

2/7/2020 • 9 minutes to read • [Edit Online](#)

Azure Cosmos DB transparently replicates your data across all the Azure regions associated with your Cosmos account. Cosmos DB employs multiple layers of redundancy for your data as shown in the following image:



- The data within Cosmos containers is [horizontally partitioned](#).
- 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 a Cosmos container and can accept writes and serve reads.

If your Cosmos account is distributed across  $N$  Azure regions, there will be at least  $N \times 4$  copies of all your data. In addition to providing low latency data access and scaling write/read throughput across the regions associated with your Cosmos account, having more regions (higher  $N$ ) further improves availability.

## SLAs for availability

As a globally distributed database, 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 Cosmos DB for single and multi-region accounts. For high availability, always configure your 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

OPERATION TYPE	SINGLE REGION	MULTI-REGION (SINGLE REGION WRITES)	MULTI-REGION (MULTI-REGION WRITES)
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 Cosmos DB in the event of regional outages

Regional outages aren't uncommon, and Azure Cosmos DB makes sure your database is always highly available. The following details capture Cosmos DB behavior during an outage, depending on your Cosmos account configuration:

- With 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.
- Multi-region accounts configured with multiple-write regions will be highly available for both writes and reads. Regional failovers are 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 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 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.
- Customers may also choose to use **manual failover** and monitor their Cosmos write endpoint URL's themselves using an agent built themselves. For customers with complex and sophisticated health monitoring needs, this can provide reduced RTO should a failure occur in the write region.
- When the previously impacted region is back online, any write data that was unreplicated 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 [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.

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

- During a read region outage, these accounts will remain highly available for reads and writes.
- 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. Eventually, when the impacted region is back online, the previously impacted read region 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 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 Cosmos account is configured with *Strong* consistency. In the event of a permanently irrecoverable write region, a multi-region 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 [multi-master replication](#) 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.

This feature is available in following Azure regions:

- UK South
- Southeast Asia
- East US
- East US 2
- Central US
- West Europe
- West US 2

### NOTE

Enabling Availability Zones for a single region Azure Cosmos account will result in charges that are equivalent to adding an additional region to your account. For details on pricing, see the [pricing page](#) and the [multi-region cost in Azure Cosmos DB](#) articles.

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
Write latency	Cross region	Cross region	Low
Regional outage – data loss	Data loss	Data loss	<p>Data loss</p> <p>When using bounded staleness consistency with multi master and more than one region, data loss is limited to the bounded staleness configured on your account</p> <p>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.</p>
Regional outage – availability	Availability loss	Availability loss	No availability loss
Throughput	X RU/s provisioned throughput	X RU/s provisioned throughput	<p>2X RU/s provisioned throughput</p> <p>This configuration mode requires twice the amount of throughput when compared to a single region with Availability Zones because there are two regions.</p>

#### NOTE

To enable Availability Zone support for a multi region Azure Cosmos account, the account must have multi-master 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:

```
$locations = @(
    @{
        "locationName"="Southeast Asia"; "failoverPriority"=0; "isZoneRedundant"= "true" },
    @{
        "locationName"="East US"; "failoverPriority"=1 }
)
```

The following command shows how to enable zone redundancy for the "EastUS" and "WestUS2" regions:

```
az cosmosdb create \
--name mycosmosdbaccount \
--resource-group myResourceGroup \
--kind GlobalDocumentDB \
--default-consistency-level Session \
--locations regionName=EastUS failoverPriority=0 isZoneRedundant=True \
--locations regionName=WestUS2 failoverPriority=1 isZoneRedundant=True
```

You can enable Availability Zones by using Azure portal when creating an Azure Cosmos account. When you create an account, make sure to enable the **Geo-redundancy, Multi-region Writes**, and choose a region where Availability Zones are supported:

Home > New > Create Azure Cosmos DB Account

## Create Azure Cosmos DB Account

Try Cosmos DB for free, up to 20K RU/s, for 30 days with unlimited renewals. [→](#)

[Basics](#) [Network](#) [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 <price>/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: Content Testing

\* Resource Group: cdbrg [Create new](#)

**Instance Details**

\* Account Name: myaccount1

\* API: Core (SQL)

Apache Spark: [Enable](#) [Disable](#)  
You're on the waitlist for Azure Cosmos with support for Apache Spark preview

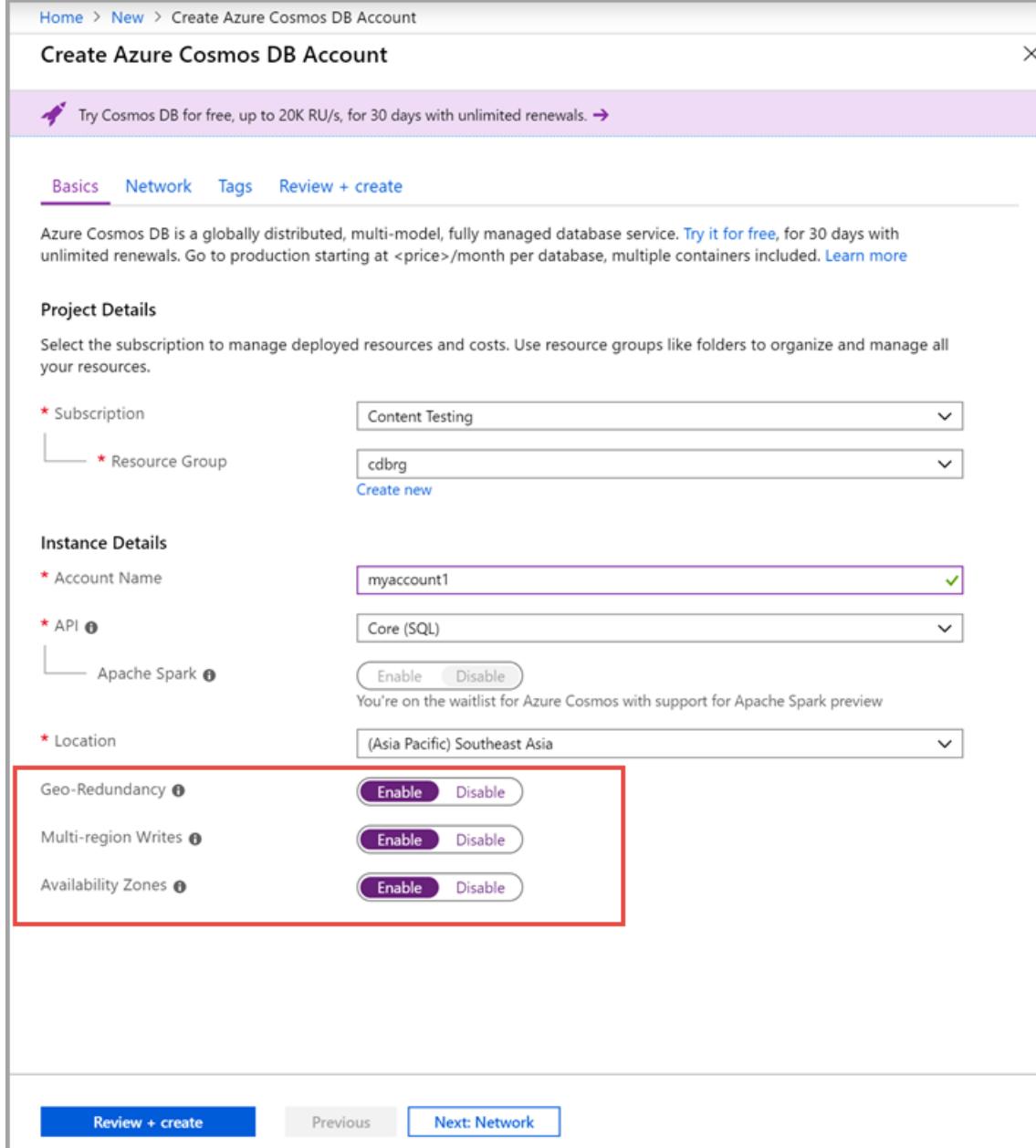
\* Location: (Asia Pacific) Southeast Asia

Geo-Redundancy: [Enable](#) [Disable](#)

Multi-region Writes: [Enable](#) [Disable](#)

Availability Zones: [Enable](#) [Disable](#)

[Review + create](#) [Previous](#) [Next: Network](#)



## Building highly available applications

- To ensure high write and read availability, configure your 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 Cosmos account with multiple write-regions](#).
- For multi-region 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 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 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

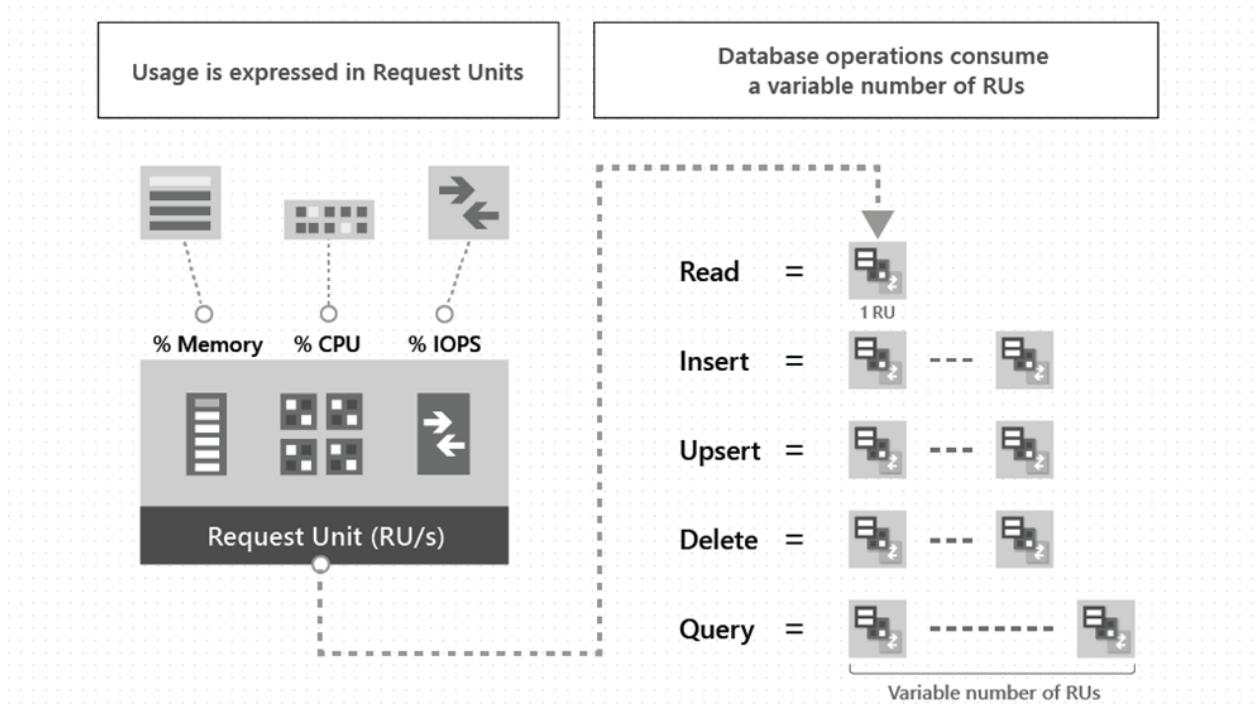
Next you can read the following articles:

- [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](#)

# Globally scale provisioned throughput

12/5/2019 • 2 minutes to read • [Edit Online](#)

In Azure Cosmos DB, provisioned throughput is represented as request units/second (RU/s or the plural form RUs). RUs measure the cost of both read and write operations against your Cosmos container as shown in the following image:



You can provision RUs on a Cosmos container or a Cosmos database. RUs provisioned on a container are exclusively available for the operations performed on that container. RUs provisioned on a database are shared among all the containers within that database (except for any containers with exclusively assigned RUs).

For elastically scaling provisioned throughput, you can increase or decrease the provisioned RU/s at any time. For more information, see [How-to provision throughput](#) and to elastically scale Cosmos containers and databases.

For globally scaling throughput, you can add or remove regions from your Cosmos account at any time. For more information, see [Add/remove regions from your database account](#). Associating multiple regions with a Cosmos account is important in many scenarios - to achieve low latency and [high availability](#) around the world.

## How provisioned throughput is distributed across 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. Each time you add a new region to your account, Cosmos DB automatically provisions ' $R$ ' RUs in the newly added region. The operations performed against your Cosmos container are guaranteed to get ' $R$ ' RUs in each region. 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, then:

- If the Cosmos account is configured with a single write region, the total RUs available globally on the container =  $R \times N$ .

- If the Cosmos account is configured with multiple write regions, the total RUs available globally on the container =  $R \times (N+1)$ . The additional  $R$  RUs are automatically provisioned to process update conflicts and anti-entropy traffic across the regions.

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

Next you can learn how to configure throughput on a container or database:

- [Get and set throughput for containers and databases](#)

# Conflict types and resolution policies

12/13/2019 • 2 minutes to read • [Edit Online](#)

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 are guaranteed to 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

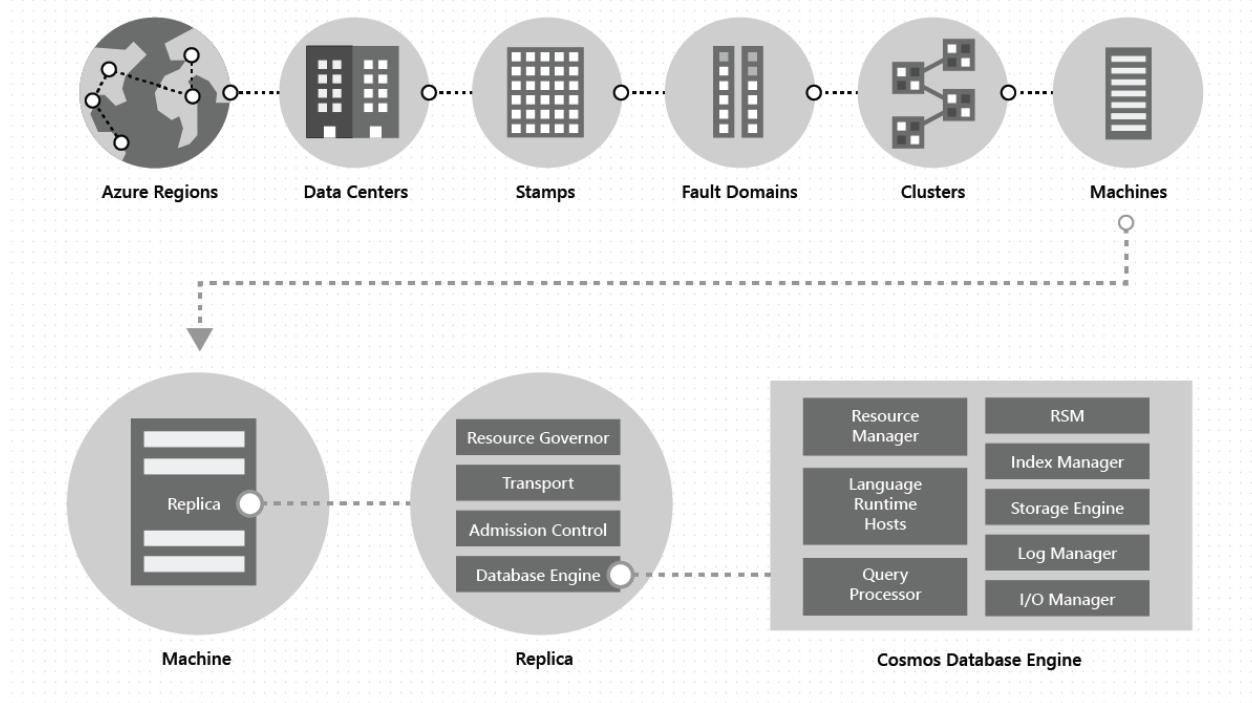
Learn how to configure conflict resolution policies:

- [How to configure multi-master in 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

12/5/2019 • 9 minutes to read • [Edit Online](#)

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:



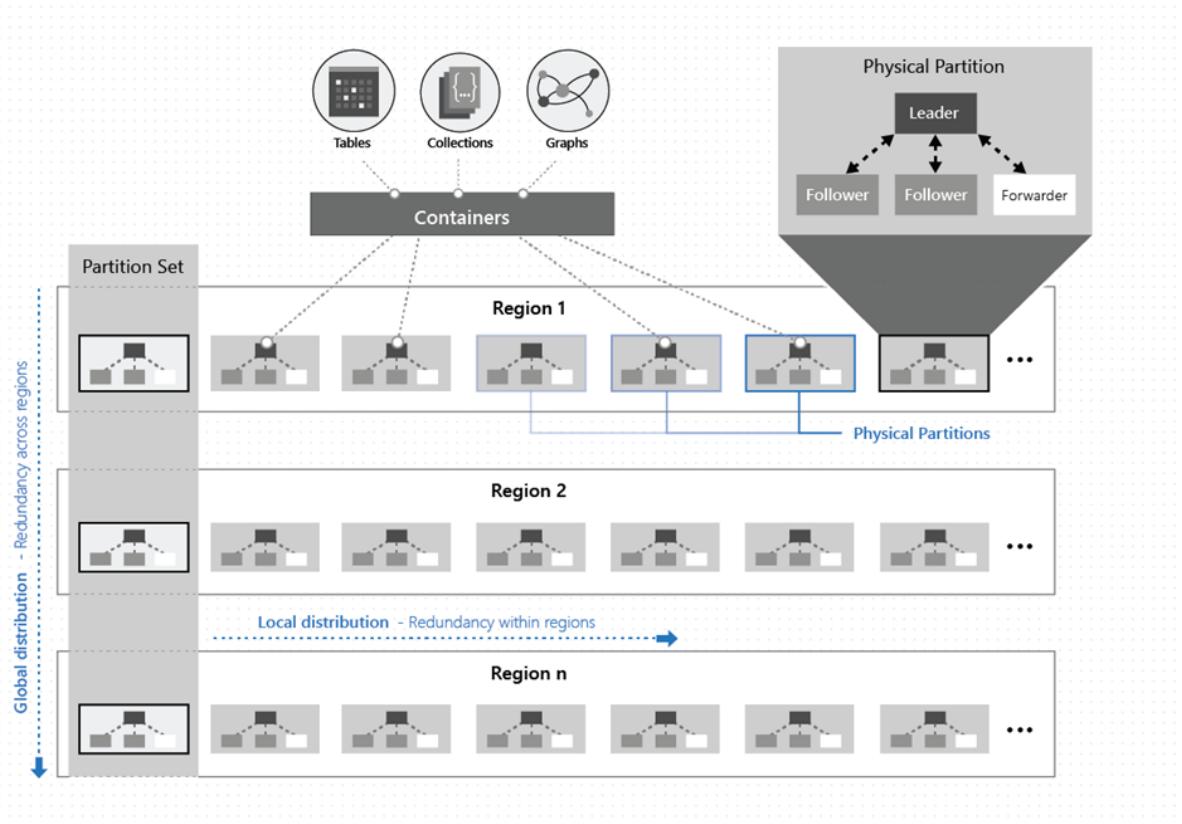
**Global distribution in Azure Cosmos DB is turnkey:** At any time, with a few clicks or programmatically with a single API call, you can add or remove the geographical regions associated with your Cosmos database. A Cosmos database, in turn, consists of a set of Cosmos containers. In Cosmos DB, containers 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.

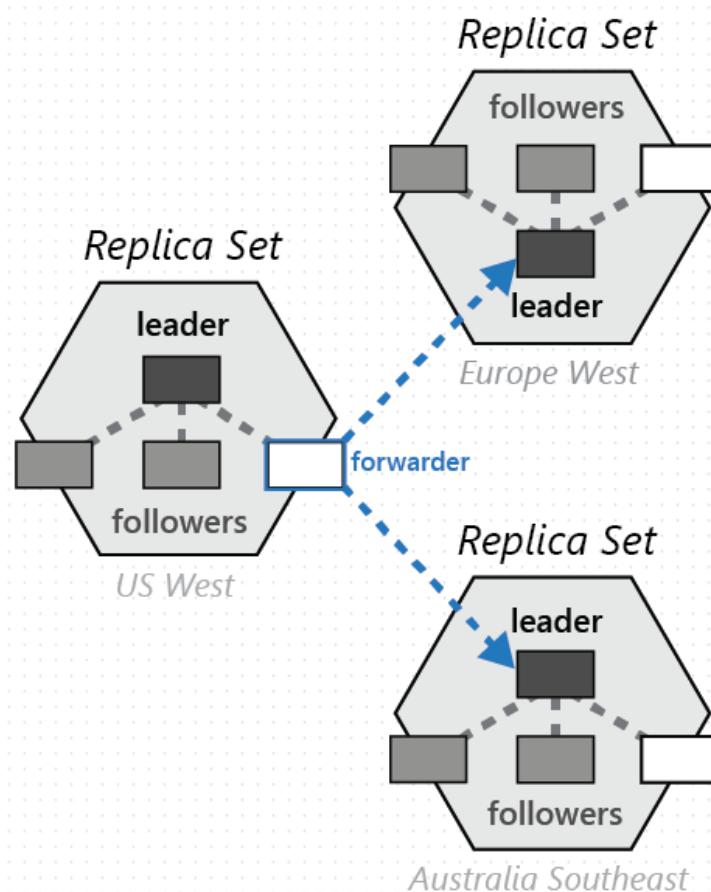
## Replica-sets

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  $N_{Min}$  and  $N_{Max}$  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-master) 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](#)

# Regional presence with Azure Cosmos DB

10/21/2019 • 2 minutes to read • [Edit Online](#)

Azure Cosmos DB is a foundational service in Azure, and, by default, is always available in all regions, where Azure is available. Currently, Azure is available in [54 regions](#) worldwide.



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.

## Regional presence with global distribution

All APIs exposed by Azure Cosmos DB (including SQL, MongoDB, Cassandra, Gremlin, and Table) are available in all Azure regions by default. For example, you can have MongoDB and Cassandra APIs exposed by Azure Cosmos DB not only in all of the global Azure regions, but also in sovereign clouds like China, Germany, Government, and Department of Defense (DoD) regions.

Azure Cosmos DB is a [globally distributed](#) database service. 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. With the turnkey global distribution capability and multi-mastered replication protocol, Azure Cosmos DB offers less than 10 ms read and write latencies at the 99th percentile, 99.999 read and write availability, and ability to elastically scale provisioned throughput for reads and

writes across all the regions associated with your Azure Cosmos account. Azure Cosmos DB, also offers five well-defined consistency models and you can choose to apply a specific consistency model to your data. Finally, Azure Cosmos DB is the only database service in the industry that provides a comprehensive [Service Level Agreement \(SLA\)](#) encompassing provisioned throughput, latency at the 99th percentile, high availability, and consistency. The above capabilities are available in all Azure clouds.

## Next steps

You can now learn about core concepts of Azure Cosmos DB with the following articles:

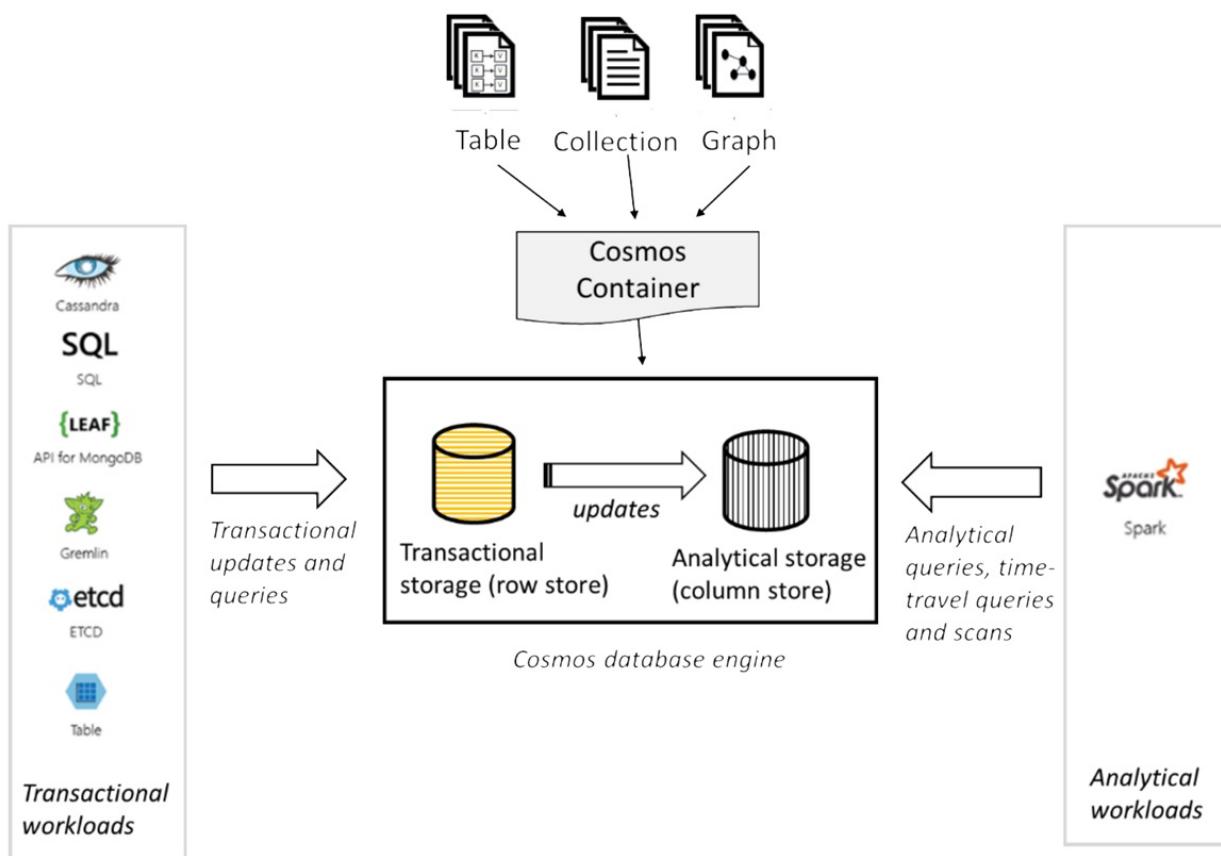
- [Global data distribution](#)
- [How to manage an Azure Cosmos DB account](#)
- [Provision throughput for Azure Cosmos containers and databases](#)
- [Azure Cosmos DB SLA](#)

# Globally distributed transactional and analytical storage for Azure Cosmos containers

2/26/2020 • 3 minutes to read • [Edit Online](#)

Azure Cosmos container is internally backed by two storage engines - transactional storage engine and an updatable analytical storage engine (in private preview). Both the storage engines are log-structured and write-optimized for faster updates. However, each of them is encoded differently:

- **Transactional storage engine** – It is encoded in row-oriented format for fast transactional reads and queries.
- **Analytical storage engine** - It is encoded in columnar format for fast analytical queries and scans.



The transactional storage engine is backed by local SSDs, whereas the analytical storage is stored on an inexpensive off-cluster SSD storage. The following table captures the notable differences between the transactional and the analytical storage.

FEATURE	TRANSACTIONAL STORAGE	ANALYTICAL STORAGE
Maximum storage per an Azure Cosmos container	Unlimited	Unlimited
Maximum storage per a logical partition key	20 GB	Unlimited

FEATURE	TRANSACTIONAL STORAGE	ANALYTICAL STORAGE
Storage encoding	Row-oriented, using an internal format.	Column-oriented, using Apache Parquet format.
Storage locality	Replicated storage backed by local/intra-cluster SSDs.	Replicated storage backed by inexpensive remote/off-cluster SSDs.
Durability	99.99999 (7-9 s)	99.99999 (7-9 s)
APIs that access the data	SQL, MongoDB, Cassandra, Gremlin, Tables, and etcd.	Apache Spark
Retention (Time-to-live or TTL)	Policy driven, configured on the Azure Cosmos container by using the <code>DefaultTimeToLive</code> property.	Policy driven, configured on the Azure Cosmos container by using the <code>ColumnStoreTimeToLive</code> property.
Price per GB	See the <a href="#">pricing page</a>	See the <a href="#">pricing page</a>
Price for storage transactions	See the <a href="#">pricing page</a>	See the <a href="#">pricing page</a>

## Benefits of transactional and analytical storage

### No ETL operations

Traditional analytical pipelines are complex with multiple stages each requiring Extract-Transform-Load (ETL) operations to and from the compute and storage tiers. It results in complex data processing architectures. Which means high costs for multiple stages of storage and compute, and high latency due to massive volumes of data transferred between various stages of storage and compute.

There is no overhead of performing ETL operations with Azure Cosmos DB. Each Azure Cosmos container is backed by both transactional and analytical storage engines, and the data transfer between the transactional and analytical storage engine is done within the database engine, and without any network hops. The resulting latency and cost are significantly lower compared to traditional analytical solutions. And you get a single globally distributed storage system for both transactional and analytical workloads.

### Store multiple versions, update, and query the analytical storage

The analytical storage is fully updatable, and it contains the complete version history of all the transactional updates that occurred on the Azure Cosmos container.

Any update made to the transactional storage is guaranteed to be visible to the analytical storage within 30 seconds.

### Globally distributed, multi-master analytical storage

If your Azure Cosmos account is scoped to a single region, the data stored (in transactional and analytical storage) in the containers is also scoped to a single region. On the other hand, if the Azure Cosmos account is globally distributed, the data stored in the containers is also globally distributed.

For Azure Cosmos accounts configured with multiple write regions, writes to the container (to both the transactional and the analytical storage) are always performed in the local region and hence they are fast.

For both single or multi-region Azure Cosmos accounts, regardless if single write region (single master) or multiple write regions (also known as multi-master), both transactional and analytical reads/queries are performed locally in the given region.

### Performance isolation between transactional and analytical workloads

In a given region, the transactional workloads operate against your container's transactional/row storage. On the other hand, the analytical workloads operate against your container's analytical/column storage. The two storage engines operate independently and provide strict performance isolation between the workloads.

The transactional workloads consume the provisioned throughput (RUs). Unlike the transactional workloads, the analytical workloads throughput is based on the actual consumption. The analytical workloads consume resources on-demand.

## Next steps

- [Time to live in Azure Cosmos DB](#)

# Partitioning in Azure Cosmos DB

2/26/2020 • 2 minutes to read • [Edit Online](#)

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 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.

## 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 servers.

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.

Queries that access data within a single logical partition are more cost-effective than queries that access multiple partitions. Transactions (in stored procedures or triggers) are allowed only against items in a single logical partition.

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

## Choosing a partition key

The following is a good guidance for choosing a partition key:

- A single logical partition has an upper limit of 20 GB of storage.
- Azure Cosmos containers have a minimum throughput of 400 request units per second (RU/s). When throughput is provisioned on a database, minimum RUs per container is 100 request units per second (RU/s). Requests to the same partition key can't exceed the throughput that's allocated to a partition. If requests exceed the allocated throughput, requests are rate-limited. So, it's important to pick a partition key that doesn't result in "hot spots" within your application.
- Choose a partition key that has a wide range of values and access patterns that are evenly spread across logical partitions. This helps 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.

- Choose a partition key that spreads the workload evenly across all partitions and evenly over time. Your choice of partition key should balance the need for efficient partition queries and transactions against the goal of distributing items across multiple partitions to achieve scalability.
- Candidates for partition keys might include properties that appear frequently as a filter in your queries. Queries can be efficiently routed by including the partition key in the filter predicate.

## 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](#).

# Partitioning and horizontal scaling in Azure Cosmos DB

10/21/2019 • 2 minutes to read • [Edit Online](#)

This article explains physical and logical partitions in Azure Cosmos DB. It also discusses best practices for scaling and partitioning.

## Logical partitions

A logical partition consists of a set of items that have the same partition key. For example, in a container where all items contain a `City` property, you can use `City` as the partition key for the container. Groups of items that have specific values for `City`, such as `London`, `Paris`, and `NYC`, form distinct logical partitions. You don't have to worry about deleting a partition when the underlying data is deleted.

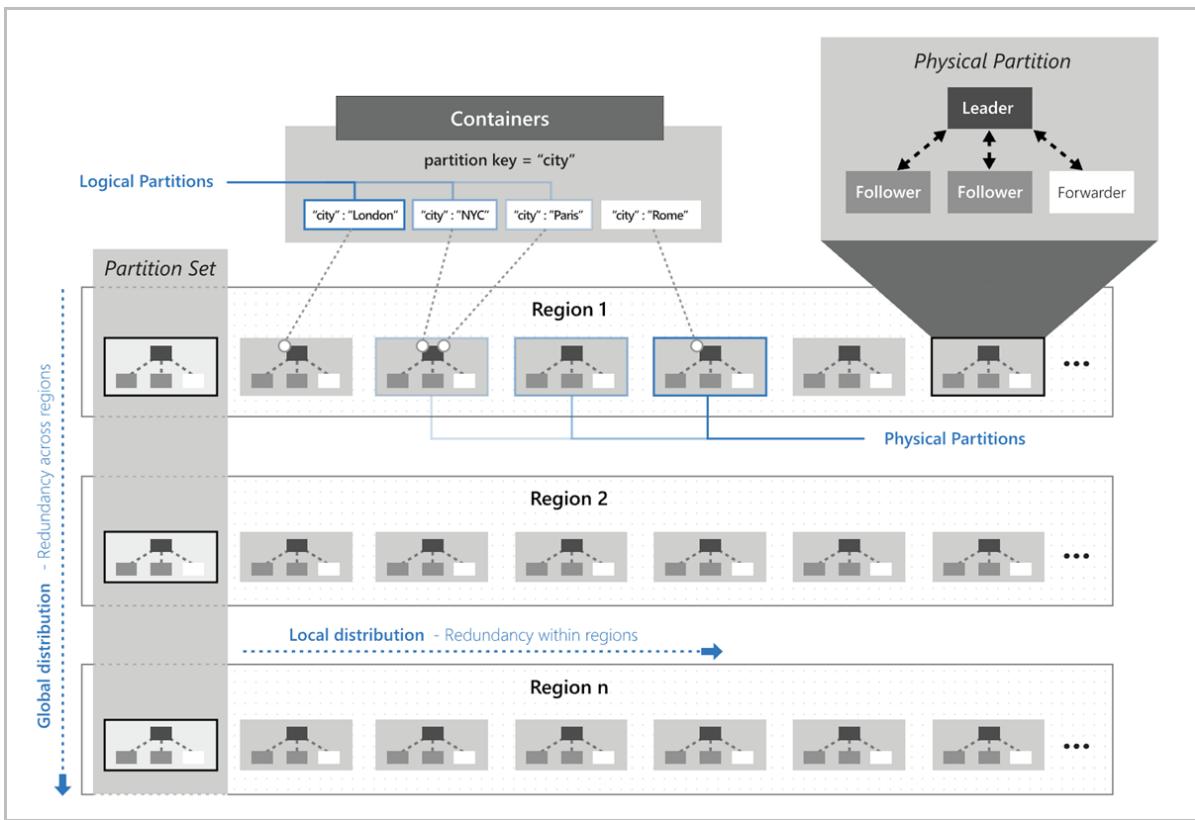
In Azure Cosmos DB, a container is the fundamental unit of scalability. Data that's added to the container and the throughput that you provision on the container are automatically (horizontally) partitioned across a set of logical partitions. Data and throughput are partitioned based on the partition key you specify for the Azure Cosmos container. For more information, see [Create an Azure Cosmos container](#).

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.

## Physical partitions

An Azure Cosmos container is scaled by distributing data and throughput across a large number of logical partitions. Internally, one or more logical partitions are mapped to a physical partition that consists of a set of replicas, also referred to as a [replica set](#). Each replica set hosts an instance of the Azure Cosmos database engine. A replica set makes the data stored within the physical partition durable, highly available, and consistent. A physical partition supports the maximum amount of storage and request units (RUs). 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.

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



Throughput provisioned for a container is divided evenly among physical partitions. A partition key design that doesn't distribute the throughput requests evenly might create "hot" partitions. Hot partitions might result in rate-limiting and in inefficient use of the provisioned throughput, and higher costs.

Unlike logical partitions, physical partitions are an internal implementation of the system. You can't control the size, placement, or count of physical partitions, and you can't control the mapping between logical partitions and physical partitions. However, you can control the number of logical partitions and the distribution of data, workload and throughput by [choosing the right logical partition key](#).

## Next steps

- Learn about [choosing a partition key](#).
- 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](#).

# Create a synthetic partition key

12/13/2019 • 3 minutes to read • [Edit Online](#)

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](#).

# Request Units in Azure Cosmos DB

11/22/2019 • 3 minutes to read • [Edit Online](#)

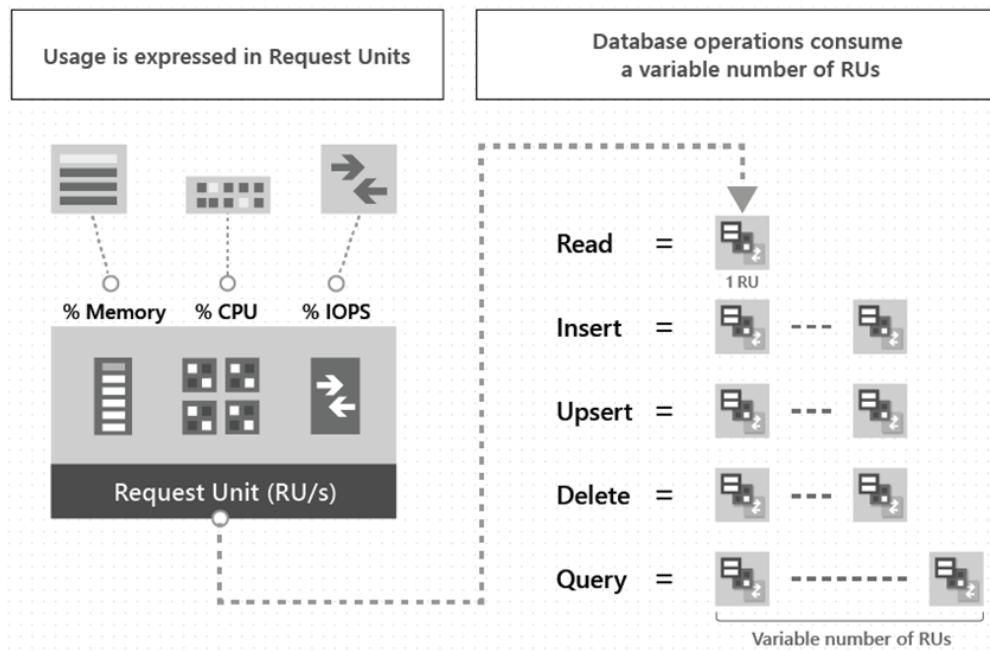
With Azure Cosmos DB, you pay for the throughput you provision and the storage you consume on an hourly basis. Throughput must be provisioned to ensure that sufficient system resources are available for your Azure Cosmos database at all times. You need enough resources to meet or exceed the [Azure Cosmos DB SLAs](#).

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 per second as the currency for throughput. RUs per second is a rate-based currency. It abstracts 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 read a 1 KB item is 1 Request Unit (or 1 RU). A minimum of 10 RU/s is required to store each 1 GB of data. 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, 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.

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. You can scale 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.

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](#).

## Request Unit considerations

While you estimate the number of RUs per second to provision, 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.
- **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

Azure Cosmos DB guarantees that the same query on the same data 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.

## Next steps

- Learn more about how to [provision throughput on Azure Cosmos containers and databases](#).
- Learn more about [logical partitions](#).
- Learn more about how to [globally scale provisioned throughput](#).
- 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](#).

# Provision throughput on containers and databases

2/26/2020 • 7 minutes to read • [Edit Online](#)

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 throughput on a container, see [Provision throughput on an Azure Cosmos container](#).

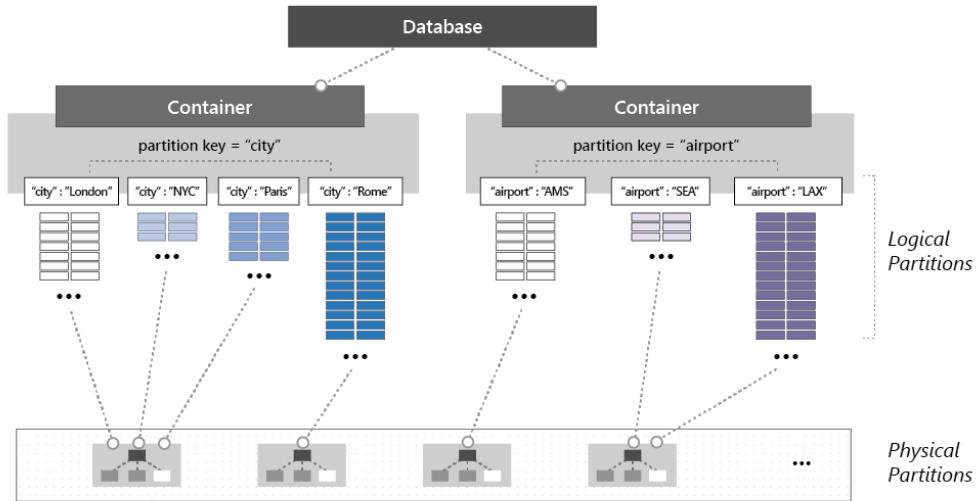
Setting provisioned throughput on a container is the most frequently used option. You can elastically scale throughput for a container by provisioning any amount of throughput by using [Request Units \(RUs\)](#).

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 that logical partition, your operations get rate-limited. When rate-limiting occurs, you can either increase the provisioned throughput for the entire container or retry the operations. For more information on partitioning, see [Logical partitions](#).

We recommend that you configure throughput at the container granularity when you want guaranteed 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](#).

Setting throughput on an Azure Cosmos database guarantees that you receive the provisioned throughput for that database all the time. 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

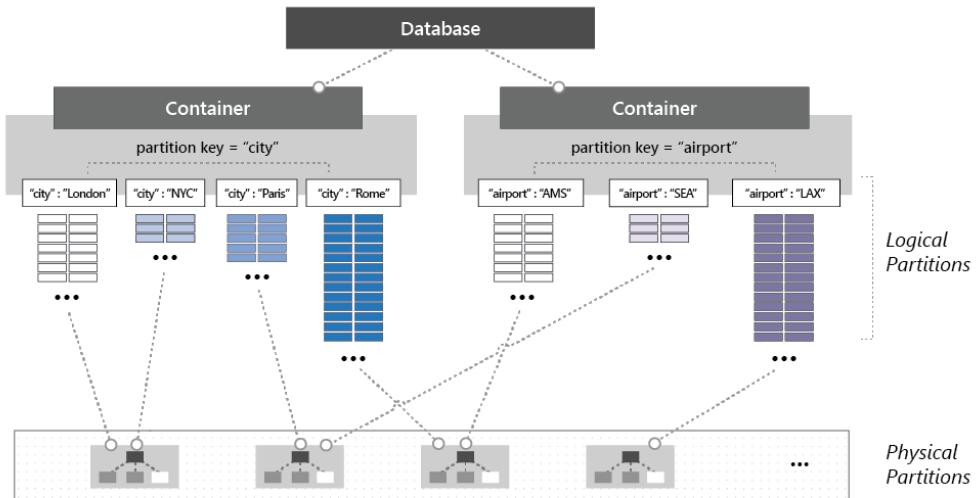
for the entire database or retry the operations. For more information on partitioning, see [Logical partitions](#).

Containers in a shared throughput database share the throughput (RU/s) allocated to that database. You can have up to four containers with a minimum of 400 RU/s on the database. Each new container after the first four will require an additional 100 RU/s minimum. For example, if you have a shared throughput database with eight containers, the minimum RU/s on the database will be 800 RU/s.

**NOTE**

In a shared throughput database, you can have a maximum of 25 containers in the database. If you already have more than 25 containers in a shared throughput database, you will not be able to create additional containers until the container count is less than 25.

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 throughput on an Azure Cosmos database and a container:

- You can create an Azure Cosmos database named Z with 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. Note that you can configure shared and dedicated throughput only when creating the database and container.

\* Container id ⓘ  
B

\* Partition key ⓘ  
/id

My partition key is larger than 100 bytes

Provision dedicated throughput for this container ⓘ

\* Throughput (400 - 100,000 RU/s) ⓘ  
400 - +

Estimated spend (USD): \$0.58 hourly / \$13.82 daily (8 regions, 400RU/s, \$0.00016/RU)

- 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. The minimum provisioned throughput depends on the following factors:

- The maximum data size that you ever store in the container
- The maximum throughput that you ever provision on the container
- The maximum number of Azure Cosmos containers that you ever create in a database with shared throughput.

You can retrieve the minimum throughput of a container or a database programmatically by using the SDKs or view the value in the Azure portal. When using the .NET SDK, the [DocumentClient.ReplaceOfferAsync](#) method allows you to scale the provisioned throughput value. When using the Java SDK, the [RequestOptions.setOfferThroughput](#) method allows you to scale the provisioned throughput value.

When using the .NET SDK, the [DocumentClient.ReadOfferAsync](#) method allows you to retrieve the minimum throughput of a container or a database.

You can scale the provisioned throughput of a container or a database at any time. When a scale operation is performed to increase the throughput, it can take longer time due to the system tasks to provision the required resources. You can check the status of the scale operation in Azure portal or programmatically using the SDKs. When using the .Net SDK, you can get the status of the scale operation by using the [DocumentClient.ReadOfferAsync](#) method.

## Comparison of models

PARAMETER	THROUGHPUT PROVISIONED ON A DATABASE	THROUGHPUT PROVISIONED ON A CONTAINER
Minimum RUs	400 (After the first four containers, each additional container requires a minimum of 100 RUs per second.)	400
Minimum RUs per container	100	400
Maximum RUs	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.
Maximum storage for a container	Unlimited.	Unlimited.
Maximum throughput per logical partition of a container	10K RUs	10K RUs
Maximum storage (data + index) per logical partition of a container	20 GB	20 GB

## Next steps

- 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](#).

# Create Azure Cosmos containers and databases in autopilot mode (Preview)

1/17/2020 • 5 minutes to read • [Edit Online](#)

Azure Cosmos DB allows you to provision throughput on your containers in either manual or autopilot mode. This article describes the benefits and use cases of autopilot mode.

## NOTE

Autopilot mode is currently available in public preview. You can [enable autopilot for new databases and containers](#) only. It is not available for existing containers and databases.

In addition to manual provisioning of throughput, you can now configure Azure Cosmos containers in autopilot mode. Containers and databases configured in autopilot mode will **automatically and instantly scale the provisioned throughput based on your application needs without impacting the availability, latency, throughput, or performance of the workload globally**.

When configuring containers and databases in autopilot mode, you need to specify the maximum throughput  $T_{max}$  not to be exceeded. Containers can then scale their throughput so that  $0.1*T_{max} < T < T_{max}$ . In other words, containers and databases scale instantly based on the workload needs, from as low as 10% of the maximum throughput value that you have configured up to the configured maximum throughput value. You can change the maximum throughput ( $T_{max}$ ) setting on an autopilot database or container at any point in time. With the autopilot option, the 400 RU/s minimum throughput per container or database is no longer applicable.

During the preview of autopilot, for the specified maximum throughput on the container or the database, the system allows operating within the calculated storage limit. If the storage limit is exceeded, then the maximum throughput is automatically adjusted to a higher value. When using database level throughput with autopilot mode, the number of containers allowed within a database is calculated as:  $0.001*T_{Max}$ . For example, if you provision 20,000 autopilot RU/s, then the database can have 20 containers.

## Benefits of autopilot mode

Azure Cosmos containers that are configured in autopilot mode have the following benefits:

- **Simple:** Containers in autopilot mode remove the complexity to manage provisioned throughput (RUs) and capacity manually for various containers.
- **Scalable:** Containers in autopilot mode seamlessly scale the provisioned throughput capacity as needed. There is no disruption to client connections, applications and they don't impact any existing SLAs.
- **Cost-effective:** When you use containers configured in autopilot mode, you only pay for the resources that your workloads need on a per-hour basis.
- **Highly available:** Containers in autopilot mode use the same globally distributed, fault-tolerant, highly available backend to ensure data durability and high availability.

## Use cases of autopilot mode

The use cases for Azure Cosmos containers configured in autopilot mode include:

- **Variable workloads:** When you are running a lightly used application with peak usage of 1 hour to several

hours a few times each day or several times per year. Examples include applications for human resources, budgeting, and operational reporting. For such scenarios, containers configured in autopilot mode can be used, and you no longer need to manually provision for either peak or average capacity.

- **Unpredictable workloads:** When you are running workloads where there is database usage throughout the day, but also peaks of activity that are hard to predict. An example includes a traffic site that sees a surge of activity when weather forecast changes. Containers configured in autopilot mode adjust the capacity to meet the needs of the application's peak load and scale back down when the surge of activity is over.
- **New applications:** If you are deploying a new application and are unsure about how much provisioned throughput (i.e., how many RUs) you need. With containers configured in autopilot mode, you can automatically scale to the capacity needs and requirements of your application.
- **Infrequently used applications:** If you have an application that is only used for a few hours several times per day or week or month, such as a low-volume application/web/blog site.
- **Development and test databases:** If you have developers using containers during work hours but don't need them on nights or weekends. With containers configured in autopilot mode, they scale down to a minimum when not in use.
- **Scheduled production workloads/queries:** When you have a series of scheduled requests/operations/queries on a single container, and if there are idle periods where you want to run at an absolute low throughput, you can now do that easily. When a scheduled query/request is submitted to a container configured in autopilot mode, it will automatically scale up as much as needed and run the operation.

Solutions to the previous problems not only require an enormous amount of time in implementation, but they also introduce complexity in configuration or your code, and frequently require manual intervention to address them. Autopilot mode enables the above scenarios out of the box, so that you do not need to worry about these problems anymore.

## Comparison – Containers configured in manual mode vs. autopilot mode

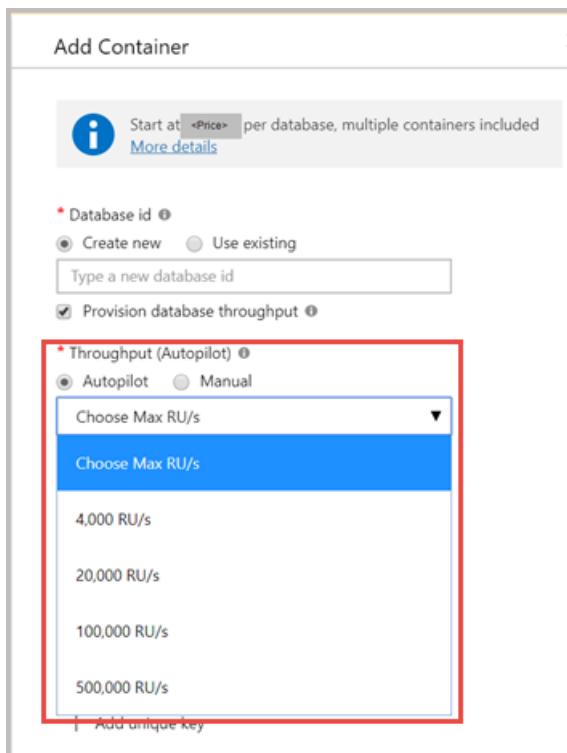
	CONTAINERS CONFIGURED IN MANUAL MODE	CONTAINERS CONFIGURED IN AUTOPILOT MODE
<b>Provisioned throughput</b>	Manually provisioned.	Automatically and instantaneously scaled based on the workload usage patterns.
<b>Rate-limiting of requests/operations (429)</b>	May happen, if consumption exceeds provisioned capacity.	Will not happen if the throughput consumed is within the max throughput that you choose with autopilot mode.
<b>Capacity planning</b>	You have to do an initial capacity planning and provision of the throughput you need.	You don't have to worry about capacity planning. The system automatically takes care of capacity planning and capacity management.

	CONTAINERS CONFIGURED IN MANUAL MODE	CONTAINERS CONFIGURED IN AUTOPILOT MODE
Pricing	Manually provisioned RU/s per hour.	For single write region accounts, you pay for the throughput used on an hourly basis, by using the autopilot RU/s per hour rate.  For accounts with multiple write regions, there is no extra charge for autopilot. You pay for the throughput used on hourly basis using the same multi-master RU/s per hour rate.
Best suited for workload types	Predictable and stable workloads	Unpredictable and variable workloads

## Create a database or a container with autopilot mode

You can configure autopilot for new databases or containers when creating them through the Azure portal. Use the following steps to create a new database or container, enable autopilot, and specify the maximum throughput (RU/s).

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 **Autopilot** option, and choose the maximum throughput (RU/s) that the database or container cannot exceed when using the autopilot option.



4. Select **OK**.

You can create a shared throughput database with autopilot mode by selecting the **Provision database throughput** option.

## Throughput and storage limits for autopilot

The following table shows the maximum throughout and storage limits for different options in autopilot mode:

MAXIMUM THROUGHPUT LIMIT	MAXIMUM STORAGE LIMIT
4000 RU/s	50 GB
20,000 RU/s	200 GB
100,000 RU/s	1 TB
500,000 RU/s	5 TB

## Next steps

- Review the [autopilot FAQ](#).
- 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](#).

# Frequently asked questions about provisioned throughput in Azure Cosmos DB autopilot mode (Preview)

12/19/2019 • 6 minutes to read • [Edit Online](#)

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

## Frequently asked questions

### **Is autopilot mode supported for all APIs?**

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

### **Is autopilot mode supported for multi-master accounts?**

Yes, autopilot mode is supported for multi-master accounts. The max RU/s are available in each region that is added to the Cosmos account.

### **What is the pricing for autopilot?**

Refer to the Azure Cosmos DB [pricing page](#) for details.

### **How do I enable autopilot for my containers or databases?**

Autopilot mode can be enabled on new containers and databases created using the Azure portal.

### **Is there CLI or SDK support to create containers or databases with autopilot mode?**

Currently, in the preview release, you can only create resources with autopilot mode from the Azure portal. Support for CLI and SDK is not yet available.

### **Can I enable autopilot on an existing container or a database?**

Currently, you can enable autopilot on new containers and databases when creating them. Support to enable autopilot mode on existing containers and databases is not yet available. You can migrate existing containers to a new container using [Azure Data Factory](#) or [change feed](#).

### **Can I turn off autopilot mode on a container or database?**

Yes, you can turn off autopilot by switching to the 'Manual' option for the provisioned throughput. In the preview release, after switching from autopilot mode to manual mode, you cannot enable autopilot again for the same resource.

### **Is autopilot mode supported for shared throughput databases?**

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

### **What is the number of allowed collections per shared throughput database when autopilot is enabled?**

For shared throughput databases with autopilot mode enabled, the number of allowed collections is: MIN(25, Max RU/s of database / 1000). For example, if the max throughput of the database is 20,000 RU/s, the database can have MIN(25, 20,000 RU/s / 1000) = 20 collections.

### **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 [autopilot 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 tier that can support that level of storage. For example, suppose a database or container is provisioned with the 4000 RU/s max throughput option, which has a storage limit of 50 GB. If the storage of the resource increases to 100 GB, the max RU/s of the database or container will be automatically increased to 20,000 RU/s, which can support up to 200 GB.

### **How quickly will autopilot scale up and down based on spikes in traffic?**

Autopilot will instantaneously scale up or scale down the RU/s within the minimum and maximum RU/s range, based on incoming traffic. Billing is done at a 1-hour granularity, where you are charged for the highest RU/s in a particular hour.

### **Can I specify a custom max throughput (RU/s) value for autopilot mode?**

Currently, during the preview release, you can select between [four options](#) for max throughput (RU/s).

### **Can I increase the max RU/s (move to a higher tier) on the database or container?**

Yes. From the **Scale & Settings** option for your container, or **Scale** option for your database, you can select a higher max RU/s for autopilot. This is an asynchronous scale-up operation that may take sometime to complete (typically 4-6 hours, depending on the RU/s selected) as the service provisions more resources to support the higher scale.

### **Can I reduce the max RU/s (move to a lower tier) on the database or container?**

Yes. As long as the current storage of the database or container is below the [storage limit](#) associated with the max RU/s tier you want to scale down to, you can reduce the max RU/s to that tier. For example, if you have selected 20,000 RU/s as the max RU/s, you can scale down the max RU/s to 4000 RU/s if you have less than 50 GB of storage (the storage limit associated with 4000 RU/s).

### **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 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 tier 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 container or database, 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 autopilot 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

container or database, 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 autopilot on an Azure Cosmos container or database](#).
- Learn about the [benefits of autopilot](#).
- Learn more about [logical and physical partitions](#).

# Getting started with SQL queries

12/5/2019 • 3 minutes to read • [Edit Online](#)

Azure Cosmos DB SQL API accounts support querying items using Structured Query Language (SQL) as a JSON query language. The design goals of the Azure Cosmos DB query language are to:

- Support SQL, one of the most familiar and popular query languages, instead of inventing a new query language. SQL provides a formal programming model for rich queries over JSON items.
- Use JavaScript's programming model as the foundation for the query language. JavaScript's type system, expression evaluation, and function invocation are the roots of the SQL API. These roots provide a natural programming model for features like relational projections, hierarchical navigation across JSON items, self-joins, spatial queries, and invocation of user-defined functions (UDFs) written entirely in JavaScript.

## Upload sample data

In your SQL API Cosmos DB account, create a container called `Families`. Create two simple JSON items in the container. You can run most of the sample queries in the Azure Cosmos DB query docs using this data set.

### Create JSON items

The following code creates two simple JSON items about families. The simple JSON items for the Andersen and Wakefield families 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

2/19/2020 • 3 minutes to read • [Edit Online](#)

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"
  }
}]
```

### Quoted property accessor

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.

```
SELECT f["lastName"]
FROM Families f
WHERE f["id"] = "AndersenFamily"
```

### Nested properties

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"
}]
```

## 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"
  }
}]
```

## Reserved keywords and special characters

If your data contains properties with the same names as reserved keywords such as "order" or "Group" then the queries against these documents will result in syntax errors. You should explicitly include the property in `[]` character to run the query successfully.

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
```

## Next steps

- [Getting started](#)
- [Azure Cosmos DB .NET samples](#)
- [WHERE clause](#)

# FROM clause in Azure Cosmos DB

2/24/2020 • 3 minutes to read • [Edit Online](#)

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

2/4/2020 • 2 minutes to read • [Edit Online](#)

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.

## 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.

## Next steps

- [Getting started](#)
- [IN keyword](#)
- [FROM clause](#)

# ORDER BY clause in Azure Cosmos DB

2/12/2020 • 4 minutes to read • [Edit Online](#)

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.

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"  
  }  
]
```

## Next steps

- [Getting started](#)
- [Indexing policies in Azure Cosmos DB](#)
- [OFFSET LIMIT clause](#)

# GROUP BY clause in Azure Cosmos DB

11/7/2019 • 2 minutes to read • [Edit Online](#)

The GROUP BY clause divides the query's results according to the values of one or more specified properties.

## NOTE

Azure Cosmos DB currently supports GROUP BY in .NET SDK 3.3 and above as well as JavaScript SDK 3.4 and above. Support for other language SDK's is not currently available but is planned.

## 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 system 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)

## 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):

```
[{
  "foodGroup": "Fast Foods",
  "foodGroupCount": 371
},
{
  "foodGroup": "Finfish and Shellfish Products",
  "foodGroupCount": 267
},
{
  "foodGroup": "Meals, Entrees, and Side Dishes",
  "foodGroupCount": 113
},
{
  "foodGroup": "Sausages and Luncheon Meats",
  "foodGroupCount": 244
}]
```

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:

```
[{
  "version": 1,
  "foodGroup": "Nut and Seed Products",
  "foodGroupCount": 133
},
{
  "version": 1,
  "foodGroup": "Finfish and Shellfish Products",
  "foodGroupCount": 267
},
{
  "version": 1,
  "foodGroup": "Fast Foods",
  "foodGroupCount": 371
},
{
  "version": 1,
  "foodGroup": "Sausages and Luncheon Meats",
  "foodGroupCount": 244
}]
```

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": 371,
  "upperFoodGroup": "FAST FOODS"
},
{
  "foodGroupCount": 267,
  "upperFoodGroup": "FINFISH AND SHELLFISH PRODUCTS"
},
{
  "foodGroupCount": 389,
  "upperFoodGroup": "LEGUMES AND LEGUME PRODUCTS"
},
{
  "foodGroupCount": 113,
  "upperFoodGroup": "MEALS, ENTREES, AND SIDE DISHES"
}]
```

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:

```
[{
  "correctVersion": true,
  "containsOrangeTag": false,
  "foodGroupCount": 8608
},
{
  "correctVersion": true,
  "containsOrangeTag": true,
  "foodGroupCount": 10
}]
```

## Next steps

- [Getting started](#)
- [SELECT clause](#)
- [Aggregate functions](#)

# OFFSET LIMIT clause in Azure Cosmos DB

1/28/2020 • 2 minutes to read • [Edit Online](#)

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 documents 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 document, including those that are skipped. The performance advantage is a savings in client resources by avoiding processing documents 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](#)

# SQL subquery examples for Azure Cosmos DB

12/5/2019 • 12 minutes to read • [Edit Online](#)

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  $\langle c, t, n, s \rangle$  tuple. For instance, if a matching document had 10 items in each of the three arrays, it will expand to  $1 \times 10 \times 10 \times 10$  (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  $1 \times 1 \times 5 \times 5 = 25$  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: 0.00000001, baseUnit: 'gram'},
        {unit: 'µg', name: 'microgram', multiplier: 0.00001, 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: 0.00000001, 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": 94.78899999999987 } }  
]
```

### 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

12/5/2019 • 5 minutes to read • [Edit Online](#)

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 ( $1 \times 1 \times 1$ ) 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  $1 \times 1 \times 2 = 2$  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](#)

# Aliasing in Azure Cosmos DB

12/5/2019 • 2 minutes to read • [Edit Online](#)

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"
    }
}]
```

## Next steps

- [Azure Cosmos DB .NET samples](#)
- [SELECT clause](#)
- [FROM clause](#)

# Working with arrays and objects in Azure Cosmos DB

12/5/2019 • 2 minutes to read • [Edit Online](#)

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

6/23/2019 • 2 minutes to read • [Edit Online](#)

This article details keywords which may be used in Azure Cosmos DB SQL queries.

## BETWEEN

As in ANSI SQL, 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
```

Unlike in ANSI SQL, you can also use the BETWEEN clause in the FROM 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.

### TIP

For faster query execution times, create an indexing policy that uses a range index type against any numeric properties or paths that the BETWEEN clause filters.

## DISTINCT

The DISTINCT keyword eliminates duplicates in the query's projection.

```
SELECT DISTINCT VALUE f.lastName
FROM Families f
```

In this example, the query projects values for each last name.

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"  
    ]  
  }  
]
```

## 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.

## 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

12/5/2019 • 2 minutes to read • [Edit Online](#)

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
<b>Undefined</b>	Single value: <b>undefined</b>
<b>Null</b>	Single value: <b>null</b>
<b>Boolean</b>	Values: <b>false, true</b> .
<b>Number</b>	A double-precision floating-point number, IEEE 754 standard.
<b>String</b>	A sequence of zero or more Unicode characters. Strings must be enclosed in single or double quotes.
<b>Array</b>	A sequence of zero or more elements. Each element can be a value of any scalar data type, except <b>Undefined</b> .
<b>Object</b>	An unordered set of zero or more name/value pairs. Name is a Unicode string, value can be of any scalar data type, except <b>Undefined</b> .

## 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
\V	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

12/5/2019 • 2 minutes to read • [Edit Online](#)

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
<b>Undefined</b>	Undefined						
<b>Null</b>	Undefined	<b>Ok</b>	Undefined	Undefined	Undefined	Undefined	Undefined
<b>Boolean</b>	Undefined	Undefined	<b>Ok</b>	Undefined	Undefined	Undefined	Undefined
<b>Number</b>	Undefined	Undefined	Undefined	<b>Ok</b>	Undefined	Undefined	Undefined
<b>String</b>	Undefined	Undefined	Undefined	Undefined	<b>Ok</b>	Undefined	Undefined
<b>Object</b>	Undefined	Undefined	Undefined	Undefined	Undefined	<b>Ok</b>	Undefined
<b>Array</b>	Undefined	Undefined	Undefined	Undefined	Undefined	Undefined	<b>Ok</b>

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`.

## 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

OR	TRUE	FALSE	UNDEFINED
True	True	True	True
False	True	False	Undefined
Undefined	True	Undefined	Undefined

### AND operator

AND	TRUE	FALSE	UNDEFINED
True	True	False	Undefined

AND	TRUE	FALSE	UNDEFINED
False	False	False	False
Undefined	Undefined	False	Undefined

## NOT operator

NOT	
True	False
False	True
Undefined	Undefined

## \* 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

12/5/2019 • 2 minutes to read • [Edit Online](#)

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

8/19/2019 • 2 minutes to read • [Edit Online](#)

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.

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. In fact, the UDFs are exquisitely designed to call from queries. As a corollary, UDFs do not have access to the context object like other JavaScript types, such as stored procedures and triggers. Queries are read-only, and can run either on primary or secondary replicas. UDFs, unlike other JavaScript types, are designed to run on secondary replicas.

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.

## Examples

```
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

12/5/2019 • 2 minutes to read • [Edit Online](#)

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 the `Families` container:

## Examples

```
SELECT COUNT(1)
FROM Families f
```

The results are:

```
[{
    "$1": 2
}]
```

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 ]
```

## 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
COUNT	Returns the number of items in the expression.

FUNCTION	DESCRIPTION
SUM	Returns the sum of all the values in the expression.
MIN	Returns the minimum value in the expression.
MAX	Returns the maximum value in the expression.
AVG	Returns the average of the values in the expression.

You can also aggregate over the results of an array iteration.

#### NOTE

In the Azure portal's Data Explorer, aggregation queries may aggregate partial results over only one query page. The SDK produces a single cumulative value across all pages. To perform aggregation queries using code, you need .NET SDK 1.12.0, .NET Core SDK 1.1.0, or Java SDK 1.9.5 or above.

## Next steps

- [Introduction to Azure Cosmos DB](#)
- [System functions](#)
- [User defined functions](#)

# System functions (Azure Cosmos DB)

12/5/2019 • 2 minutes to read • [Edit Online](#)

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.	<a href="#">ARRAY_CONCAT</a> , <a href="#">ARRAY_CONTAINS</a> , <a href="#">ARRAY_LENGTH</a> , <a href="#">ARRAY_SLICE</a>
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.	<a href="#">GetCurrentDateTime</a> , <a href="#">GetCurrentTimestamp</a>
Mathematical functions	The mathematical functions each perform a calculation, usually based on input values that are provided as arguments, and return a numeric value.	<a href="#">ABS</a> , <a href="#">ACOS</a> , <a href="#">ASIN</a> , <a href="#">ATAN</a> , <a href="#">ATN2</a> , <a href="#">CEILING</a> , <a href="#">COS</a> , <a href="#">COT</a> , <a href="#">DEGREES</a> , <a href="#">EXP</a> , <a href="#">FLOOR</a> , <a href="#">LOG</a> , <a href="#">LOG10</a> , <a href="#">PI</a> , <a href="#">POWER</a> , <a href="#">RADIANS</a> , <a href="#">RAND</a> , <a href="#">ROUND</a> , <a href="#">SIGN</a> , <a href="#">SIN</a> , <a href="#">SQRT</a> , <a href="#">SQUARE</a> , <a href="#">TAN</a> , <a href="#">TRUNC</a>
Spatial functions	The spatial functions perform an operation on a spatial object input value and return a numeric or Boolean value.	<a href="#">ST_DISTANCE</a> , <a href="#">ST_INTERSECTS</a> , <a href="#">ST_ISVALID</a> , <a href="#">ST_ISVALIDDETAILED</a> , <a href="#">ST_WITHIN</a>
String functions	The string functions perform an operation on a string input value and return a string, numeric or Boolean value.	<a href="#">CONCAT</a> , <a href="#">CONTAINS</a> , <a href="#">ENDSWITH</a> , <a href="#">INDEX_OF</a> , <a href="#">LEFT</a> , <a href="#">LENGTH</a> , <a href="#">LOWER</a> , <a href="#">LTRIM</a> , <a href="#">REPLACE</a> , <a href="#">REPLICATE</a> , <a href="#">REVERSE</a> , <a href="#">RIGHT</a> , <a href="#">RTRIM</a> , <a href="#">STARTSWITH</a> , <a href="#">StringToArray</a> , <a href="#">StringToBoolean</a> , <a href="#">StringToNull</a> , <a href="#">StringToNumber</a> , <a href="#">StringToObject</a> , <a href="#">SUBSTRING</a> , <a href="#">ToString</a> , <a href="#">TRIM</a> , <a href="#">UPPER</a>
Type checking functions	The type checking functions allow you to check the type of an expression within SQL queries.	<a href="#">IS_ARRAY</a> , <a href="#">IS_BOOL</a> , <a href="#">IS_DEFINED</a> , <a href="#">IS_NULL</a> , <a href="#">IS_NUMBER</a> , <a href="#">IS_OBJECT</a> , <a href="#">IS_PRIMITIVE</a> , <a href="#">IS_STRING</a>

## 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)

12/5/2019 • 2 minutes to read • [Edit Online](#)

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:

<a href="#">ARRAY_CONCAT</a>	<a href="#">ARRAY_CONTAINS</a>	<a href="#">ARRAY_LENGTH</a>
<a href="#">ARRAY_SLICE</a>		

## Next steps

- [System functions Azure Cosmos DB](#)
- [Introduction to Azure Cosmos DB](#)
- [User Defined Functions](#)
- [Aggregates](#)

# ARRAY\_CONCAT (Azure Cosmos DB)

12/5/2019 • 2 minutes to read • [Edit Online](#)

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"]}]
```

## Next steps

- [Array functions Azure Cosmos DB](#)
- [System functions Azure Cosmos DB](#)
- [Introduction to Azure Cosmos DB](#)

# ARRAY\_CONTAINS (Azure Cosmos DB)

12/5/2019 • 2 minutes to read • [Edit Online](#)

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  
}]
```

## Next steps

- [Array functions Azure Cosmos DB](#)
- [System functions Azure Cosmos DB](#)
- [Introduction to Azure Cosmos DB](#)

# ARRAY\_LENGTH (Azure Cosmos DB)

12/5/2019 • 2 minutes to read • [Edit Online](#)

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}]
```

## Next steps

- [Array functions Azure Cosmos DB](#)
- [System functions Azure Cosmos DB](#)
- [Introduction to Azure Cosmos DB](#)

# ARRAY\_SLICE (Azure Cosmos DB)

12/5/2019 • 2 minutes to read • [Edit Online](#)

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": []
}]
```

## Next steps

- [Array functions Azure Cosmos DB](#)
- [System functions Azure Cosmos DB](#)
- [Introduction to Azure Cosmos DB](#)

# Date and time functions (Azure Cosmos DB)

12/5/2019 • 2 minutes to read • [Edit Online](#)

The date and time functions let you perform DateTime and timestamp operations in Azure Cosmos DB.

## Functions

The following scalar 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:

<a href="#">GetCurrentDateTime</a>	<a href="#">GetCurrentTimestamp</a>	

## Next steps

- [System functions Azure Cosmos DB](#)
- [Introduction to Azure Cosmos DB](#)
- [User Defined Functions](#)
- [Aggregates](#)

# GetCurrentDateTime (Azure Cosmos DB)

9/27/2019 • 2 minutes to read • [Edit Online](#)

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.sssZ` where:

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)
.sss	three digits of decimal fractions of a second
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.

## 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": "2019-05-03T20:36:17.784Z"  
}]
```

## Next steps

- [Date and time functions Azure Cosmos DB](#)
- [System functions Azure Cosmos DB](#)
- [Introduction to Azure Cosmos DB](#)

# GetCurrentTimestamp (Azure Cosmos DB)

9/27/2019 • 2 minutes to read • [Edit Online](#)

Returns the number of milliseconds that have elapsed since 00:00:00 Thursday, 1 January 1970.

## Syntax

```
GetCurrentTimestamp ()
```

## Return types

Returns a 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).

## 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](#)

# Mathematical functions (Azure Cosmos DB)

12/5/2019 • 2 minutes to read • [Edit Online](#)

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)

12/5/2019 • 2 minutes to read • [Edit Online](#)

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}]
```

## Next steps

- [Mathematical functions Azure Cosmos DB](#)
- [System functions Azure Cosmos DB](#)
- [Introduction to Azure Cosmos DB](#)

# ACOS (Azure Cosmos DB)

12/5/2019 • 2 minutes to read • [Edit Online](#)

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}]
```

## Next steps

- [Mathematical functions Azure Cosmos DB](#)
- [System functions Azure Cosmos DB](#)
- [Introduction to Azure Cosmos DB](#)

# ASIN (Azure Cosmos DB)

12/5/2019 • 2 minutes to read • [Edit Online](#)

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}]
```

## Next steps

- [Mathematical functions Azure Cosmos DB](#)
- [System functions Azure Cosmos DB](#)
- [Introduction to Azure Cosmos DB](#)

# ATAN (Azure Cosmos DB)

12/5/2019 • 2 minutes to read • [Edit Online](#)

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}]
```

## Next steps

- [Mathematical functions Azure Cosmos DB](#)
- [System functions Azure Cosmos DB](#)
- [Introduction to Azure Cosmos DB](#)

# ATN2 (Azure Cosmos DB)

12/5/2019 • 2 minutes to read • [Edit Online](#)

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}]
```

## Next steps

- [Mathematical functions Azure Cosmos DB](#)
- [System functions Azure Cosmos DB](#)
- [Introduction to Azure Cosmos DB](#)

# CEILING (Azure Cosmos DB)

12/20/2019 • 2 minutes to read • [Edit Online](#)

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}]
```

## Next steps

- [Mathematical functions Azure Cosmos DB](#)
- [System functions Azure Cosmos DB](#)
- [Introduction to Azure Cosmos DB](#)

# COS (Azure Cosmos DB)

12/5/2019 • 2 minutes to read • [Edit Online](#)

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}]
```

## Next steps

- [Mathematical functions Azure Cosmos DB](#)
- [System functions Azure Cosmos DB](#)
- [Introduction to Azure Cosmos DB](#)

# COT (Azure Cosmos DB)

12/5/2019 • 2 minutes to read • [Edit Online](#)

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}]
```

## Next steps

- [Mathematical functions Azure Cosmos DB](#)
- [System functions Azure Cosmos DB](#)
- [Introduction to Azure Cosmos DB](#)

# DEGREES (Azure Cosmos DB)

12/5/2019 • 2 minutes to read • [Edit Online](#)

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}]
```

## Next steps

- [Mathematical functions Azure Cosmos DB](#)
- [System functions Azure Cosmos DB](#)
- [Introduction to Azure Cosmos DB](#)

# EXP (Azure Cosmos DB)

12/5/2019 • 2 minutes to read • [Edit Online](#)

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: 19.99999999999996, exp2: 20}]
```

## Next steps

- [Mathematical functions Azure Cosmos DB](#)
- [System functions Azure Cosmos DB](#)
- [Introduction to Azure Cosmos DB](#)

# FLOOR (Azure Cosmos DB)

12/5/2019 • 2 minutes to read • [Edit Online](#)

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 f11, FLOOR(-123.45) AS f12, FLOOR(0.0) AS f13
```

Here is the result set.

```
[{f11: 123, f12: -124, f13: 0}]
```

## Next steps

- [Mathematical functions Azure Cosmos DB](#)
- [System functions Azure Cosmos DB](#)
- [Introduction to Azure Cosmos DB](#)

# LOG (Azure Cosmos DB)

12/5/2019 • 2 minutes to read • [Edit Online](#)

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.

## 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: 10.00000000000002}]
```

## Next steps

- [Mathematical functions Azure Cosmos DB](#)
- [System functions Azure Cosmos DB](#)
- [Introduction to Azure Cosmos DB](#)

# LOG10 (Azure Cosmos DB)

12/5/2019 • 2 minutes to read • [Edit Online](#)

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,  $10 ^ \text{LOG10}(n) = n$ .

## 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)

9/27/2019 • 2 minutes to read • [Edit Online](#)

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)

9/27/2019 • 2 minutes to read • [Edit Online](#)

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)

9/27/2019 • 2 minutes to read • [Edit Online](#)

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  
}]
```

## Next steps

- [Mathematical functions Azure Cosmos DB](#)
- [System functions Azure Cosmos DB](#)
- [Introduction to Azure Cosmos DB](#)

# RAND (Azure Cosmos DB)

9/27/2019 • 2 minutes to read • [Edit Online](#)

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.

## 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)

9/27/2019 • 2 minutes to read • [Edit Online](#)

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.

<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)

9/27/2019 • 2 minutes to read • [Edit Online](#)

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}]
```

## Next steps

- [Mathematical functions Azure Cosmos DB](#)
- [System functions Azure Cosmos DB](#)
- [Introduction to Azure Cosmos DB](#)

# SIN (Azure Cosmos DB)

9/27/2019 • 2 minutes to read • [Edit Online](#)

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}]
```

## Next steps

- [Mathematical functions Azure Cosmos DB](#)
- [System functions Azure Cosmos DB](#)
- [Introduction to Azure Cosmos DB](#)

# SQRT (Azure Cosmos DB)

9/27/2019 • 2 minutes to read • [Edit Online](#)

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}]
```

## Next steps

- [Mathematical functions Azure Cosmos DB](#)
- [System functions Azure Cosmos DB](#)
- [Introduction to Azure Cosmos DB](#)

# SQUARE (Azure Cosmos DB)

9/27/2019 • 2 minutes to read • [Edit Online](#)

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}]
```

## Next steps

- [Mathematical functions Azure Cosmos DB](#)
- [System functions Azure Cosmos DB](#)
- [Introduction to Azure Cosmos DB](#)

# TAN (Azure Cosmos DB)

9/27/2019 • 2 minutes to read • [Edit Online](#)

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 }]
```

## Next steps

- [Mathematical functions Azure Cosmos DB](#)
- [System functions Azure Cosmos DB](#)
- [Introduction to Azure Cosmos DB](#)

# TRUNC (Azure Cosmos DB)

9/27/2019 • 2 minutes to read • [Edit Online](#)

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}]
```

## Next steps

- [Mathematical functions Azure Cosmos DB](#)
- [System functions Azure Cosmos DB](#)
- [Introduction to Azure Cosmos DB](#)

# Spatial functions (Azure Cosmos DB)

9/27/2019 • 2 minutes to read • [Edit Online](#)

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.

<a href="#">ST_DISTANCE</a>	<a href="#">ST_INTERSECTS</a>	<a href="#">ST_ISVALID</a>		
<a href="#">ST_ISVALIDDETAILED</a>	<a href="#">ST_WITHIN</a>			

## Next steps

- [System functions Azure Cosmos DB](#)
- [Introduction to Azure Cosmos DB](#)
- [User Defined Functions](#)
- [Aggregates](#)

# ST\_DISTANCE (Azure Cosmos DB)

9/27/2019 • 2 minutes to read • [Edit Online](#)

Returns the distance between the two GeoJSON Point, Polygon, or LineString expressions.

## 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"
}]
```

## Next steps

- [Spatial functions Azure Cosmos DB](#)
- [System functions Azure Cosmos DB](#)
- [Introduction to Azure Cosmos DB](#)

# ST\_WITHIN (Azure Cosmos DB)

9/27/2019 • 2 minutes to read • [Edit Online](#)

Returns a Boolean expression indicating whether the GeoJSON object (Point, Polygon, or LineString) specified in the first argument is within the GeoJSON (Point, Polygon, 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" }]
```

## Next steps

- [Spatial functions Azure Cosmos DB](#)
- [System functions Azure Cosmos DB](#)
- [Introduction to Azure Cosmos DB](#)

# ST\_INTERSECTS (Azure Cosmos DB)

9/27/2019 • 2 minutes to read • [Edit Online](#)

Returns a Boolean expression indicating whether the GeoJSON object (Point, Polygon, or LineString) specified in the first argument intersects the GeoJSON (Point, Polygon, 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" }]
```

## Next steps

- [Spatial functions Azure Cosmos DB](#)
- [System functions Azure Cosmos DB](#)
- [Introduction to Azure Cosmos DB](#)

# ST\_ISVALID (Azure Cosmos DB)

9/27/2019 • 2 minutes to read • [Edit Online](#)

Returns a Boolean value indicating whether the specified GeoJSON Point, Polygon, 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)

9/27/2019 • 2 minutes to read • [Edit Online](#)

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)

9/27/2019 • 2 minutes to read • [Edit Online](#)

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	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)

12/5/2019 • 2 minutes to read • [Edit Online](#)

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"}]
```

## Next steps

- [String functions Azure Cosmos DB](#)
- [System functions Azure Cosmos DB](#)
- [Introduction to Azure Cosmos DB](#)

# CONTAINS (Azure Cosmos DB)

12/5/2019 • 2 minutes to read • [Edit Online](#)

Returns a Boolean indicating whether the first string expression contains the second.

## Syntax

```
CONTAINS(<str_expr1>, <str_expr2>)
```

## Arguments

*str\_expr1*

Is the string expression to be searched.

*str\_expr2*

Is the string expression to find.

## Return types

Returns a Boolean expression.

## Examples

The following example checks if "abc" contains "ab" and if "abc" contains "d".

```
SELECT CONTAINS("abc", "ab") AS c1, CONTAINS("abc", "d") AS c2
```

Here is the result set.

```
[{"c1": true, "c2": false}]
```

## Next steps

- [String functions Azure Cosmos DB](#)
- [System functions Azure Cosmos DB](#)
- [Introduction to Azure Cosmos DB](#)

# ENDSWITH (Azure Cosmos DB)

12/5/2019 • 2 minutes to read • [Edit Online](#)

Returns a Boolean indicating whether the first string expression ends with the second.

## Syntax

```
ENDSWITH(<str_expr1>, <str_expr2>)
```

## Arguments

*str\_expr1*

Is a string expression.

*str\_expr2*

Is a string expression to be compared to the end of *str\_expr1*.

## Return types

Returns a Boolean expression.

## Examples

The following example returns the "abc" ends with "b" and "bc".

```
SELECT ENDSWITH("abc", "b") AS e1, ENDSWITH("abc", "bc") AS e2
```

Here is the result set.

```
[{"e1": false, "e2": true}]
```

## Next steps

- [String functions Azure Cosmos DB](#)
- [System functions Azure Cosmos DB](#)
- [Introduction to Azure Cosmos DB](#)

# INDEX\_OF (Azure Cosmos DB)

9/27/2019 • 2 minutes to read • [Edit Online](#)

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)

9/27/2019 • 2 minutes to read • [Edit Online](#)

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 11, LEFT("abc", 2) AS 12
```

Here is the result set.

```
[{"11": "a", "12": "ab"}]
```

## Next steps

- [String functions Azure Cosmos DB](#)
- [System functions Azure Cosmos DB](#)
- [Introduction to Azure Cosmos DB](#)

# LENGTH (Azure Cosmos DB)

9/27/2019 • 2 minutes to read • [Edit Online](#)

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}]
```

## Next steps

- [String functions Azure Cosmos DB](#)
- [System functions Azure Cosmos DB](#)
- [Introduction to Azure Cosmos DB](#)

# LOWER (Azure Cosmos DB)

1/8/2020 • 2 minutes to read • [Edit Online](#)

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"}]
```

## Next steps

- [String functions Azure Cosmos DB](#)
- [System functions Azure Cosmos DB](#)
- [Introduction to Azure Cosmos DB](#)

# LTRIM (Azure Cosmos DB)

12/5/2019 • 2 minutes to read • [Edit Online](#)

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 11, LTRIM("abc") AS 12, LTRIM("abc    ") AS 13
```

Here is the result set.

```
[{"11": "abc", "12": "abc", "13": "abc    "}]
```

## Next steps

- [String functions Azure Cosmos DB](#)
- [System functions Azure Cosmos DB](#)
- [Introduction to Azure Cosmos DB](#)

# REPLACE (Azure Cosmos DB)

9/27/2019 • 2 minutes to read • [Edit Online](#)

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"}]
```

## Next steps

- [String functions Azure Cosmos DB](#)
- [System functions Azure Cosmos DB](#)
- [Introduction to Azure Cosmos DB](#)

# REPLICATE (Azure Cosmos DB)

9/27/2019 • 2 minutes to read • [Edit Online](#)

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.  $(\text{length}(\text{str\_expr}) * \text{num\_expr}) \leq 10,000$ .

## 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)

9/27/2019 • 2 minutes to read • [Edit Online](#)

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"}]
```

## Next steps

- [String functions Azure Cosmos DB](#)
- [System functions Azure Cosmos DB](#)
- [Introduction to Azure Cosmos DB](#)

# RIGHT (Azure Cosmos DB)

9/27/2019 • 2 minutes to read • [Edit Online](#)

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"}]
```

## Next steps

- [String functions Azure Cosmos DB](#)
- [System functions Azure Cosmos DB](#)
- [Introduction to Azure Cosmos DB](#)

# RTRIM (Azure Cosmos DB)

9/27/2019 • 2 minutes to read • [Edit Online](#)

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"}]
```

## Next steps

- [String functions Azure Cosmos DB](#)
- [System functions Azure Cosmos DB](#)
- [Introduction to Azure Cosmos DB](#)

# STARTSWITH (Azure Cosmos DB)

9/27/2019 • 2 minutes to read • [Edit Online](#)

Returns a Boolean indicating whether the first string expression starts with the second.

## Syntax

```
STARTSWITH(<str_expr1>, <str_expr2>)
```

## Arguments

*str\_expr1*

Is a string expression.

*str\_expr2*

Is a string expression to be compared to the beginning of *str\_expr1*.

## Return types

Returns a Boolean expression.

## Examples

The following example checks if the string "abc" begins with "b" and "a".

```
SELECT STARTSWITH("abc", "b") AS s1, STARTSWITH("abc", "a") AS s2
```

Here is the result set.

```
[{"s1": false, "s2": true}]
```

## Next steps

- [String functions Azure Cosmos DB](#)
- [System functions Azure Cosmos DB](#)
- [Introduction to Azure Cosmos DB](#)

# StringToArray (Azure Cosmos DB)

9/27/2019 • 2 minutes to read • [Edit Online](#)

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](#)

## 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)

9/27/2019 • 2 minutes to read • [Edit Online](#)

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  
    String.ToBoolean("null"),  
    String.ToBoolean(undefined),  
    String.ToBoolean(NaN),  
    String.ToBoolean(false),  
    String.ToBoolean(true)
```

Here is the result set.

```
[{}]
```

## Next steps

- [String functions Azure Cosmos DB](#)
- [System functions Azure Cosmos DB](#)
- [Introduction to Azure Cosmos DB](#)

# StringToNull (Azure Cosmos DB)

9/27/2019 • 2 minutes to read • [Edit Online](#)

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.

```
[{}]
```

## Next steps

- [String functions Azure Cosmos DB](#)
- [System functions Azure Cosmos DB](#)
- [Introduction to Azure Cosmos DB](#)

# StringToNumber (Azure Cosmos DB)

9/27/2019 • 2 minutes to read • [Edit Online](#)

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.

```
{()}
```

## Next steps

- [String functions Azure Cosmos DB](#)
- [System functions Azure Cosmos DB](#)
- [Introduction to Azure Cosmos DB](#)

# StringToObject (Azure Cosmos DB)

9/27/2019 • 2 minutes to read • [Edit Online](#)

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.

```
[{}]
```

## Next steps

- [String functions Azure Cosmos DB](#)
- [System functions Azure Cosmos DB](#)
- [Introduction to Azure Cosmos DB](#)

# SUBSTRING (Azure Cosmos DB)

9/27/2019 • 2 minutes to read • [Edit Online](#)

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"}]
```

## Next steps

- [String functions Azure Cosmos DB](#)
- [System functions Azure Cosmos DB](#)
- [Introduction to Azure Cosmos DB](#)

# ToString (Azure Cosmos DB)

9/27/2019 • 2 minutes to read • [Edit Online](#)

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":"41b"},  
 {"$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"}]
```

## Next steps

- [String functions Azure Cosmos DB](#)
- [System functions Azure Cosmos DB](#)
- [Introduction to Azure Cosmos DB](#)

# TRIM (Azure Cosmos DB)

9/27/2019 • 2 minutes to read • [Edit Online](#)

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"}]
```

## Next steps

- [String functions Azure Cosmos DB](#)
- [System functions Azure Cosmos DB](#)
- [Introduction to Azure Cosmos DB](#)

# UPPER (Azure Cosmos DB)

1/8/2020 • 2 minutes to read • [Edit Online](#)

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"}]
```

## Next steps

- [String functions Azure Cosmos DB](#)
- [System functions Azure Cosmos DB](#)
- [Introduction to Azure Cosmos DB](#)

# Type checking functions (Azure Cosmos DB)

9/27/2019 • 2 minutes to read • [Edit Online](#)

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)

9/27/2019 • 2 minutes to read • [Edit Online](#)

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,"isArray7":false}]
```

## Next steps

- [Type checking functions Azure Cosmos DB](#)
- [System functions Azure Cosmos DB](#)
- [Introduction to Azure Cosmos DB](#)

# IS\_BOOL (Azure Cosmos DB)

9/27/2019 • 2 minutes to read • [Edit Online](#)

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":false}]
```

## Next steps

- [Type checking functions Azure Cosmos DB](#)
- [System functions Azure Cosmos DB](#)
- [Introduction to Azure Cosmos DB](#)

# IS\_DEFINED (Azure Cosmos DB)

9/27/2019 • 2 minutes to read • [Edit Online](#)

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}]
```

## Next steps

- [Type checking functions Azure Cosmos DB](#)
- [System functions Azure Cosmos DB](#)
- [Introduction to Azure Cosmos DB](#)

# IS\_NULL (Azure Cosmos DB)

9/27/2019 • 2 minutes to read • [Edit Online](#)

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) ASisNull1,
    IS_NULL(1) ASisNull2,
    IS_NULL("value") ASisNull3,
    IS_NULL(null) ASisNull4,
    IS_NULL({prop: "value"}) ASisNull5,
    IS_NULL([1, 2, 3]) ASisNull6,
    IS_NULL({prop: "value"}.prop2) ASisNull7
```

Here is the result set.

```
[{"isNull1":false,"isNull2":false,"isNull3":false,"isNull4":true,"isNull5":false,"isNull6":false,"isNull7":false}]
```

## Next steps

- [Type checking functions Azure Cosmos DB](#)
- [System functions Azure Cosmos DB](#)
- [Introduction to Azure Cosmos DB](#)

# IS\_NUMBER (Azure Cosmos DB)

9/27/2019 • 2 minutes to read • [Edit Online](#)

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}]
```

## Next steps

- [Type checking functions Azure Cosmos DB](#)
- [System functions Azure Cosmos DB](#)
- [Introduction to Azure Cosmos DB](#)

# IS\_OBJECT (Azure Cosmos DB)

9/27/2019 • 2 minutes to read • [Edit Online](#)

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}]
```

## Next steps

- [Type checking functions Azure Cosmos DB](#)
- [System functions Azure Cosmos DB](#)
- [Introduction to Azure Cosmos DB](#)

# IS\_PRIMITIVE (Azure Cosmos DB)

9/27/2019 • 2 minutes to read • [Edit Online](#)

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}]
```

## Next steps

- [Type checking functions Azure Cosmos DB](#)
- [System functions Azure Cosmos DB](#)
- [Introduction to Azure Cosmos DB](#)

# IS\_STRING (Azure Cosmos DB)

9/27/2019 • 2 minutes to read • [Edit Online](#)

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}]
```

## 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

2/23/2020 • 5 minutes to read • [Edit Online](#)

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?

## 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. Common use cases often involve proximity queries, for example, "find all coffee shops near my current location."

Azure Cosmos DB's SQL API supports the **geography** data type. 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 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

2/23/2020 • 4 minutes to read • [Edit Online](#)

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

2/23/2020 • 2 minutes to read • [Edit Online](#)

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 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

## 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.

## 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](#)

# Parameterized queries in Azure Cosmos DB

12/5/2019 • 2 minutes to read • [Edit Online](#)

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 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 Cosmos DB is schemaless, parameters aren't validated against any type.

## Next steps

- [Azure Cosmos DB .NET samples](#)
- [Model document data](#)

# LINQ to SQL translation

12/13/2019 • 4 minutes to read • [Edit Online](#)

The Azure Cosmos DB query provider performs a best effort mapping from a LINQ query into a Cosmos DB SQL query. The following description assumes a basic familiarity with LINQ.

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 };
```

## Supported LINQ operators

The LINQ provider included with the SQL .NET SDK supports the following operators:

- Select**: Projections translate to SQL SELECT, including object construction.
- Where**: Filters translate to SQL WHERE, and support translation between `&&`, `||`, and `!` to the SQL operators
- SelectMany**: Allows unwinding of arrays to the SQL 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 SQL 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 SQL built-in functions.
- **String functions:** Supports translation from .NET `Concat`, `Contains`, `Count`, `EndsWith`, `IndexOf`, `Replace`, `Reverse`, `StartsWith`, `SubString`, `ToLower`, `ToUpper`, `TrimEnd`, and `TrimStart` to the equivalent SQL built-in functions.
- **Array functions:** Supports translation from .NET `Concat`, `Contains`, and `Count` to the equivalent SQL built-in functions.
- **Geospatial Extension functions:** Supports translation from stub methods `Distance`, `IsValid`, `IsValidDetailed`, and `Within` to the equivalent SQL built-in 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 SQL IN, depending on context.

## Examples

The following examples illustrate how some of the standard LINQ query operators translate to Cosmos DB queries.

### Select operator

The syntax is `input.Select(x => f(x))`, where `f` is a scalar expression.

#### 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](#)

# Azure Cosmos DB SQL query execution

12/5/2019 • 6 minutes to read • [Edit Online](#)

Any language capable of making HTTP/HTTPS requests can call the Cosmos DB REST API. Cosmos DB also offers programming libraries for .NET, Node.js, JavaScript, and Python programming languages. The REST API and libraries all support querying through SQL, and the .NET SDK also supports [LINQ querying](#).

The following examples show how to create a query and submit it against a Cosmos database account.

## REST API

Cosmos DB offers an open RESTful programming model over HTTP. The resource model consists of a set of resources under a database account, which an Azure subscription provisions. The database account consists of a set of *databases*, each of which can contain multiple *containers*, which in turn contain *items*, UDFs, and other resource types. Each Cosmos DB resource is addressable using a logical and stable URI. A set of resources is called a *feed*.

The basic interaction model with these resources is through the HTTP verbs `GET`, `PUT`, `POST`, and `DELETE`, with their standard interpretations. Use `POST` to create a new resource, execute a stored procedure, or issue a Cosmos DB query. Queries are always read-only operations with no side-effects.

The following examples show a `POST` for a SQL API query against the sample items. The query has a simple filter on the JSON `name` property. The `x-ms-documentdb-isquery` and `Content-Type: application/query+json` headers denote that the operation is a query. Replace `mysqlapicosmosdb.documents.azure.com:443` with the URI for your Cosmos DB account.

```
POST https://mysqlapicosmosdb.documents.azure.com:443/docs HTTP/1.1
...
x-ms-documentdb-isquery: True
Content-Type: application/query+json

{
    "query": "SELECT * FROM Families f WHERE f.id = @familyId",
    "parameters": [
        {"name": "@familyId", "value": "AndersenFamily"}
    ]
}
```

The results are:

```

HTTP/1.1 200 Ok
x-ms-activity-id: 8b4678fa-a947-47d3-8dd3-549a40da6eed
x-ms-item-count: 1
x-ms-request-charge: 0.32

{
    "_rid": "u1NXANcKogE=",
    "Documents": [
        {
            "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"
            },
            "_rid": "u1NXANcKogEcAAAAAAA==",
            "_ts": 1407691744,
            "_self": "dbs\u2f82u1NXAA==\u2f82colls\u2f82u1NXANcKogE=\u2f82docs\u2f82u1NXANcKogEcAAAAAAA==\u2f82",
            "_etag": "00002b00-0000-0000-0000-53e7abe00000",
            "_attachments": "_attachments\u2f82"
        }
    ],
    "count": 1
}

```

The next, more complex query returns multiple results from a join:

```

POST https://https://mysqlapicosmosdb.documents.azure.com:443/docs HTTP/1.1
...
x-ms-documentdb-isquery: True
Content-Type: application/query+json

{
    "query": "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",
    "parameters": []
}

```

The results are:

```
HTTP/1.1 200 Ok
x-ms-activity-id: 568f34e3-5695-44d3-9b7d-62f8b83e509d
x-ms-item-count: 1
x-ms-request-charge: 7.84

{
    "_rid": "u1NXANcKogE=",
    "Documents": [
        {
            "familyName": "AndersenFamily",
            "childFirstName": "Henriette Thaulow",
            "petName": "Fluffy"
        },
        {
            "familyName": "WakefieldFamily",
            "childGivenName": "Jesse",
            "petName": "Goofy"
        },
        {
            "familyName": "WakefieldFamily",
            "childGivenName": "Jesse",
            "petName": "Shadow"
        }
    ],
    "count": 3
}
```

If a query's results can't fit in a single page, the REST API returns a continuation token through the `x-ms-continuation-token` response header. Clients can paginate results by including the header in the subsequent results. You can also control the number of results per page through the `x-ms-max-item-count` number header.

If a query has an aggregation function like COUNT, the query page may return a partially aggregated value over only one page of results. Clients must perform a second-level aggregation over these results to produce the final results. For example, sum over the counts returned in the individual pages to return the total count.

To manage the data consistency policy for queries, use the `x-ms-consistency-level` header as in all REST API requests. Session consistency also requires echoing the latest `x-ms-session-token` cookie header in the query request. The queried container's indexing policy can also influence the consistency of query results. With the default indexing policy settings for containers, the index is always current with the item contents, and query results match the consistency chosen for data. For more information, see [Azure Cosmos DB consistency levels] [consistency-levels].

If the configured indexing policy on the container can't support the specified query, the Azure Cosmos DB server returns 400 "Bad Request". This error message returns for queries with paths explicitly excluded from indexing. You can specify the `x-ms-documentdb-query-enable-scan` header to allow the query to perform a scan when an index isn't available.

You can get detailed metrics on query execution by setting the `x-ms-documentdb-populatequerymetrics` header to `true`. For more information, see [SQL query metrics for Azure Cosmos DB](#).

## C# (.NET SDK)

The .NET SDK supports both LINQ and SQL querying. The following example shows how to perform the preceding filter query with .NET:

```

foreach (var family in client.CreateDocumentQuery(containerLink,
    "SELECT * FROM Families f WHERE f.id = \"AndersenFamily\""))
{
    Console.WriteLine("\tRead {0} from SQL", family);
}

SqlQuerySpec query = new SqlQuerySpec("SELECT * FROM Families f WHERE f.id = @familyId");
query.Parameters = new SqlParameterCollection();
query.Parameters.Add(new SqlParameter("@familyId", "AndersenFamily"));

foreach (var family in client.CreateDocumentQuery(containerLink, query))
{
    Console.WriteLine("\tRead {0} from parameterized SQL", family);
}

foreach (var family in (
    from f in client.CreateDocumentQuery(containerLink)
    where f.Id == "AndersenFamily"
    select f))
{
    Console.WriteLine("\tRead {0} from LINQ query", family);
}

foreach (var family in client.CreateDocumentQuery(containerLink)
    .Where(f => f.Id == "AndersenFamily")
    .Select(f => f))
{
    Console.WriteLine("\tRead {0} from LINQ lambda", family);
}

```

The following example compares two properties for equality within each item, and uses anonymous projections.

```

foreach (var family in client.CreateDocumentQuery(containerLink,
    @"SELECT {"Name": f.id, "City":f.address.city} AS Family
    FROM Families f
    WHERE f.address.city = f.address.state"))
{
    Console.WriteLine("\tRead {0} from SQL", family);
}

foreach (var family in (
    from f in client.CreateDocumentQuery<Family>(containerLink)
    where f.address.city == f.address.state
    select new { Name = f.Id, City = f.address.city }))
{
    Console.WriteLine("\tRead {0} from LINQ query", family);
}

foreach (var family in
    client.CreateDocumentQuery<Family>(containerLink)
    .Where(f => f.address.city == f.address.state)
    .Select(f => new { Name = f.Id, City = f.address.city }))
{
    Console.WriteLine("\tRead {0} from LINQ lambda", family);
}

```

The next example shows joins, expressed through LINQ `SelectMany`.

```

foreach (var pet in client.CreateDocumentQuery(containerLink,
    @"SELECT p
      FROM Families f
      JOIN c IN f.children
      JOIN p IN c.pets
     WHERE p.givenName = ""Shadow""")
{
    Console.WriteLine("\tRead {0} from SQL", pet);
}

// Equivalent in Lambda expressions:
foreach (var pet in
    client.CreateDocumentQuery<Family>(containerLink)
    .SelectMany(f => f.children)
    .SelectMany(c => c.pets)
    .Where(p => p.givenName == "Shadow"))
{
    Console.WriteLine("\tRead {0} from LINQ lambda", pet);
}

```

The .NET client automatically iterates through all the pages of query results in the `foreach` blocks, as shown in the preceding example. The query options introduced in the [REST API](#) section are also available in the .NET SDK, using the `FeedOptions` and `FeedResponse` classes in the `CreateDocumentQuery` method. You can control the number of pages by using the `MaxItemCount` setting.

You can also explicitly control paging by creating `IDocumentQueryable` using the `IQueryable` object, then by reading the `ResponseContinuationToken` values and passing them back as `RequestContinuationToken` in `FeedOptions`. You can set `EnableScanInQuery` to enable scans when the query isn't supported by the configured indexing policy. For partitioned containers, you can use `PartitionKey` to run the query against a single partition, although Azure Cosmos DB can automatically extract this from the query text. You can use `EnableCrossPartitionQuery` to run queries against multiple partitions.

For more .NET samples with queries, see the [Azure Cosmos DB .NET samples](#) in GitHub.

## JavaScript server-side API

Azure Cosmos DB provides a programming model for [executing JavaScript based application](#) logic directly on containers, using stored procedures and triggers. The JavaScript logic registered at the container level can then issue database operations on the items of the given container, wrapped in ambient ACID transactions.

The following example shows how to use `queryDocuments` in the JavaScript server API to make queries from inside stored procedures and triggers:

```
function findName(givenName, familyName) {
    var context = getContext();
    var containerManager = context.getCollection();
    var containerLink = containerManager.getSelfLink()

    // create a new item.
    containerManager.createDocument(containerLink,
        { givenName: givenName, familyName: familyName },
        function (err, documentCreated) {
            if (err) throw new Error(err.message);

            // filter items by familyName
            var filterQuery = "SELECT * from root r WHERE r.familyName = 'Wakefield'";
            containerManager.queryDocuments(containerLink,
                filterQuery,
                function (err, matchingDocuments) {
                    if (err) throw new Error(err.message);
                    context.getResponse().setBody(matchingDocuments.length);

                    // Replace the familyName for all items that satisfied the query.
                    for (var i = 0; i < matchingDocuments.length; i++) {
                        matchingDocuments[i].familyName = "Robin Wakefield";
                        // we don't need to execute a callback because they are in parallel
                        containerManager.replaceDocument(matchingDocuments[i]._self,
                            matchingDocuments[i]);
                    }
                })
            });
        });
}
```

## Next steps

- [Introduction to Azure Cosmos DB](#)
- [Azure Cosmos DB .NET samples](#)
- [Azure Cosmos DB consistency levels](#)

# Work with Azure Cosmos account

12/5/2019 • 2 minutes to read • [Edit Online](#)

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. 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](#).

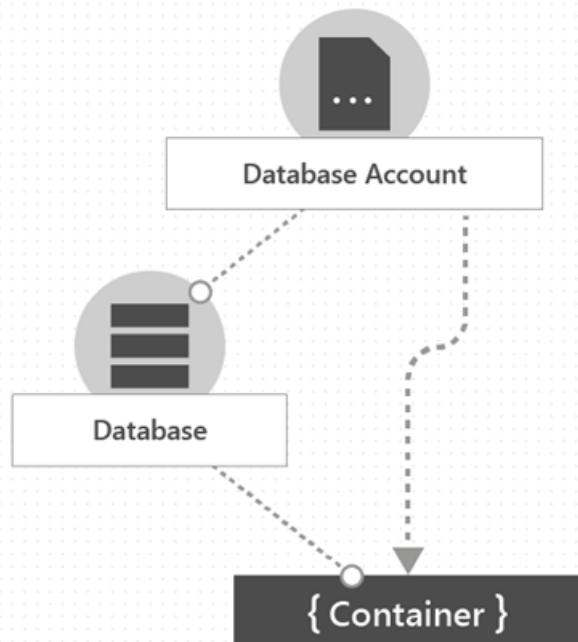
The Azure Cosmos account is the fundamental unit of global distribution and high availability. For globally distributing your data and throughput across multiple Azure regions, you can add and remove Azure regions to your Azure Cosmos account at any time. You can configure your Azure Cosmos account to have either a single or multiple write regions. For more information, see [how to add and remove Azure regions to your Azure Cosmos account](#). You can configure the [default consistency](#) level on Azure Cosmos account. Azure Cosmos DB provides comprehensive SLAs encompassing throughput, latency at the 99th percentile, consistency, and high availability. For more information, see [Azure Cosmos DB SLAs](#).

To securely manage access to all the data within your Azure Cosmos account, you can use the [master keys](#) associated with your account. To further secure access to your data, you can configure a [VNET service endpoint](#) and [IP-firewall](#) on your Azure Cosmos 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. For more information, see [working with Azure Cosmos containers and items](#).

Currently, you can create a maximum of 100 Azure Cosmos accounts under an Azure subscription. 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:



## Next steps

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

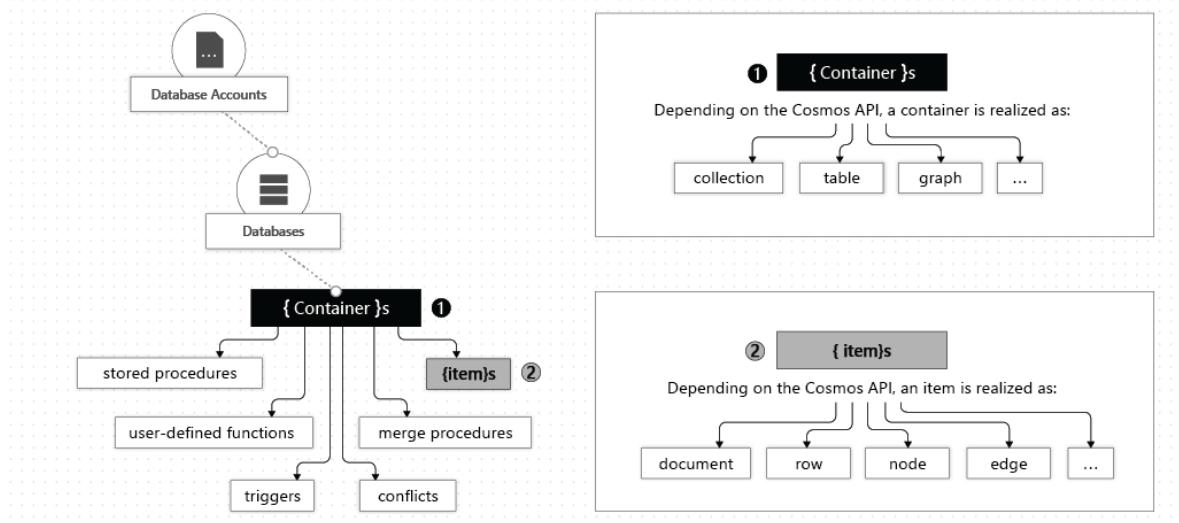
- [How-to manage your Azure Cosmos account](#)
- [Global distribution](#)
- [Consistency levels](#)
- [Working with Azure Cosmos containers and items](#)
- [VNET service endpoint for your Azure Cosmos account](#)
- [IP-firewall for your Azure Cosmos account](#)
- [How-to add and remove Azure regions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

# Work with databases, containers, and items in Azure Cosmos DB

1/26/2020 • 7 minutes to read • [Edit Online](#)

After you create an [Azure Cosmos DB account](#) under your Azure subscription, you can manage data in your account by creating databases, containers, and items. This article describes each of these entities.

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 an Azure Cosmos 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
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 and the throughput that you provision on it are automatically distributed across a set of logical partitions based on the partition key. To learn more about partitioning and partition keys, see [Partition data](#).

When you create an Azure Cosmos 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 an Azure Cosmos 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 an Azure Cosmos 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.

An Azure Cosmos 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 an Azure Cosmos 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 Azure Cosmos 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](#).

An Azure Cosmos 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

### 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

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	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
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 <a href="#">Time to Live</a> .	Yes	No	No	No	Yes
changeFeedPolicy	User-configurable	Used to read changes made to items in a container. For details, see <a href="#">Change feed</a> .	Yes	No	No	No	Yes

SYSTEM-DEFINED PROPERTY	SYSTEM-GENERATED OR USER-CONFIGURABLE	PURPOSE	SQL API	CASSANDRA API	AZURE COSMOS DB API FOR MONGODB	GREMLIN API	TABLE API
uniqueKey Policy	User-configurable	Used to ensure the uniqueness of one or more values in a logical partition. For more information, see <a href="#">Unique key constraints</a> .	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	Document	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
_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 about these tasks and concepts:

- [Provision throughput on an Azure Cosmos database](#)
- [Provision throughput on an Azure Cosmos container](#)
- [Work with logical partitions](#)
- [Configure TTL on an Azure Cosmos container](#)
- [Build reactive applications by using change feed](#)
- [Configure a unique key constraint on your Azure Cosmos container](#)

# Data modeling in Azure Cosmos DB

10/21/2019 • 14 minutes to read • [Edit Online](#)

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?

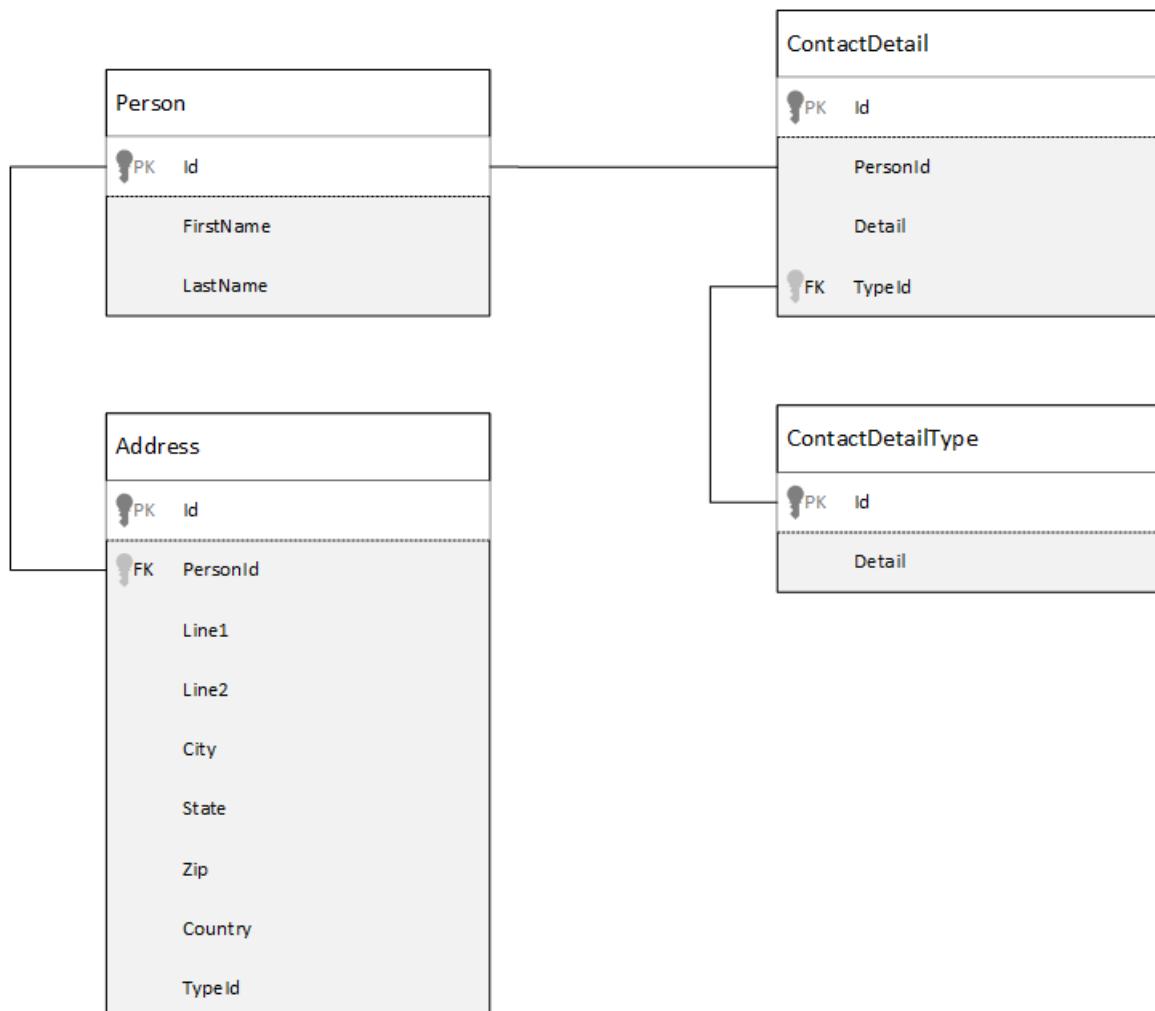
After reading this article, you will be able to answer the following questions:

- 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": 100000001, "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"
}

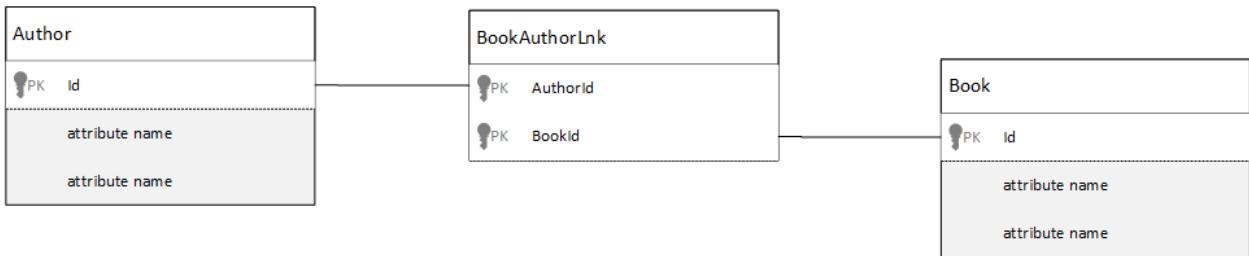
Book documents:
{"id": "1", "name": "Azure Cosmos DB 101", "pub-id": "mspress"}
{"id": "2", "name": "Azure Cosmos DB for RDBMS Users", "pub-id": "mspress"}
{"id": "3", "name": "Taking over the world one JSON doc at a time"}
...
{"id": "100", "name": "Learn about Azure Cosmos DB", "pub-id": "mspress"}
...
{"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](#).

# Indexing in Azure Cosmos DB - Overview

12/5/2019 • 4 minutes to read • [Edit Online](#)

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](#).

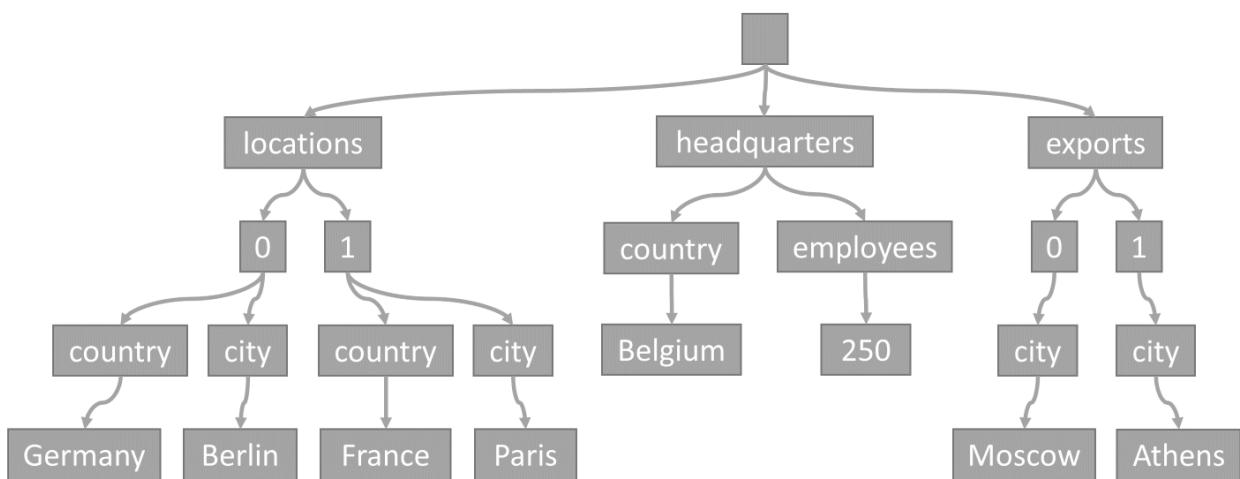
## From items to trees

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 prefix matches (CONTAINS keyword will not leverage the range index):

```
SELECT * FROM c WHERE STARTSWITH(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.

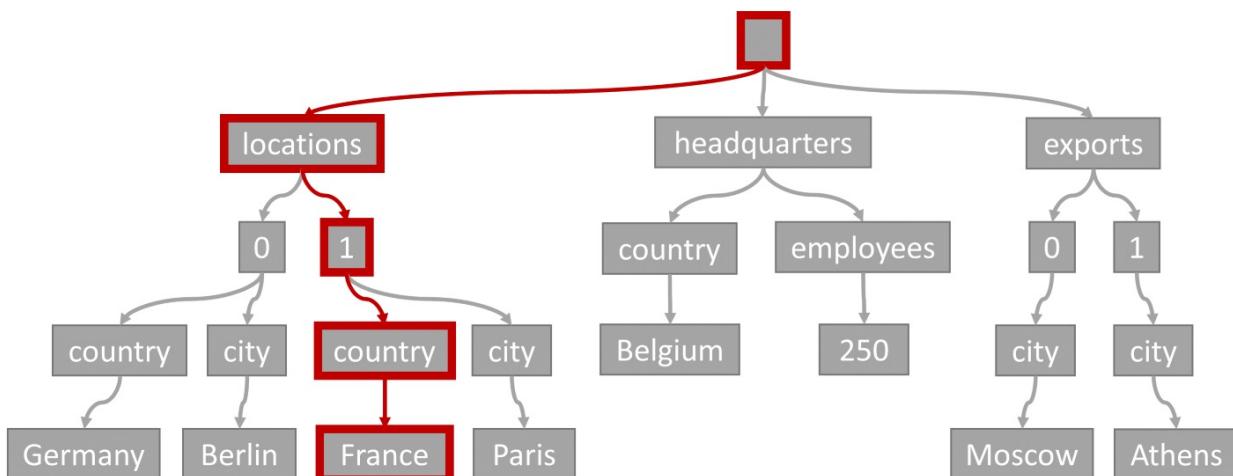
- You can speed up queries and avoid full container scans when using functions that don't use the index (e.g. `CONTAINS`) by adding additional filter predicates that do use the index. The order of filter clauses isn't important. The query engine will figure out which predicates are more selective and 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) 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](#)

# Indexing policies in Azure Cosmos DB

10/18/2019 • 12 minutes to read • [Edit Online](#)

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, enforcing range indexes for any string or number, and spatial indexes for any GeoJSON object of type Point. This allows you to get high 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.

## 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

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. Additionally, using Lazy indexing in place of 'None' for bulk operations also provides no benefit as any change to the Index Mode will cause the index to be dropped and recreated. For these reasons we recommend against customers using it. To improve performance for bulk operations, set index mode to None, then return to Consistent mode and monitor the [IndexTransformationProgress](#) property on the container until complete.

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 lower the amount of storage used by your container and improve the latency 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.

When including and excluding paths, you may encounter the following attributes:

- `kind` can be either `range` or `hash`. Range index functionality provides all of the functionality of a hash index, so we 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.

## 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.

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.

Consider the following examples where a composite index is defined on properties name, age, and timestamp:

COMPOSITE INDEX	SAMPLE QUERY	SUPPORTED BY COMPOSITE INDEX?
<code>(name ASC, age ASC)</code>	<code>SELECT * FROM c WHERE c.name = "John" AND c.age = 18</code>	Yes
<code>(name ASC, age ASC)</code>	<code>SELECT * FROM c WHERE c.name = "John" AND c.age &gt; 18</code>	Yes
<code>(name DESC, age ASC)</code>	<code>SELECT * FROM c WHERE c.name = "John" AND c.age &gt; 18</code>	Yes
<code>(name ASC, age ASC)</code>	<code>SELECT * FROM c WHERE c.name != "John" AND c.age &gt; 18</code>	No
<code>(name ASC, age ASC, timestamp ASC)</code>	<code>SELECT * FROM c WHERE c.name = "John" AND c.age = 18 AND c.timestamp &gt; 123049923</code>	Yes
<code>(name ASC, age ASC, timestamp ASC)</code>	<code>SELECT * FROM c WHERE c.name = "John" AND c.age &lt; 18 AND c.timestamp = 123049923</code>	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 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 <code>ORDER BY</code> QUERY	SUPPORTED BY COMPOSITE INDEX?
(name ASC, timestamp ASC)	<code>SELECT * FROM c WHERE c.name = "John" ORDER BY c.name ASC, c.timestamp ASC</code>	Yes
(name ASC, timestamp ASC)	<code>SELECT * FROM c WHERE c.name = "John" ORDER BY c.timestamp ASC, c.name ASC</code>	No
(name ASC, timestamp ASC)	<code>SELECT * FROM c WHERE c.name = "John" ORDER BY c.timestamp ASC</code>	No
(age ASC, name ASC, timestamp ASC)	<code>SELECT * FROM c WHERE c.age = 18 and c.name = "John" ORDER BY c.age ASC, c.name ASC, c.timestamp ASC</code>	Yes
(age ASC, name ASC, timestamp ASC)	<code>SELECT * FROM c WHERE c.age = 18 and c.name = "John" ORDER BY c.timestamp ASC</code>	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 policy's index is efficiently transformed to the new policy without affecting the write 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.

### NOTE

While adding a range or spatial index, queries may not return all the matching results, and will do so without returning any errors. This means that query results may not be consistent until the index transformation is completed. It is possible to track the progress of index transformation [by using one of the SDKs](#).

If the new indexing policy's mode is set to Consistent, no other indexing policy change can be applied while the index transformation is in progress. A running index transformation can be canceled by setting the indexing policy's mode to None (which will immediately drop the index).

## Indexing policies and TTL

The [Time-to-Live \(TTL\) feature](#) requires indexing to be active on the container it is turned on. 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, and
- no included path, and
- `/*` as the only excluded path.

## Next steps

Read more about the indexing in the following articles:

- [Indexing overview](#)
- [How to manage indexing policy](#)

# Working with Dates in Azure Cosmos DB

12/5/2019 • 3 minutes to read • [Edit Online](#)

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.sssZ` 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 UTC [DateTime](#) value.

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 client.CreateDocumentAsync("/ dbs/orderdb/colls/orders",
    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.

## Indexing DateTimes for range queries

Range queries are common with DateTime values. For example, if you need to find all orders created since yesterday, or find all orders shipped in the last five minutes, you need to perform range queries. To execute these queries efficiently, you must configure your collection for Range indexing on strings.

```
DocumentCollection collection = new DocumentCollection { Id = "orders" };  
collection.IndexingPolicy = new IndexingPolicy(new RangeIndex(DataType.String) { Precision = -1 });  
await client.CreateDocumentCollectionAsync("/dbs/orderdb", collection);
```

You can learn more about how to configure indexing policies at [Azure Cosmos DB Indexing Policies](#).

## 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 = client.CreateDocumentQuery<Order>("/dbs/orderdb/colls/orders")  
    .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"] >= "2016-12-18T21:55:03.45569Z")
```

You can learn more about Azure Cosmos DB's SQL query language and the LINQ provider at [Querying Cosmos DB](#).

In this article, we looked at how to store, index, and query DateTimes in Azure Cosmos DB.

## Next Steps

- Download and run the [Code samples on GitHub](#)
- Learn more about [SQL queries](#)
- Learn more about [Azure Cosmos DB Indexing Policies](#)

# Time to Live (TTL) in Azure Cosmos DB

2/12/2020 • 3 minutes to read • [Edit Online](#)

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.

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.

## 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 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

12/5/2019 • 3 minutes to read • [Edit Online](#)

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.

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
Fabrikam		Duperre	gaby@fabrikam.com
Fabrikam			gaby@fabrikam.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

12/4/2019 • 5 minutes to read • [Edit Online](#)

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
Paginated Query	Read	Multi-item transaction

OPERATION	OPERATION TYPE	SINGLE OR 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 guarantees 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](#)
- [Stored procedures, triggers, and user-defined functions](#)

# Change feed in Azure Cosmos DB - overview

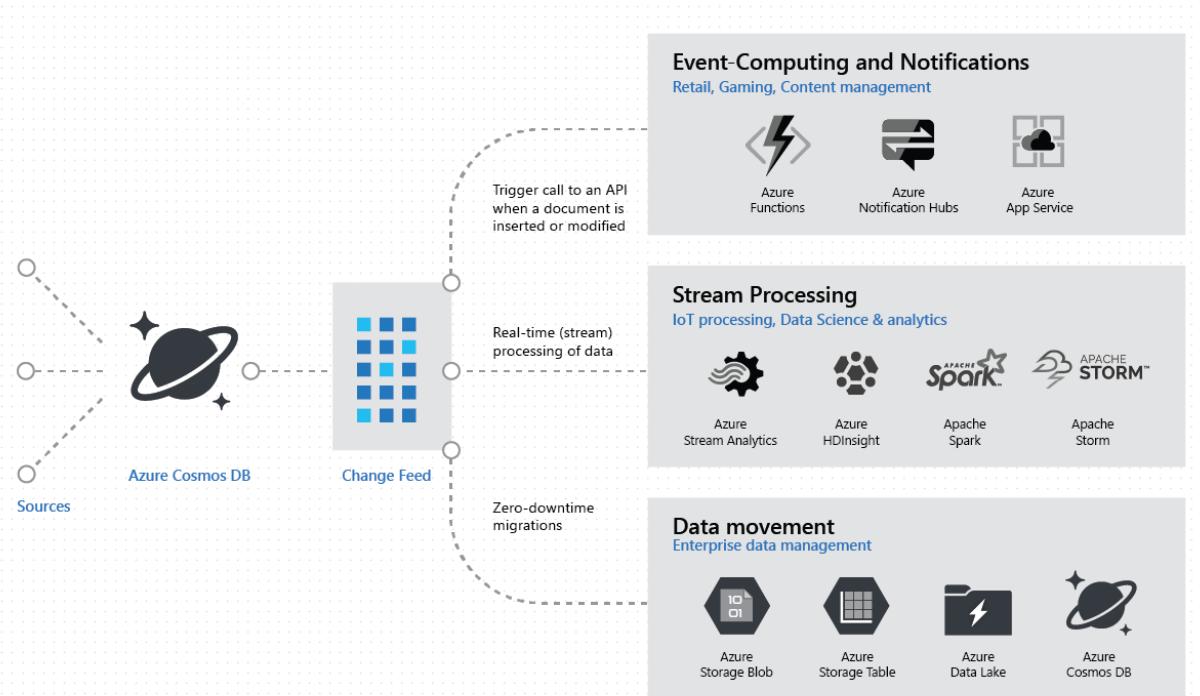
12/13/2019 • 7 minutes to read • [Edit Online](#)

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 changes are persisted, can be processed asynchronously and incrementally, and the output can be distributed across one or more consumers for parallel processing.

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.
- Additional data movement by either synchronizing with a cache or a search engine or a data warehouse or archiving data to 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:



## Supported APIs and client SDKs

This feature is currently supported by the following Azure Cosmos DB APIs and client SDKs.

CLIENT DRIVERS	AZURE CLI	SQL API	AZURE COSMOS DB'S API FOR CASSANDRA	AZURE COSMOS DB'S API FOR MONGODB	GREMLIN API	TABLE API
.NET	NA	Yes	Yes	Yes	Yes	No

CLIENT DRIVERS	AZURE CLI	SQL API	AZURE COSMOS DB'S API FOR CASSANDRA	AZURE COSMOS DB'S API FOR MONGODB	GREMLIN API	TABLE API
Java	NA	Yes	Yes	Yes	Yes	No
Python	NA	Yes	Yes	Yes	Yes	No
Node/JS	NA	Yes	Yes	Yes	Yes	No

## Change feed and different operations

Today, you see all operations in the change feed. The functionality where you can control change feed, for specific operations such as updates only and not inserts is not yet available. You can 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. If the item is not deleted you can read the change feed as far as the origin of your container.

### 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.

### 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 is used for concurrency control tells about the version of the item, whereas ETag is used for sequencing the feed.

## Change feed use cases and scenarios

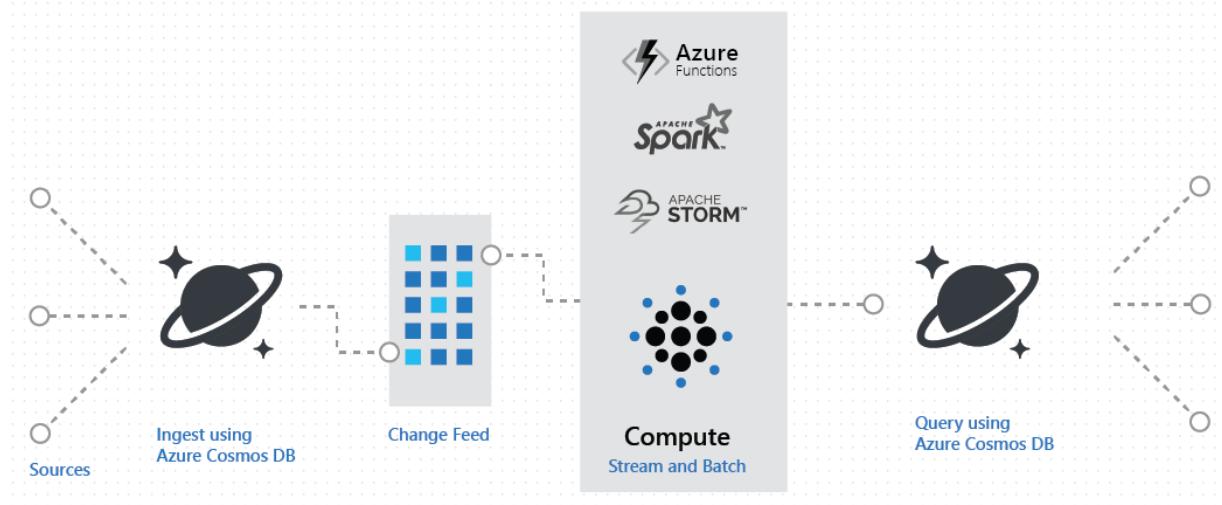
Change feed enables efficient processing of large datasets with a high volume of writes. Change feed also offers an alternative to querying an entire dataset to identify what has changed.

### Use cases

For example, with change feed you can perform the following tasks efficiently:

- Update a cache, update a search index, or update a data warehouse with data stored in Azure Cosmos DB.
- Implement an application-level data tiering and archival, for example, store "hot data" in Azure Cosmos DB and age out "cold data" to other storage systems, for example, [Azure Blob Storage](#).

- Perform zero down-time migrations to another Azure Cosmos account or another Azure Cosmos container with a different logical partition key.
- Implement [Lambda architecture](#) using Azure Cosmos DB, where Azure Cosmos DB supports both real-time, batch and query serving layers, thus enabling lambda architecture with low TCO.
- Receive and store event data from devices, sensors, infrastructure and applications, and process these events in real time, for example, using [Spark](#). The following image shows how you can implement lambda architecture using Azure Cosmos DB via change feed:



## Scenarios

The following are some of the scenarios you can easily implement with change feed:

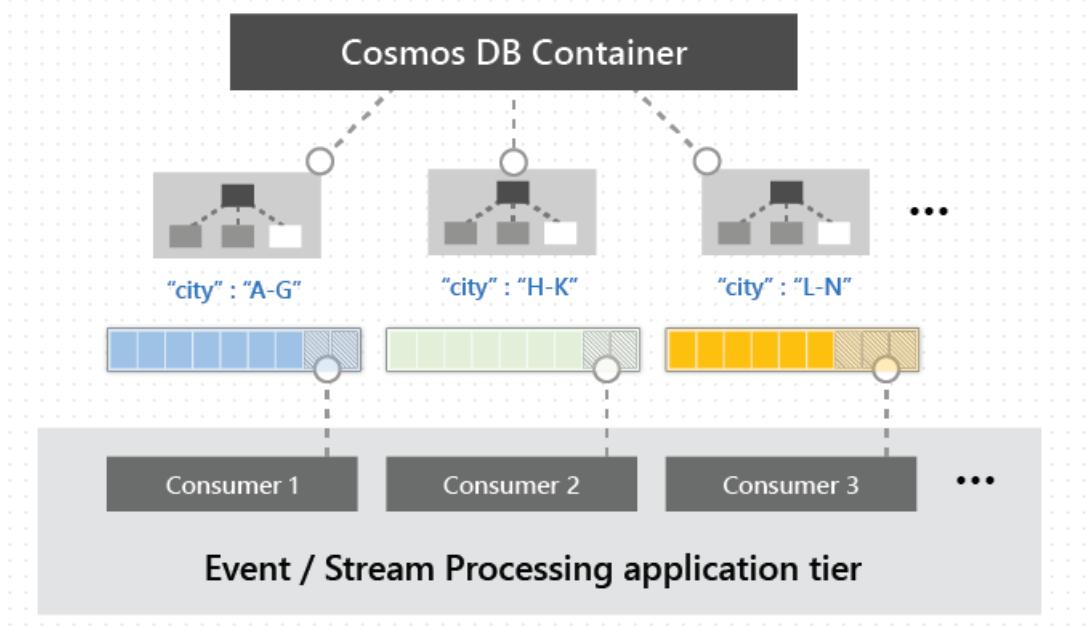
- Within your [serverless](#) web or mobile apps, you can track events such as all the changes to your customer's profile, preferences, or their location and trigger certain actions, for example, sending push notifications to their devices using [Azure Functions](#).
- If you're using Azure Cosmos DB to build a game, you can, for example, use change feed to implement real-time leaderboards based on scores from completed games.

## 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 log, 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, wins 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](#)

# Reading Azure Cosmos DB change feed

12/2/2019 • 2 minutes to read • [Edit Online](#)

You can work with the Azure Cosmos DB change feed using any of the following options:

- Using Azure Functions
- Using the change feed processor library
- Using the Azure Cosmos DB SQL API SDK

## Using Azure Functions

Azure Functions is the simplest and recommended option. When you create an Azure Functions trigger for Cosmos DB, you can select the container to connect, and the Azure Function gets triggered whenever there is a change to the container. Triggers can be created by using the Azure Functions portal, the Azure Cosmos DB portal or programmatically with 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.

## Using the change feed processor library

The change feed processor library hides complexity and still gives you a complete control of the change feed. The library follows the observer pattern, where your processing function is called by the library. If you have a high throughput change feed, you can instantiate multiple clients to read the change feed. Because you're using change feed processor library, it will automatically divide the load among the different clients without you having to implement this logic. All the complexity is handled by the library. If you want to have your own load balancer, then you can implement `IPartitionLoadBalancingStrategy` for a custom partition strategy to process change feed. To learn more, see [using change feed processor library](#).

## Using the Azure Cosmos DB SQL API SDK

With the SDK, you get a low-level control of the change feed. You can manage the checkpoint, access a particular logical partition key, etc. If you have multiple readers, you can use `ChangeFeedOptions` to distribute read load to different threads or different clients.

## 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:

- [Overview of change feed](#)
- [Using change feed with Azure Functions](#)
- [Using change feed processor library](#)

# Change feed processor in Azure Cosmos DB

2/24/2020 • 4 minutes to read • [Edit Online](#)

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.

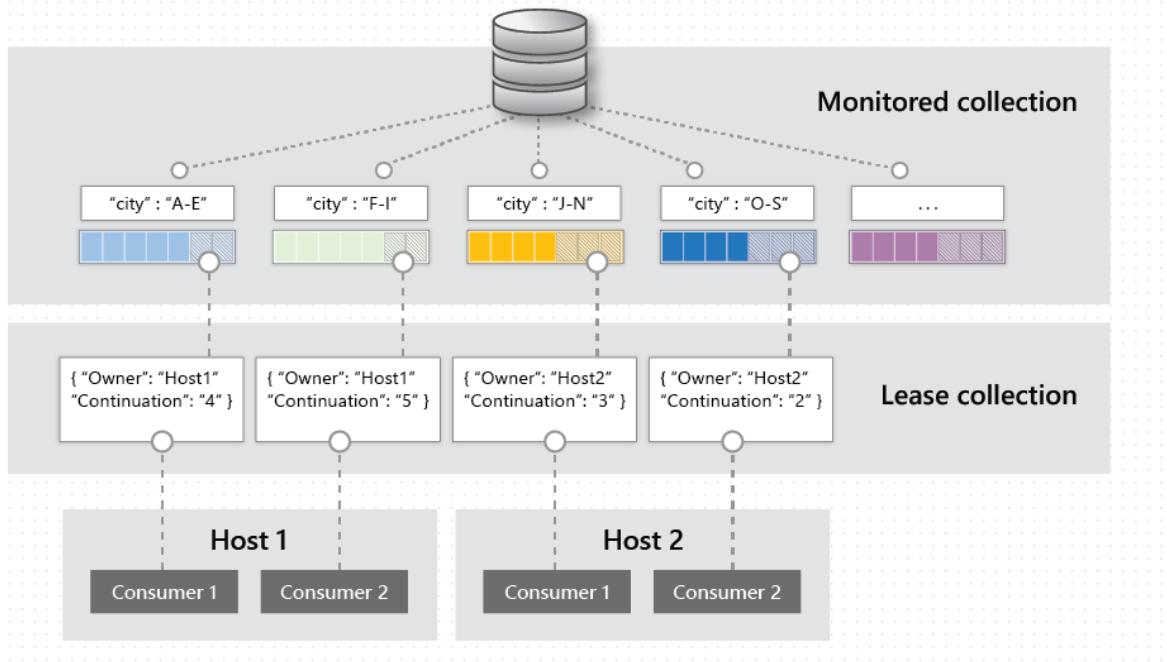
The main benefit of change feed processor library is its fault-tolerant behavior that assures an "at-least-once" delivery of all the events in the change feed.

## 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 HandlerChangesAsync
/// 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>("changeFeedSample", HandleChangesAsync)
        .WithInstanceId("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, it will retry that batch.

## Dynamic scaling

As mentioned during the introduction, the change feed processor can distribute compute across multiple instances automatically. You can deploy multiple instances of your application using the change feed processor and take advantage of it, the only key requirements are:

1. All instances should have the same lease container configuration.
2. All instances should have the same workflow name.
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 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

You are charged for RUs consumed, since data movement in and out of Cosmos containers always consumes RUs. You are charged for RUs consumed by the lease container.

## Additional resources

- [Azure Cosmos DB SDK](#)
- [Usage samples on GitHub](#)
- [Additional samples on GitHub](#)

## Next steps

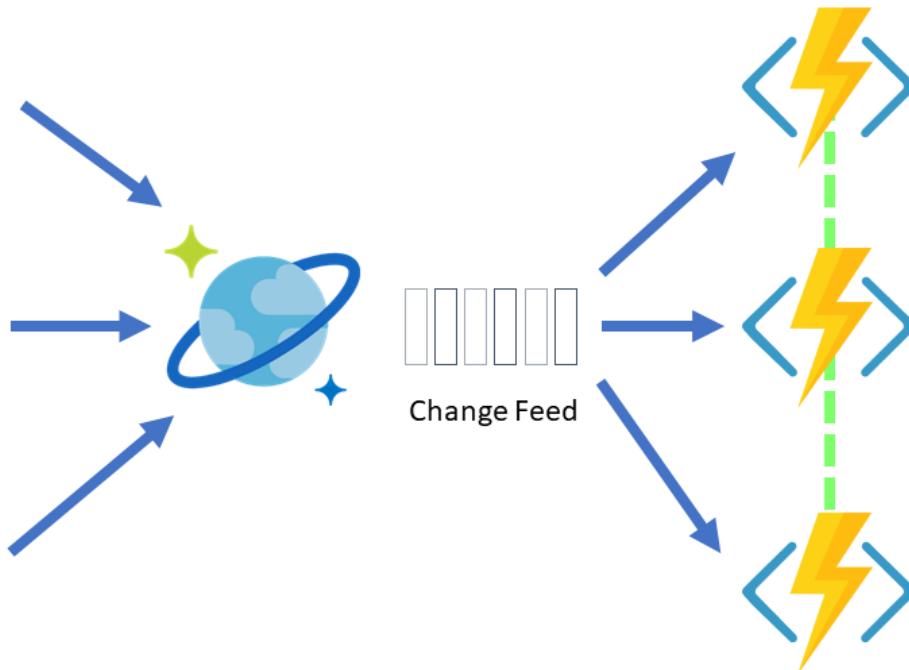
You can now proceed to learn more about change feed processor in the following articles:

- [Overview of change feed](#)
- [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

2/25/2020 • 2 minutes to read • [Edit Online](#)

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 Core 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](#)

# Use-cases for built-in analytics with Azure Cosmos DB

10/21/2019 • 4 minutes to read • [Edit Online](#)

The following use-cases can be achieved by using the built-in analytics with Apache Spark in Azure Cosmos DB:

## HTAP scenarios

Built-in analytics in Azure Cosmos DB can be used to:

- Detect fraud during transaction processing.
- Provide recommendations to shoppers as they browse an ECommerce store.
- Alert operators to the impending failure of a critical piece of manufacturing equipment.
- Create fast, actionable insight by embedding real-time analytics directly on operational data.

Azure Cosmos DB supports Hybrid Transactional/Analytical Processing (HTAP) scenarios by using the natively built-in Apache Spark. Azure Cosmos DB removes the operational and analytical separation that comes with the traditional systems.

## Globally distributed data lake without requiring any ETL

With natively built-in Apache Spark, Azure Cosmos DB provides a fast, simple, and scalable way to build a globally distributed data lake that provides a single system image. The globally distributed multi-model data eliminates the need to invest in expensive ETL pipelines and scales on-demand, revolutionizing the way data teams analyze their data sets.

## Time series analytics over historic data

In some cases, you may need to answer questions based on data as at a specific point in time over events completed in the past. For example, to get the count of CRM activity statuses at a certain date. If you ran the report a week ago, the count of the statuses would be as per the statuses of each activity at that point in time. Running the same report today will give you the count of the activities whose statuses are as they are today, which may have changed since last week, as they go through their life cycle from open to close. So, you need to report on the snapshot at each stage of the case's life cycle.

In traditional data warehouse scenarios, the concept of snapshot isn't possible because the data warehouses aren't designed to incorporate it and the data only provides a current view of what's happening. With Azure Cosmos DB, users have the possibility to implement the concept of time travel, being able to query and run analytics on the data retrospectively and ask for how the data looked at a specific point of time in the history. This means users can easily view both the current and historic views of the data and run analytics on it.

## Globally distributed machine learning and AI

As businesses contend with quickly growing volumes of data and an expanding variety of data types and formats, the ability to gain deeper and more accurate insights becomes near impossible at petabyte scale across the globe. With natively built-in Apache Spark, Azure Cosmos DB provides a globally distributed analytics platform that offers extensive library of machine learning algorithms. You can use interactive Jupyter notebooks to build and train models, and cluster management capabilities. These capabilities enable you to provision highly tuned and auto-elastic Spark clusters on-demand.

## Deep Learning on multi-model globally distributed data

Deep learning is the ideal way to provide big data predictive analytics solutions as data volume and complexity continues to grow. With deep learning, businesses can harness the power of unstructured and semi-structured data to deliver use cases that leverage techniques like AI, natural language processing, and more.

## Reporting (integrating with Power Apps, Power BI)

Power BI provides interactive visualizations with self-service business intelligence capabilities, enabling end users to create reports and dashboards on their own. By using the built-in Spark connector, you can connect Power BI Desktop to Apache Spark clusters in Azure Cosmos DB. This connector lets you use direct query to offload processing to Apache Spark in Azure Cosmos DB, which is great when you have a massive amount of data that you don't want to load into Power BI or when you want to perform near real-time analysis.

## IoT analytics at global scale

The data generated from increasing network sensors brings an unprecedented visibility into previously opaque systems and processes. The key is to find actionable insights in this torrent of information regardless of where IoT devices are distributed around the globe. Azure Cosmos DB allows IOT companies to analyze high-velocity sensor, and time-series data in real-time anywhere around the world. It allows you to harness the true value of an interconnected world to deliver improved customer experiences, operational efficiencies, and new revenue opportunities.

## Stream processing and event analytics

From log files to sensor data, businesses increasingly have the need to cope with "streams" of data. This data arrives in a steady stream, often from multiple sources simultaneously. While it is feasible to store these data streams on disk and analyze them retrospectively, it can sometimes be sensible or important to process and act upon the data as it arrives. For example, streams of data related to financial transactions can be processed in real time to identify and refuse potentially fraudulent transactions.

## Interactive analytics

In addition to running pre-defined queries to create static dashboards for sales, productivity or stock prices, you may want to explore the data interactively. Interactive analytics allow you to ask questions, view the results, alter the initial question based on the response or drill deeper into results. Apache Spark in Azure Cosmos DB supports interactive queries by responding and adapting quickly.

## Data exploration using Jupyter notebooks

When you have a new dataset, before you dive into running models and tests, you need to inspect your data. In other words, you need to perform exploratory data analysis. Data exploration can inform several decisions. For example, you can find details such as the methods that are appropriate to use on your data, whether the data meets certain modeling assumptions, whether the data should be cleaned, restructured etc. By using the Azure Cosmos DB's natively built-in Jupyter notebooks and Apache Spark, you can do quick and effective exploratory data analysis on transactional and analytical data.

## Next steps

To get started with these use cases on go to the following articles:

- [Built-in Jupyter notebooks in Azure Cosmos DB](#)
- [How to enable notebooks for Azure Cosmos accounts](#)
- [Create a notebook to analyze and visualize data](#)

# Solutions using globally distributed analytics in Azure Cosmos DB

11/6/2019 • 3 minutes to read • [Edit Online](#)

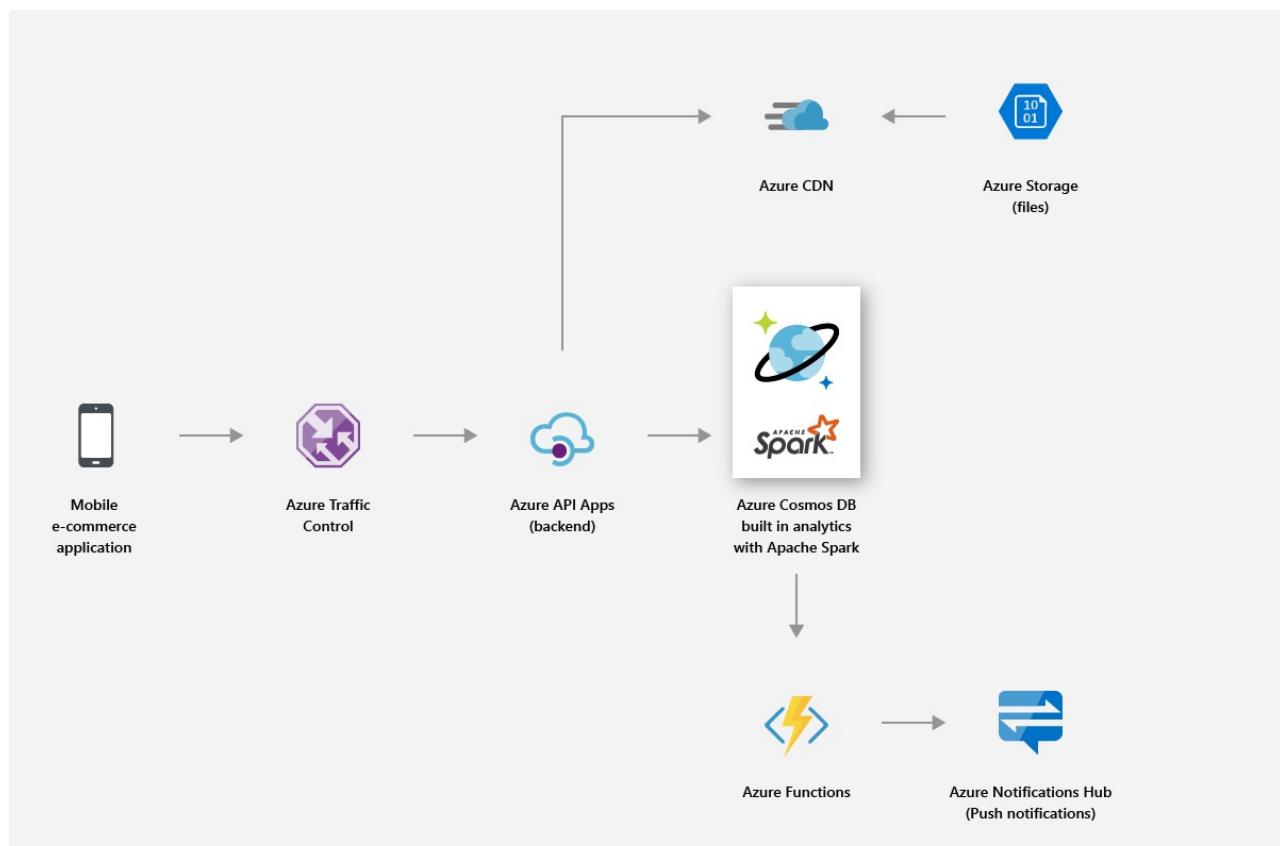
This article describes the solutions that can be built using the globally distributed analytics in Azure Cosmos DB.

## Retail and consumer goods

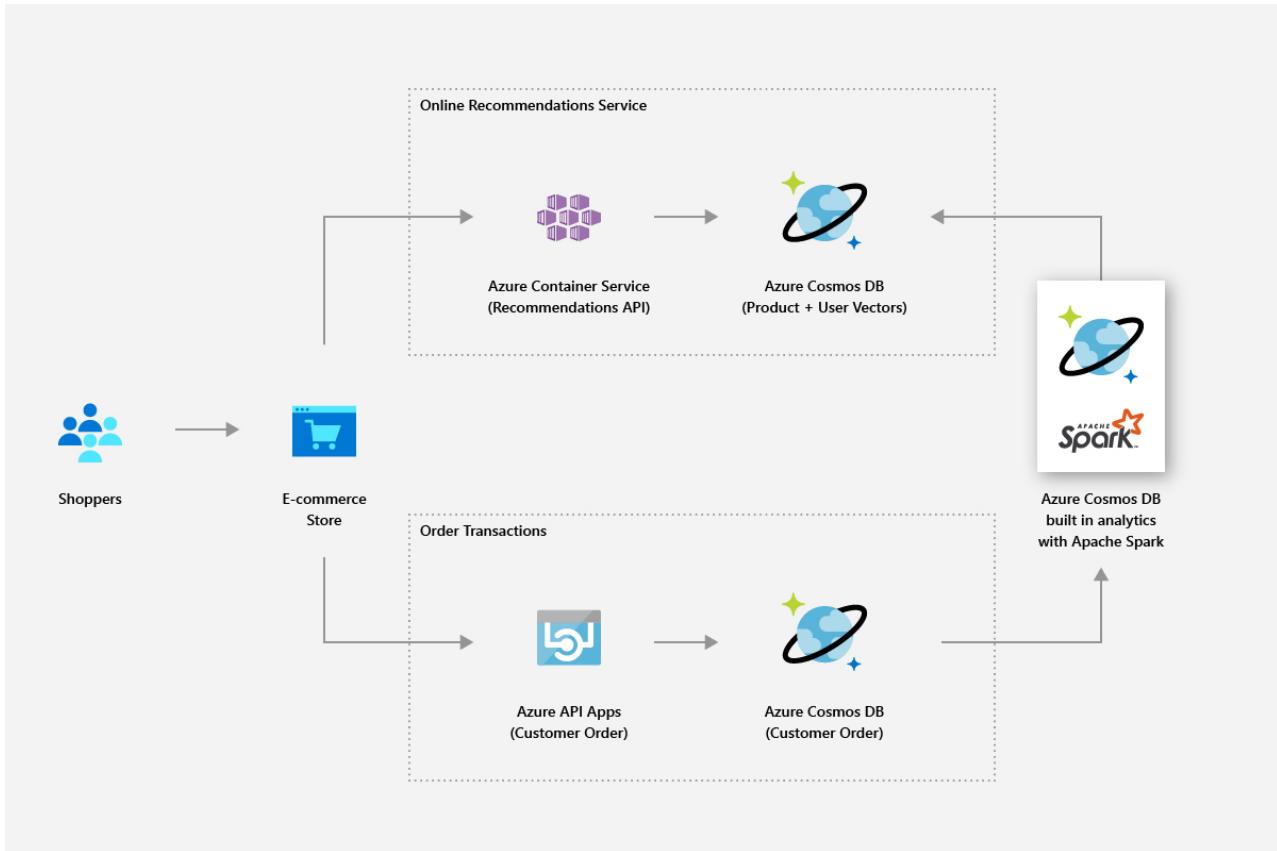
You can use Spark support in Azure Cosmos DB to deliver real-time recommendations and offers. You can help customers discover the items they will need with real-time personalization and product recommendations.

- You can use the built-in Machine Learning support provided by the Apache Spark runtime to generate real-time recommendations across the product catalogs.
- You can mine click stream data, purchase data, and customer data to provide targeted recommendations that drive lifetime value.
- Using the Azure Cosmos DB's global distribution feature, high volumes of product data that is spread across regions can be analyzed in milliseconds.
- You can quickly get insights for the geographically distributed users and data. You can improve the promotion conversion rate by serving the right ad to the right user at the right time.
- You can leverage the inbuilt Spark streaming capability to enrich live data by combining it with static customer data. This way you can deliver more personalized and targeted ads in real time and in context with what customers are doing.

The following image shows how Azure Cosmos DB Spark support is used to optimize pricing and promotions:



The following image shows how Azure Cosmos DB support is used in real-time recommendation engine:

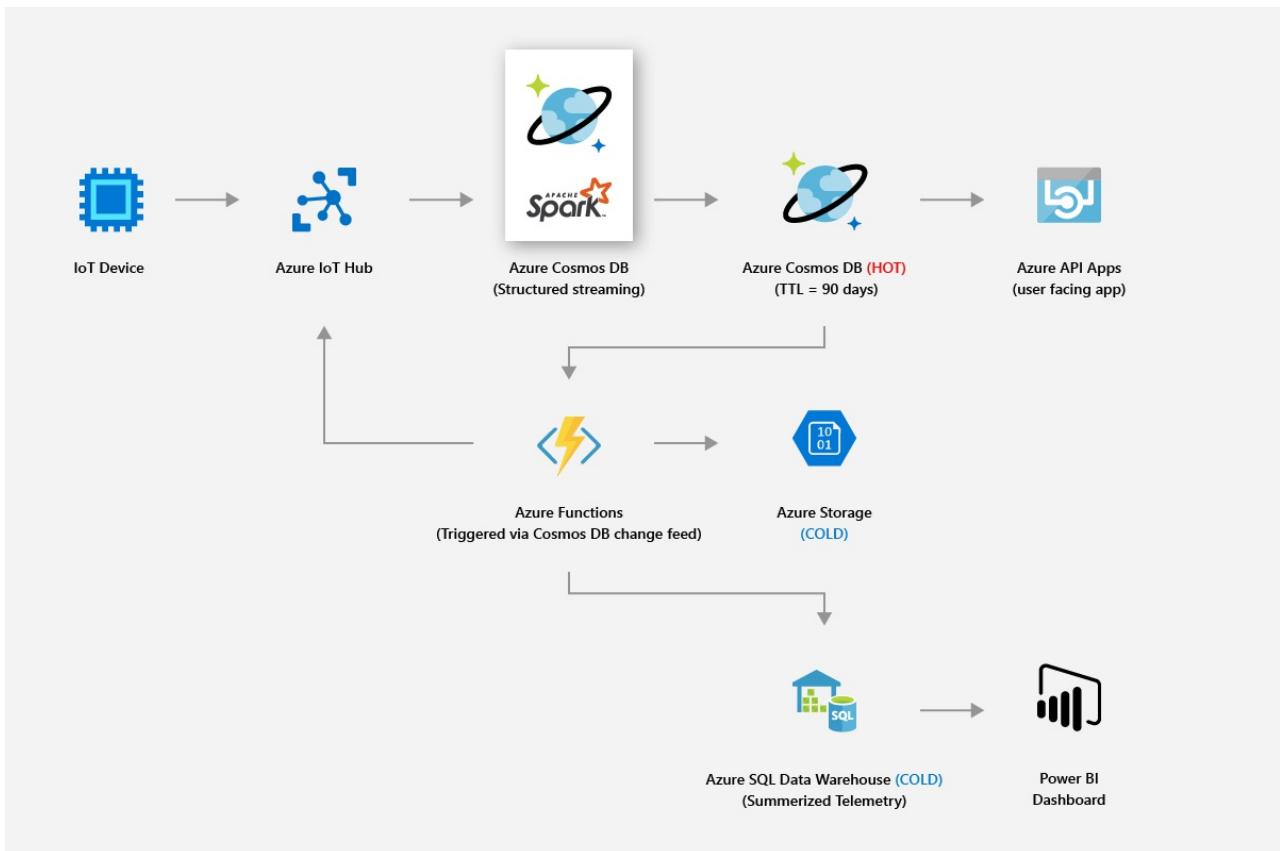


## Manufacturing and IoT

Azure Cosmos DB's in-built analytics platform allows you to enable real-time analysis of IoT data from millions of devices at global scale. You can make modern innovations like predicting weather patterns, predictive analysis, and energy optimizations.

- By using Azure Cosmos DB, you can mine data such as real-time asset metrics and weather factors, then apply smart grid analytics to optimize performance of connected devices in the field. Smart grid analytics is the key to control operating costs, to improve grid reliability, and deliver personalized energy services to consumers.

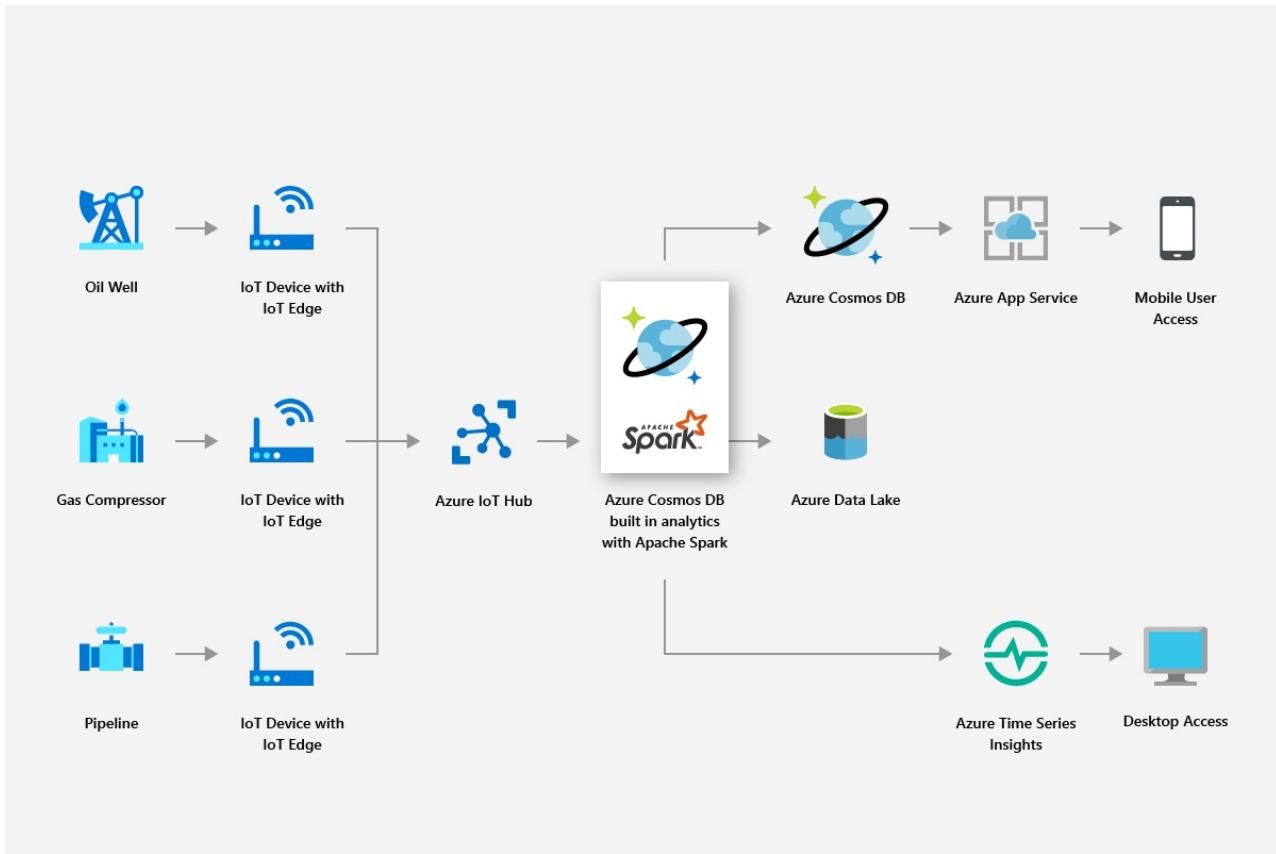
The following image shows how Azure Cosmos DB's Spark support is used to read metrics from IoT devices and apply smart grid analytics:



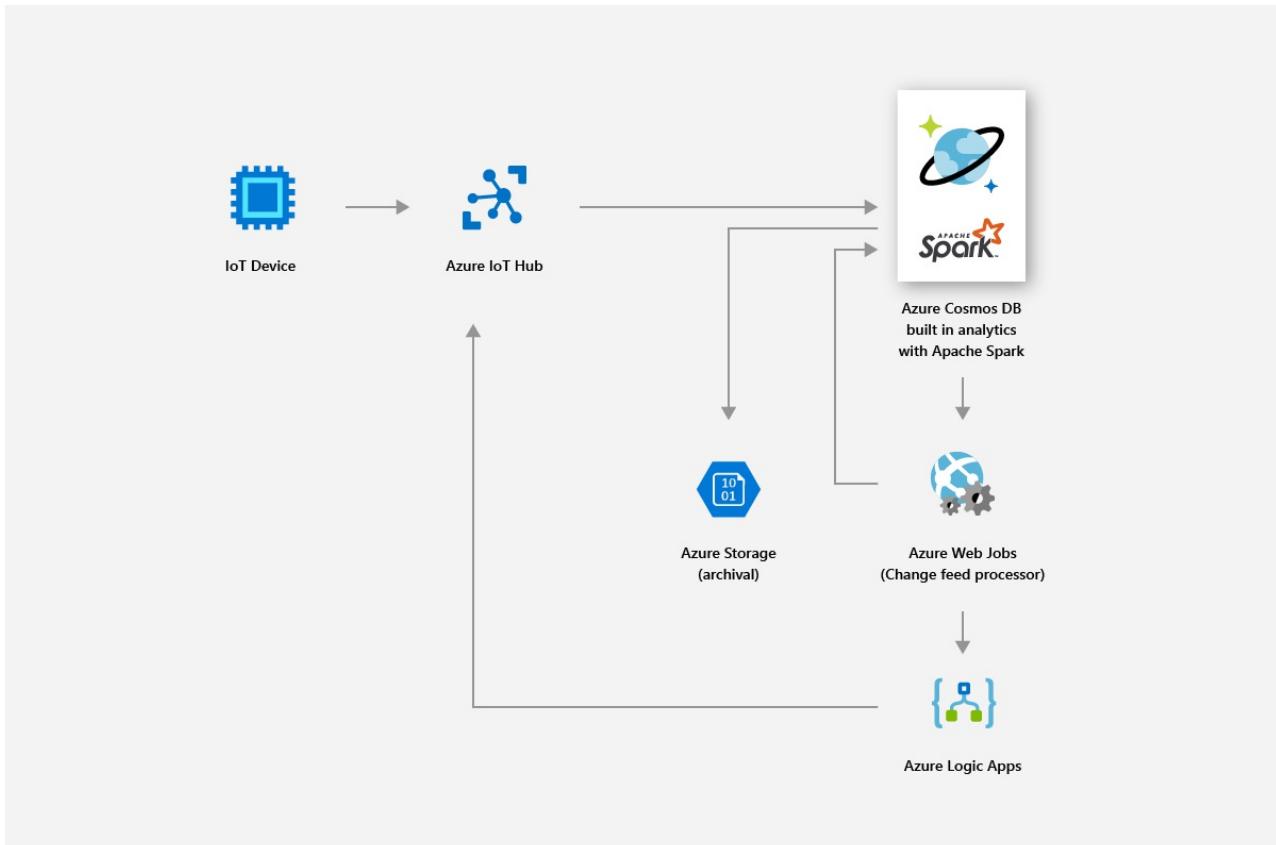
## Predictive maintenance

- Maintaining assets such as compressors that are used in small drilling rigs to deep-water platforms is a complex endeavor. These assets are located across the globe and generate petabytes of data. By using Azure Cosmos DB, you can build an end-to-end predictive data pipeline that uses Spark streaming to process large amounts of sensor telemetry, store asset parts, and sensor mappings data.
- You can build and deploy machine learning models to predict asset failures before they happen and issue maintenance work orders before the failure occurs.

The following image shows how Azure Cosmos DB's Spark support is used to build a predictive maintenance system:



The following image shows how Azure Cosmos DB's Spark support is used to build a real-time vehicle diagnostic system:

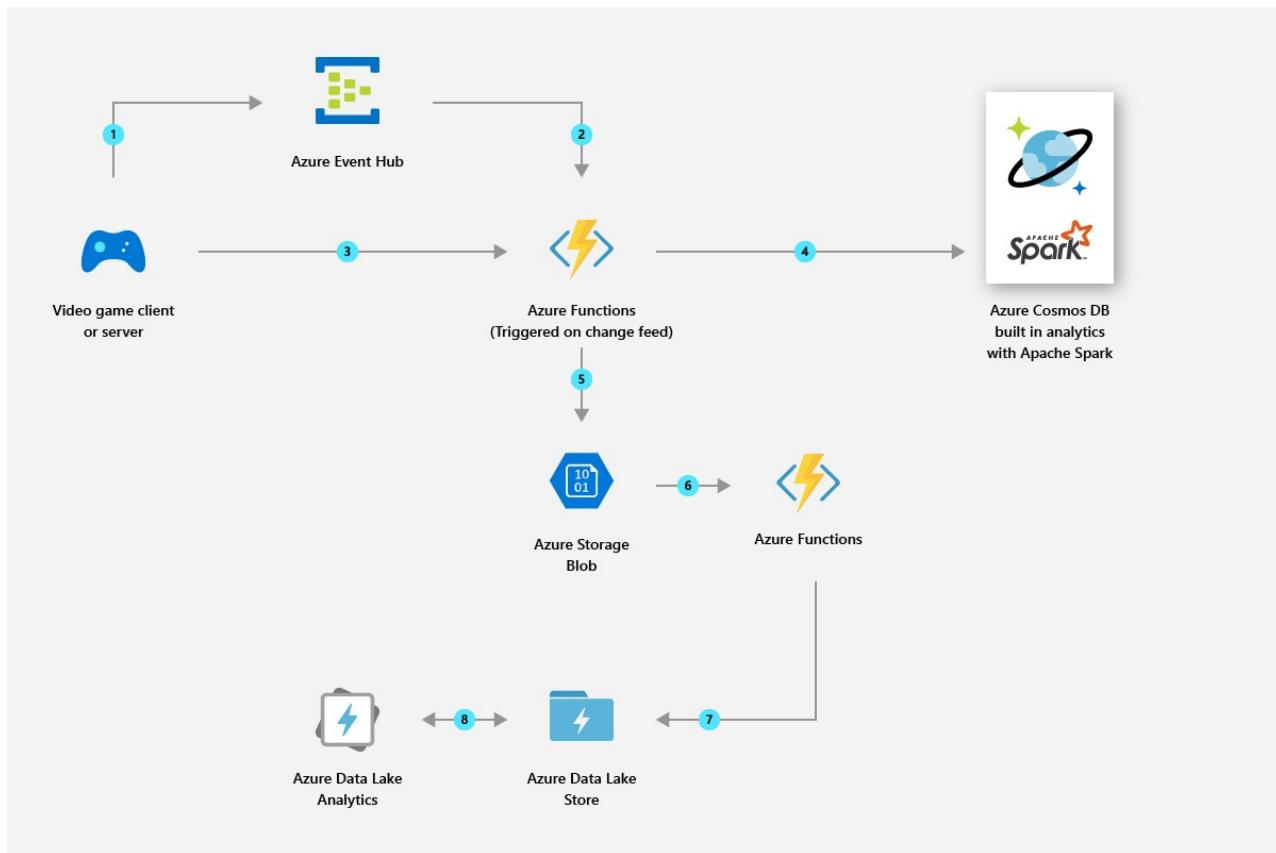


## Gaming

- With built-in Spark support, Azure Cosmos DB enables you to easily build, scale, and deploy advanced analytics and machine learning models in minutes to build the best gaming experience possible.

- You can analyze player, purchase, and behavioral data to create relevant personalized offers to attain high conversion rates.
- Using Spark machine learning, you can analyze and gain insights on game telemetry data. You can diagnose and prevent slow load times and in-game issues.

The following image shows how Azure Cosmos DB's Spark support is used in gaming analytics:



## Next steps

- To learn about the benefits of Azure Cosmos DB, see the [overview](#) article.
- [Get started with the Azure Cosmos DB API for MongoDB](#)
- [Get started with the Azure Cosmos DB Cassandra API](#)
- [Get started with the Azure Cosmos DB Gremlin API](#)
- [Get started with the Azure Cosmos DB Table API](#)

# Azure Cosmos DB: Implement a lambda architecture on the Azure platform

1/24/2020 • 10 minutes to read • [Edit Online](#)

Lambda architectures enable efficient data processing of massive data sets. Lambda architectures use batch-processing, stream-processing, and a serving layer to minimize the latency involved in querying big data.

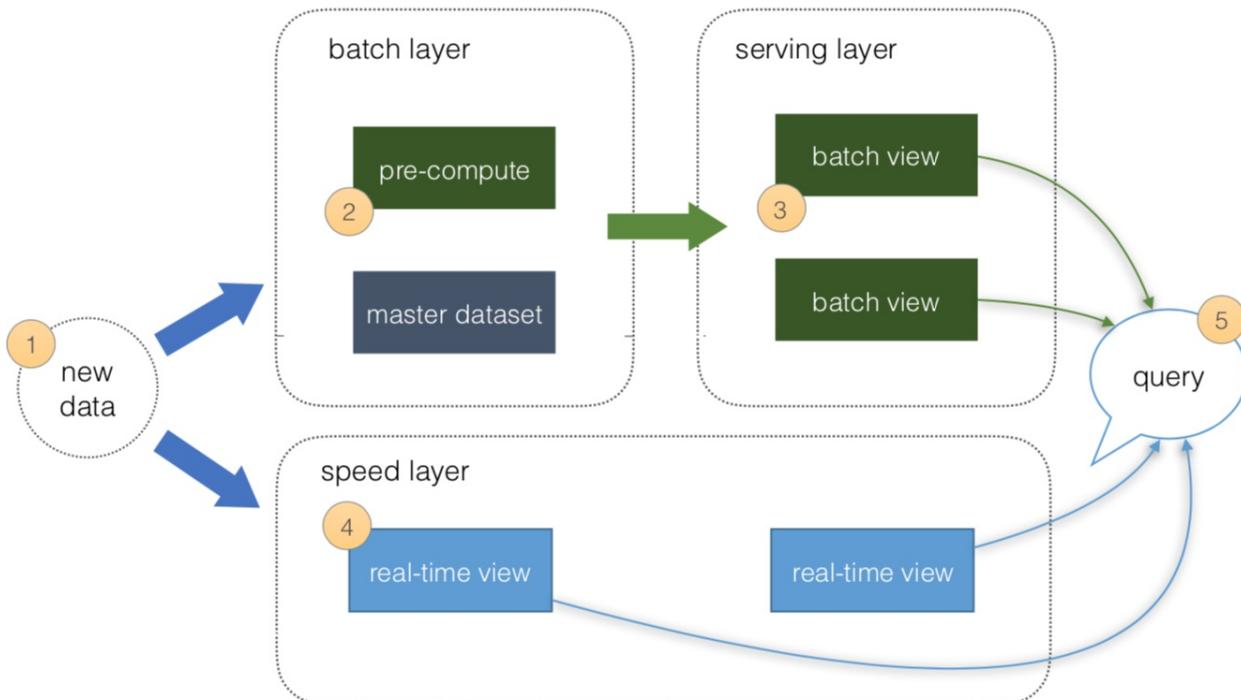
To implement a lambda architecture on Azure, you can combine the following technologies to accelerate real-time big data analytics:

- [Azure Cosmos DB](#), the industry's first globally distributed, multi-model database service.
- [Apache Spark for Azure HDInsight](#), a processing framework that runs large-scale data analytics applications
- Azure Cosmos DB [change feed](#), which streams new data to the batch layer for HDInsight to process
- The [Spark to Azure Cosmos DB Connector](#)

This article describes the fundamentals of a lambda architecture based on the original multi-layer design and the benefits of a "rearchitected" lambda architecture that simplifies operations.

## What is a lambda architecture?

A lambda architecture is a generic, scalable, and fault-tolerant data processing architecture to address batch and speed latency scenarios as described by [Nathan Marz](#).



Source: <http://lambda-architecture.net/>

The basic principles of a lambda architecture are described in the preceding diagram as per <http://lambda-architecture.net>.

1. All **data** is pushed into *both* the **batch layer** and **speed layer**.
2. The **batch layer** has a **master dataset** (immutable, append-only set of raw data) and pre-computes the batch views.

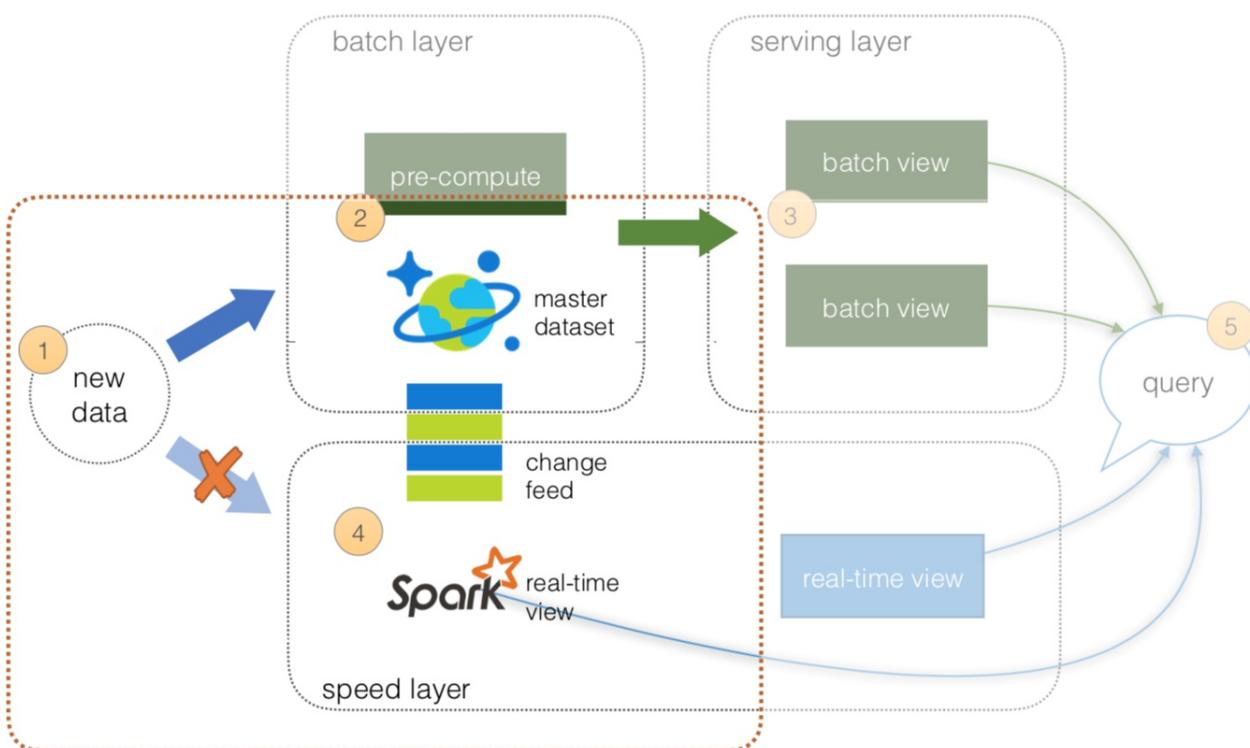
3. The **serving layer** has batch views for fast queries.
4. The **speed layer** compensates for processing time (to the serving layer) and deals with recent data only.
5. All queries can be answered by merging results from batch views and real-time views or pinging them individually.

Upon further reading, we will be able to implement this architecture using only the following:

- Azure Cosmos container(s)
- HDInsight (Apache Spark 2.1) cluster
- Spark Connector 1.0

## Speed layer

From an operations perspective, maintaining two streams of data while ensuring the correct state of the data can be a complicated endeavor. To simplify operations, utilize the [Azure Cosmos DB change feed support](#) to keep the state for the *batch layer* while revealing the Azure Cosmos DB change log via the *Change Feed API* for your *speed layer*.



What's important in these layers:

1. All **data** is pushed *only* into Azure Cosmos DB, thus you can avoid multi-casting issues.
2. The **batch layer** has a master dataset (immutable, append-only set of raw data) and pre-computes the batch views.
3. The **serving layer** is discussed in the next section.
4. The **speed layer** utilizes HDInsight (Apache Spark) to read the Azure Cosmos DB change feed. This enables you to persist your data as well as to query and process it concurrently.
5. All queries can be answered by merging results from batch views and real-time views or pinging them individually.

### Code Example: Spark structured streaming to an Azure Cosmos DB change feed

To run a quick prototype of the Azure Cosmos DB change feed as part of the **speed layer**, can test it out using Twitter data as part of the [Stream Processing Changes using Azure Cosmos DB Change Feed and Apache Spark](#) example. To jump-start your Twitter output, see the code sample in [Stream feed from Twitter to Cosmos DB](#). With

In the preceding example, you're loading Twitter data into Azure Cosmos DB and you can then set up your HDInsight (Apache Spark) cluster to connect to the change feed. For more information on how to set up this configuration, see [Apache Spark to Azure Cosmos DB Connector Setup](#).

The following code snippet shows how to configure `spark-shell` to run a structured streaming job to connect to an Azure Cosmos DB change feed, which reviews the real-time Twitter data stream, to perform a running interval count.

```
// Import Libraries
import com.microsoft.azure.cosmosdb.spark._
import com.microsoft.azure.cosmosdb.spark.schema._
import com.microsoft.azure.cosmosdb.spark.config.Config
import org.codehaus.jackson.map.ObjectMapper
import com.microsoft.azure.cosmosdb.spark.streaming._
import java.time._

// Configure connection to Azure Cosmos DB Change Feed
val sourceConfigMap = Map(
  "Endpoint" -> "[COSMOSDB ENDPOINT]",
  "Masterkey" -> "[MASTER KEY]",
  "Database" -> "[DATABASE]",
  "Collection" -> "[COLLECTION]",
  "ConnectionMode" -> "Gateway",
  "ChangeFeedCheckpointLocation" -> "checkpointlocation",
  "changefeedqueryname" -> "Streaming Query from Cosmos DB Change Feed Interval Count")

// Start reading change feed as a stream
var streamData =
  spark.readStream.format(classOf[CosmosDBSourceProvider].getName).options(sourceConfigMap).load()

// Start streaming query to console sink
val query = streamData.withColumn("countcol", streamData.col("id").substr(0,
  0)).groupBy("countcol").count().writeStream.outputMode("complete").format("console").start()
```

For complete code samples, see [azure-cosmosdb-spark/lambda/samples](#), including:

- [Streaming Query from Cosmos DB Change Feed.scala](#)
- [Streaming Tags Query from Cosmos DB Change Feed.scala](#)

The output of this is a `spark-shell` console, which continuously runs a structured streaming job that performs an interval count against the Twitter data from the Azure Cosmos DB change feed. The following image shows the output of the stream job and the interval counts.

```

scala> val query = streamData.withColumn("countcol", streamData.col("id").substr(0, 0)).groupBy("countcol").count().writeStream
.outputMode("complete").format("console").start()
query: org.apache.spark.sql.streaming.StreamingQuery = org.apache.spark.sql.execution.streaming.StreamingQueryWrapper@39dfadd7

-----
Batch: 0
-----
+-----+----+
|countcol|count|
+-----+----+
|      | 13|
+-----+----+

-----
Batch: 1
-----
+-----+----+
|countcol|count|
+-----+----+
|      | 20|
+-----+----+

-----
Batch: 2
-----
|[Stage 16:----->      (9 + 1) / 10] |

+-----+----+
|countcol|count|
+-----+----+
|      | 24|
+-----+----+

-----
Batch: 3
-----
+-----+----+
|countcol|count|
+-----+----+
|      | 38|
+-----+----+

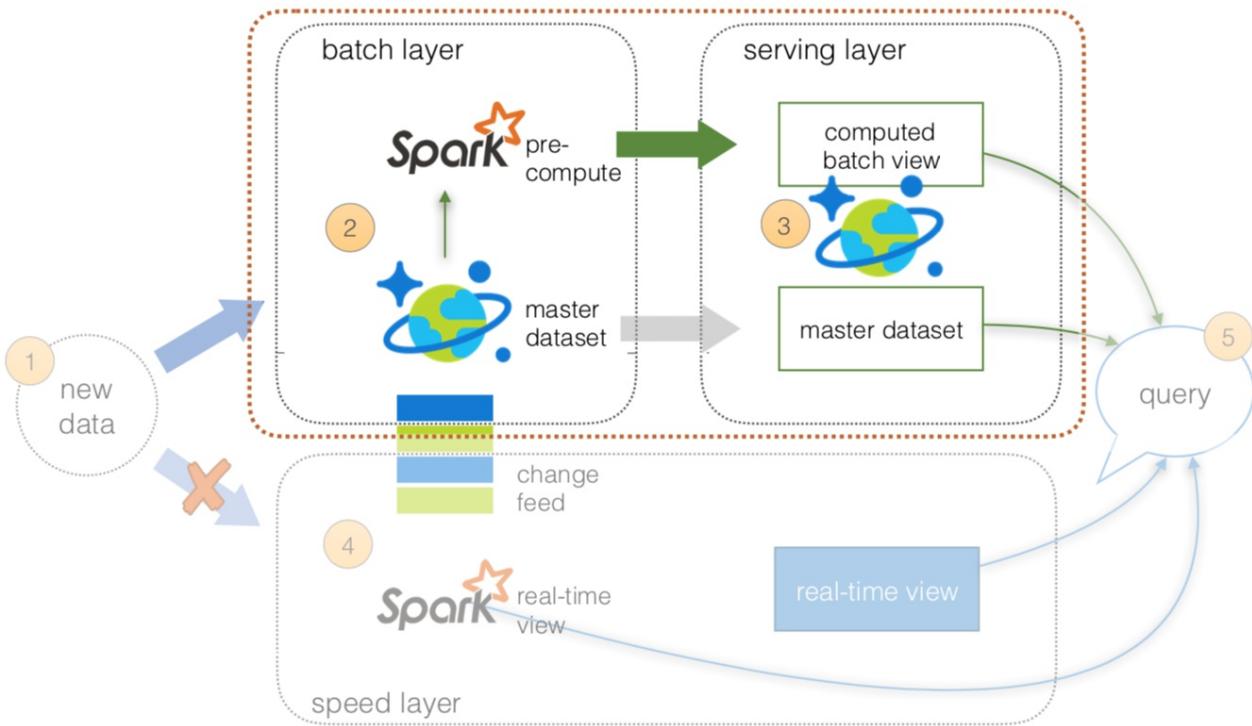
```

For more information on Azure Cosmos DB change feed, see:

- [Working with the change feed support in Azure Cosmos DB](#)
- [Introducing the Azure CosmosDB Change Feed Processor Library](#)
- [Stream Processing Changes: Azure CosmosDB change feed + Apache Spark](#)

## Batch and serving layers

Since the new data is loaded into Azure Cosmos DB (where the change feed is being used for the speed layer), this is where the **master dataset** (an immutable, append-only set of raw data) resides. From this point onwards, use HDInsight (Apache Spark) to perform the pre-compute functions from the **batch layer to serving layer**, as shown in the following image:



What's important in these layers:

1. All **data** is pushed only into Azure Cosmos DB (to avoid multi-cast issues).
2. The **batch layer** has a master dataset (immutable, append-only set of raw data) stored in Azure Cosmos DB. Using HDI Spark, you can pre-compute your aggregations to be stored in your computed batch views.
3. The **serving layer** is an Azure Cosmos database with collections for the master dataset and computed batch view.
4. The **speed layer** is discussed later in this article.
5. All queries can be answered by merging results from the batch views and real-time views, or pinging them individually.

#### Code example: Pre-computing batch views

To showcase how to execute pre-calculated views against your **master dataset** from Apache Spark to Azure Cosmos DB, use the following code snippets from the notebooks [Lambda Architecture Rearchitected - Batch Layer](#) and [Lambda Architecture Rearchitected - Batch to Serving Layer](#). In this scenario, use the Twitter data stored in Azure Cosmos DB.

Let's start by creating the configuration connection to the Twitter data within Azure Cosmos DB using the PySpark code below.

```

# Configuration to connect to Azure Cosmos DB
tweetsConfig = {
    "Endpoint" : "[Endpoint URL]",
    "Masterkey" : "[Master Key]",
    "Database" : "[Database]",
    "Collection" : "[Collection]",
    "preferredRegions" : "[Preferred Regions]",
    "SamplingRatio" : "1.0",
    "schema_samplesize" : "200000",
    "query_custom" : "[Cosmos DB SQL Query]"
}

# Create DataFrame
tweets = spark.read.format("com.microsoft.azure.cosmosdb.spark").options(**tweetsConfig).load()

# Create Temp View (to run Spark SQL statements)
tweets.createOrReplaceTempView("tweets")

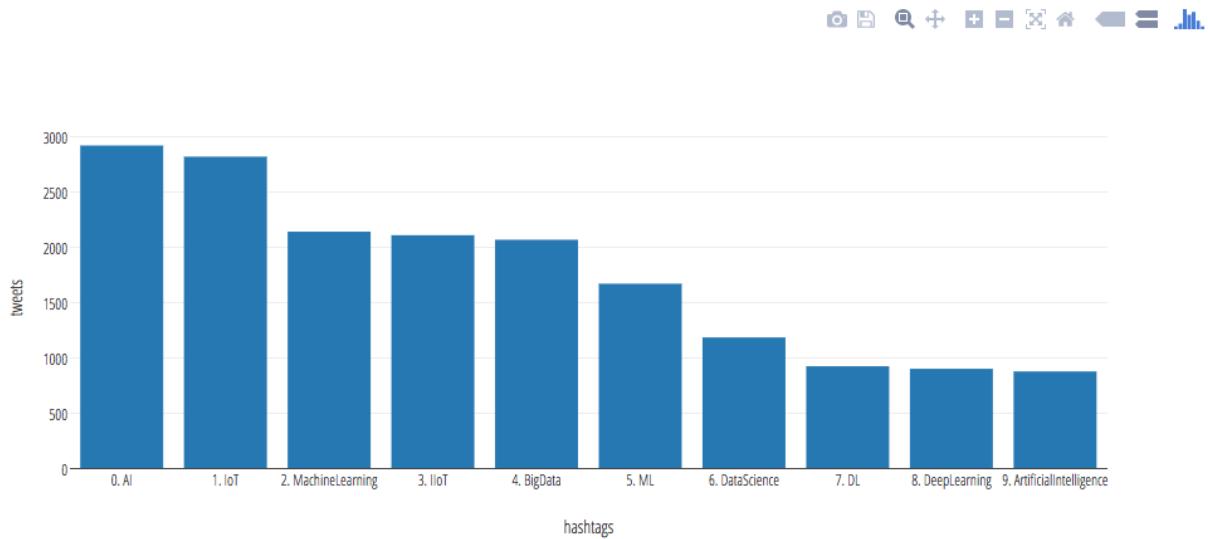
```

Next, let's run the following Spark SQL statement to determine the top 10 hashtags of the set of tweets. For this Spark SQL query, we're running this in a Jupyter notebook without the output bar chart directly following this code snippet.

```

%%sql
select hashtags.text, count(distinct id) as tweets
from (
    select
        explode(hashtags) as hashtags,
        id
    from tweets
) a
group by hashtags.text
order by tweets desc
limit 10

```



Now that you have your query, let's save it back to a collection by using the Spark Connector to save the output data into a different collection. In this example, use Scala to showcase the connection. Similar to the previous example, create the configuration connection to save the Apache Spark DataFrame to a different Azure Cosmos container.

```

val writeConfigMap = Map(
  "Endpoint" -> "[Endpoint URL]",
  "Masterkey" -> "[Master Key]",
  "Database" -> "[Database]",
  "Collection" -> "[New Collection]",
  "preferredRegions" -> "[Preferred Regions]",
  "SamplingRatio" -> "1.0",
  "schema_samplesize" -> "200000"
)

// Configuration to write
val writeConfig = Config(writeConfigMap)

```

After specifying the `SaveMode` (indicating whether to `Overwrite` or `Append` documents), create a `tweets_bytags` DataFrame similar to the Spark SQL query in the previous example. With the `tweets_bytags` DataFrame created, you can save it using the `write` method using the previously specified `writeConfig`.

```

// Import SaveMode so you can Overwrite, Append, ErrorIfExists, Ignore
import org.apache.spark.sql.{Row, SaveMode, SparkSession}

// Create new DataFrame of tweets tags
val tweets_bytags = spark.sql("select hashtags.text as hashtags, count(distinct id) as tweets from ( select explode(hashtags) as hashtags, id from tweets ) a group by hashtags.text order by tweets desc")

// Save to Cosmos DB (using Append in this case)
tweets_bytags.write.mode(SaveMode.Overwrite).cosmosDB(writeConfig)

```

This last statement now has saved your Spark DataFrame into a new Azure Cosmos container; from a lambda architecture perspective, this is your **batch view** within the **serving layer**.

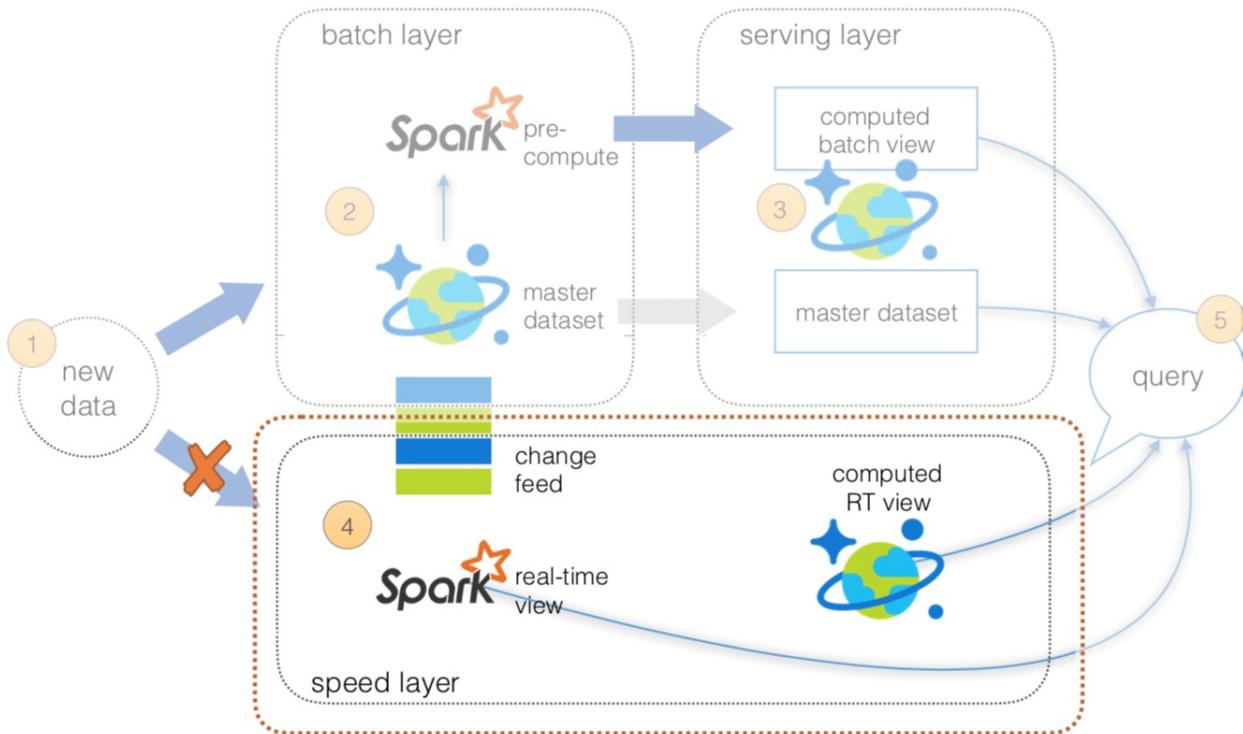
#### Resources

For complete code samples, see [azure-cosmosdb-spark/lambda/samples](#) including:

- Lambda Architecture Rearchitected - Batch Layer [HTML](#) | [ipynb](#)
- Lambda Architecture Rearchitected - Batch to Serving Layer [HTML](#) | [ipynb](#)

## Speed layer

As previously noted, using the Azure Cosmos DB Change Feed Library allows you to simplify the operations between the batch and speed layers. In this architecture, use Apache Spark (via HDInsight) to perform the *structured streaming* queries against the data. You may also want to temporarily persist the results of your structured streaming queries so other systems can access this data.



To do this, create a separate Azure Cosmos container to save the results of your structured streaming queries. This allows you to have other systems access this information not just Apache Spark. As well with the Cosmos DB Time-to-Live (TTL) feature, you can configure your documents to be automatically deleted after a set duration. For more information on the Azure Cosmos DB TTL feature, see [Expire data in Azure Cosmos containers automatically with time to live](#)

```

// Import Libraries
import com.microsoft.azure.cosmosdb.spark._
import com.microsoft.azure.cosmosdb.spark.schema._
import com.microsoft.azure.cosmosdb.spark.config.Config
import org.codehaus.jackson.map.ObjectMapper
import com.microsoft.azure.cosmosdb.spark.streaming._
import java.time._

// Configure connection to Azure Cosmos DB Change Feed
val sourceCollectionName = "[SOURCE COLLECTION NAME]"
val sinkCollectionName = "[SINK COLLECTION NAME]"

val configMap = Map(
  "Endpoint" -> "[COSMOSDB ENDPOINT]",
  "Masterkey" -> "[COSMOSDB MASTER KEY]",
  "Database" -> "[DATABASE NAME]",
  "Collection" -> sourceCollectionName,
  "ChangeFeedCheckpointLocation" -> "changefeedcheckpointlocation")

val sourceConfigMap = configMap.+(("changefeedqueryname", "Structured Stream replication streaming test"))

// Start to read the stream
var streamData =
  spark.readStream.format(classOf[CosmosDBSourceProvider].getName).options(sourceConfigMap).load()
val sinkConfigMap = configMap.-("collection").+(("collection", sinkCollectionName))

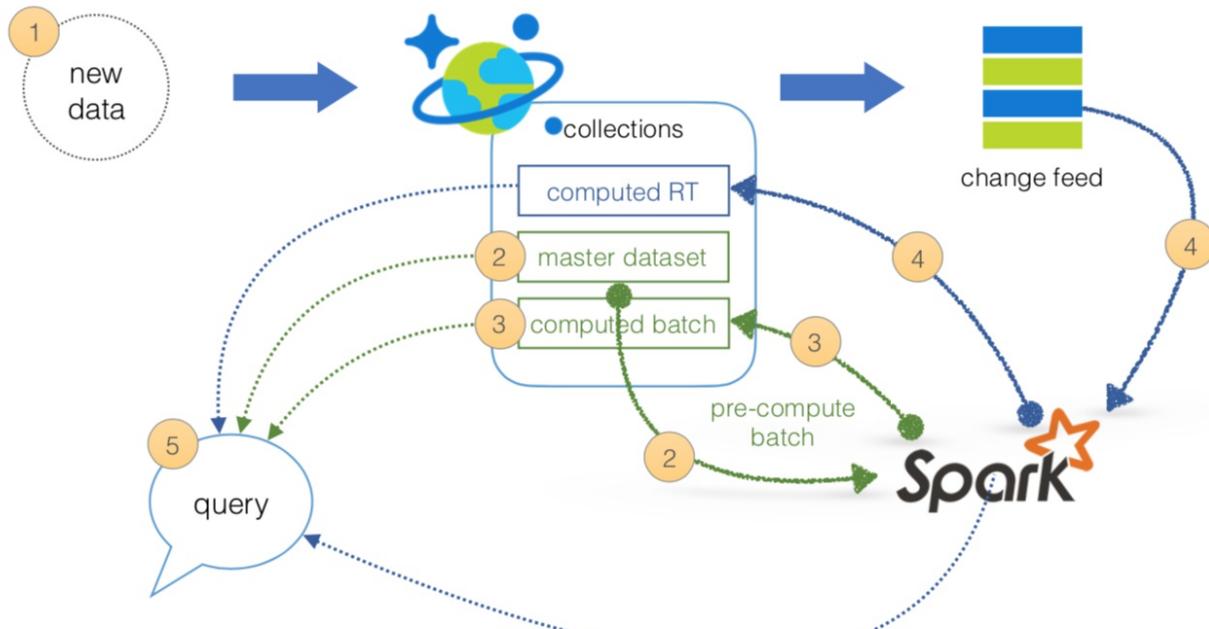
// Start the stream writer to new collection
val streamingQueryWriter =
  streamData.writeStream.format(classOf[CosmosDBSinkProvider].getName).outputMode("append").options(sinkConfigMap).option("checkpointLocation", "streamingcheckpointlocation")
var streamingQuery = streamingQueryWriter.start()

```

## Lambda architecture: Rearchitected

As noted in the previous sections, you can simplify the original lambda architecture by using the following components:

- Azure Cosmos DB
- The Azure Cosmos DB Change Feed Library to avoid the need to multi-cast your data between the batch and speed layers
- Apache Spark on HDInsight
- The Spark Connector for Azure Cosmos DB



With this design, you only need two managed services, Azure Cosmos DB and HDInsight. Together, they address the batch, serving, and speed layers of the lambda architecture. This simplifies not only the operations but also the data flow.

1. All data is pushed into Azure Cosmos DB for processing
2. The batch layer has a master dataset (immutable, append-only set of raw data) and pre-computes the batch views
3. The serving layer has batch views of data for fast queries.
4. The speed layer compensates for processing time (to the serving layer) and deals with recent data only.
5. All queries can be answered by merging results from batch views and real-time views.

## Resources

- **New data:** The [stream feed from Twitter to CosmosDB](#), which is the mechanism to push new data into Azure Cosmos DB.
- **Batch layer:** The batch layer is composed of the *master dataset* (an immutable, append-only set of raw data) and the ability to pre-compute batch views of the data that are pushed into the **serving layer**.
  - The [Lambda Architecture Rearchitected - Batch Layer](#) notebook [ipynb](#) | [html](#) queries the *master dataset* set of batch views.
- **Serving layer:** The **serving layer** is composed of pre-computed data resulting in batch views (for example aggregations, specific slicers, etc.) for fast queries.
  - The [Lambda Architecture Rearchitected - Batch to Serving Layer](#) notebook [ipynb](#) | [html](#) pushes the batch data to the serving layer; that is, Spark queries a batch collection of tweets, processes it, and stores it into another collection (a computed batch).
  - **Speed layer:** The **speed layer** is composed of Spark utilizing the Azure Cosmos DB change feed to read and act on immediately. The data can also be saved to *computed RT* so that other systems can query the processed real-time data as opposed to running a real-time query themselves.
  - The [Streaming Query from Cosmos DB Change Feed](#) scala script executes a streaming query from the Azure Cosmos DB change feed to compute an interval count from the spark-shell.
  - The [Streaming Tags Query from Cosmos DB Change Feed](#) scala script executes a streaming query from the Azure Cosmos DB change feed to compute an interval count of tags from the spark-shell.

## Next steps

If you haven't already, download the Spark to Azure Cosmos DB connector from the [azure-cosmosdb-spark](#) GitHub repository and explore the additional resources in the repo:

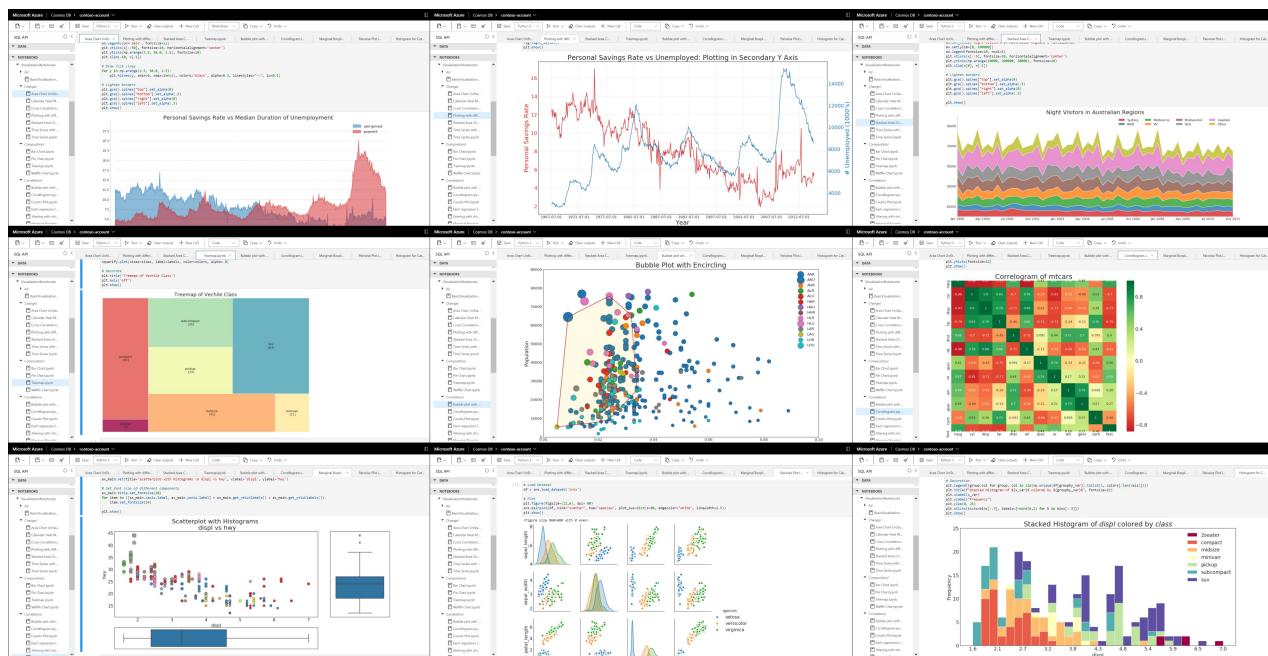
- [Lambda architecture](#)
- [Distributed aggregations examples](#)
- [Sample scripts and notebooks](#)
- [Structured streaming demos](#)
- [Change feed demos](#)
- [Stream processing changes using Azure Cosmos DB Change Feed and Apache Spark](#)

You might also want to review the [Apache Spark SQL, DataFrames, and Datasets Guide](#) and the [Apache Spark on Azure HDInsight](#) article.

# Built-in Jupyter notebooks support in Azure Cosmos DB (preview)

1/24/2020 • 3 minutes to read • [Edit Online](#)

Jupyter notebook is an open-source web application that allows you to create and share documents containing live code, equations, visualizations, and narrative text. Azure Cosmos DB supports built-in Jupyter notebooks for all APIs such as Cassandra, MongoDB, SQL, Gremlin, and Table. The built-in notebook support for all Azure Cosmos DB APIs and data models allows you to interactively run queries. The Jupyter notebooks run within the Azure Cosmos accounts and they enable developers to perform data exploration, data cleaning, data transformations, numerical simulations, statistical modeling, data visualization, and machine learning.



The Jupyter notebooks supports magic functions that extend the capabilities of the kernel by supporting additional commands. Cosmos magic is a command that extends the capabilities of the Python kernel in Jupyter notebook so you can run Azure Cosmos SQL API queries in addition to Apache Spark. You can easily combine Python and SQL API queries to query and visualize data by using rich visualization libraries integrated with render commands. Azure portal natively integrates Jupyter notebook experience into Azure Cosmos accounts as shown in the following image:

```

Dashboard > earthquakes - Data Explorer
earthquakes - Data Explorer
Azure Cosmos DB account
SQL API < 
DATA
earthquakes
earthquakes
NOTEBOOKS
cosmos.ipynb
CVE_ML.ipynb
CVE-test.ipynb
CVE.ipynb
geodemo.ipynb
ipygmap.ipynb
Untitled.ipynb
Untitled2.ipynb
Utils.ipynb
geodemo.ipynb x
geodemo.ipynb
Jupyter Notebook with geospatial queries running inside Cosmos DB
We are retrieving earthquakes within a certain radius of a given point, using Cosmos DB SQL API with geospatial query.
[278]: %%sql
SELECT * FROM e WHERE ST_DISTANCE(e.geometry, {'type': 'Point', 'coordinates':[-122.3, 47.6]}) < 20000
[278]:
[279]:

```

## Benefits of Jupyter notebooks

Jupyter notebooks were originally developed for data science applications written in Python, R. However, they can be used in various ways for different kinds of projects such as:

- **Data visualizations:** Jupyter notebooks allow you to visualize data in the form of a shared notebook that renders some data set as a graphic. Jupyter notebook lets you author visualizations, share them, and allow interactive changes to the shared code and data set.
- **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:** Jupyter notebook code is dynamic; it can be edited and re-run incrementally in real time. Notebooks can also embed user controls (e.g., sliders or text input fields) that can be 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 in Azure Cosmos DB, with real-time output all along the way, you could embed it in a Jupyter Notebook. The code will remain fully functional. You can add interactivity along with the documentation at the same time.
- **Cosmos magic commands:** In Jupyter notebooks, you can use custom magic commands for Azure Cosmos DB to make interactive computing easier. For example, the %%sql magic that allows one to query a Cosmos container using SQL API directly in a notebook.
- **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:

- **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 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:** Graphics and charts can be generated from the code, using modules like Matplotlib, Plotly, or Bokeh. Similar to the output, these visualizations appear inline next to the code that generates them.
- **Multimedia:** Because Jupyter notebook is built on the web technology, it can display all the types of multimedia supported in 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:** Data from Azure Cosmos containers and results of the queries can be imported into a Jupyter notebook programmatically. For example, by including code in the notebook to query the data using any of the Cosmos DB APIs or natively built-in Apache Spark.

## 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 notebook features and commands](#)

# Stored procedures, triggers, and user-defined functions

12/13/2019 • 8 minutes to read • [Edit Online](#)

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 guarantees that the 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 you 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

If a stored procedure is associated with an Azure Cosmos container, then the stored procedure is executed in the transaction scope of a logical partition key. Each stored procedure execution must include a logical partition key value that corresponds to 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.

## 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

1/30/2020 • 3 minutes to read • [Edit Online](#)

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
<code>chain() ... .value([callback] [, options])</code>	Starts a chained call that must be terminated with value().
<code>filter(predicateFunction [, options] [, callback])</code>	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.
<code>flatten([isShallow] [, options] [, callback])</code>	Combines and flattens arrays from each input item into a single array. This function behaves similar to SelectMany in LINQ.
<code>map(transformationFunction [, options] [, callback])</code>	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.
<code>pluck([propertyName] [, options] [, callback])</code>	This function is a shortcut for a map that extracts the value of a single property from each input item.
<code>sortBy([predicate] [, options] [, callback])</code>	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.
<code>sortByDescending([predicate] [, options] [, callback])</code>	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.
<code>unwind(collectionSelector, [resultSelector], [options], [callback])</code>	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
<code>SELECT * FROM docs</code>	<code>__.map(function(doc) {     return doc; });</code>	Results in all documents (paginated with continuation token) as is.
<code>SELECT docs.id, docs.message AS msg, docs.actions FROM docs</code>	<code>__.map(function(doc) {     return {         id: doc.id,         msg: doc.message,         actions: doc.actions     }; });</code>	Projects the id, message (aliased to msg), and action from all documents.
<code>SELECT * FROM docs WHERE docs.id="X998_Y998"</code>	<code>__.filter(function(doc) {     return doc.id === "X998_Y998"; });</code>	Queries for documents with the predicate: id = "X998_Y998".
<code>SELECT * FROM docs WHERE ARRAY_CONTAINS(docs.Tags, 123)</code>	<code>__.filter(function(x) {     return x.Tags &amp;&amp; x.Tags.indexOf(123) &gt; -1; });</code>	Queries for documents that have a Tags property and Tags is an array containing the value 123.
<code>SELECT docs.id, docs.message AS msg FROM docs WHERE docs.id="X998_Y998"</code>	<code>__.chain() .filter(function(doc) {     return doc.id === "X998_Y998"; }) .map(function(doc) {     return {         id: doc.id,         msg: doc.message     }; }) .value();</code>	Queries for documents with a predicate, id = "X998_Y998", and then projects the id and message (aliased to msg).

SQL	JAVASCRIPT QUERY API	DESCRIPTION
<pre>SELECT VALUE tag FROM docs JOIN tag IN docs.Tags ORDER BY docs._ts</pre>	<pre>__.chain()   .filter(function(doc) {     return doc.Tags &amp;&amp; Array.isArray(doc.Tags);   })   .sortBy(function(doc) {     return doc._ts;   })   .pluck("Tags")   .flatten()   .value()</pre>	<p>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.</p>

## 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 \(ECMA 2015\)](#)

# Plan and manage costs for Azure Cosmos DB

2/11/2020 • 4 minutes to read • [Edit Online](#)

This article describes how you can plan and manage costs for Azure Cosmos DB. First, you use the Azure Cosmos DB capacity calculator to help plan for costs before you add any resources. Next, as you add the Azure resources, you can review the estimated costs. After you've started using Azure Cosmos DB resources, use the cost management features to set budgets and monitor costs. You can also review the forecasted costs and identify spending trends to identify areas where you might want to act.

Understand that the costs for Azure Cosmos DB are only a portion of the monthly costs in your Azure bill. If you are using other Azure services, you're billed for all the Azure services and resources used in your Azure subscription, including the third-party services. This article explains how to plan for and manage costs for Azure Cosmos DB. After you're familiar with managing costs for Azure Cosmos DB, you can apply similar methods to manage costs for all the Azure services used in your subscription.

## Prerequisites

Cost analysis supports different kinds of Azure account types. 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 your Azure account. For information about assigning access to Azure Cost Management data, see [Assign access to data](#).

## Review estimated costs with capacity calculator

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:

The calculator below offers you a quick estimate of the workload cost on **Cosmos DB**. For a more precise estimate and ability to tweak more parameters, please [sign in](#) with an account you use for Azure.

**Cosmos Account Settings**

The simplified Azure Cosmos calculator assumes commonly used settings for indexing policy, consistency, and other parameters. For a more accurate estimate, please [sign in](#) to provide your workload details.

Number of regions  1

Multi-region writes  Disabled  Enabled

**Workload per region**

For a more accurate cost estimate based on your own data, please [sign in](#) and upload your data items.

Total data stored  GB 10

Item size  1 KB

Reads/sec per region

Writes/sec per region

**Calculate**

**Cost Estimate**

**Storage**

Cost per GB/month	<price> USD
Total Data stored per region	x 10 GB
EST. STORAGE COST PER MONTH	<price> USD

**Workload**

Cost per 100 RU/s per hour	<price> USD
EST. THROUGHPUT REQUIRED <a href="#">Show Details</a>	x 595 RU/s
EST. WORKLOAD COST/MONTH	<price> USD

Number of regions x 1

EST. TOTAL COST/MONTH <price> USD

**Sign in to save estimate**

**SAVE UP TO 65% WITH RESERVED CAPACITY**  
[See here for more details](#)

**YOU WILL SAVE UP TO 70% TCO WITH COSMOS**  
[Learn more about Cosmos TCO](#)

## Review estimated costs from the Azure portal

As you create 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 **Data Explorer**.
3. Create a new container such as a graph container.
4. 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:

The screenshot shows the Azure Cosmos DB Data Explorer interface for a database account named 'cdbgraph1'. The left sidebar includes links for Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Quick start, Notifications, and Data Explorer (which is selected and highlighted with a red box). The main area displays the 'GREMLIN API' section with a list of existing graphs: graphdb, GraphDemoDatabase, mygraphdb, and sample-database. A 'New Graph' button is highlighted with a red box. On the right, an 'Add Graph' dialog is open, prompting for a Database id (set to 'Create new' with value 'MyDBGraph'), Provision database throughput (checked), Throughput (set to 'Manual' at 400 RU/s), and estimated spend information. It also asks for a Graph id ('MyGraph1') and Partition key ('/userid'). A note indicates that partition keys must be less than 100 bytes.

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](#).

## Use budgets and cost alerts

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. However, they may have limited functionality to manage individual Azure service costs like the cost of Azure Cosmos DB because they are designed to track costs at a higher level.

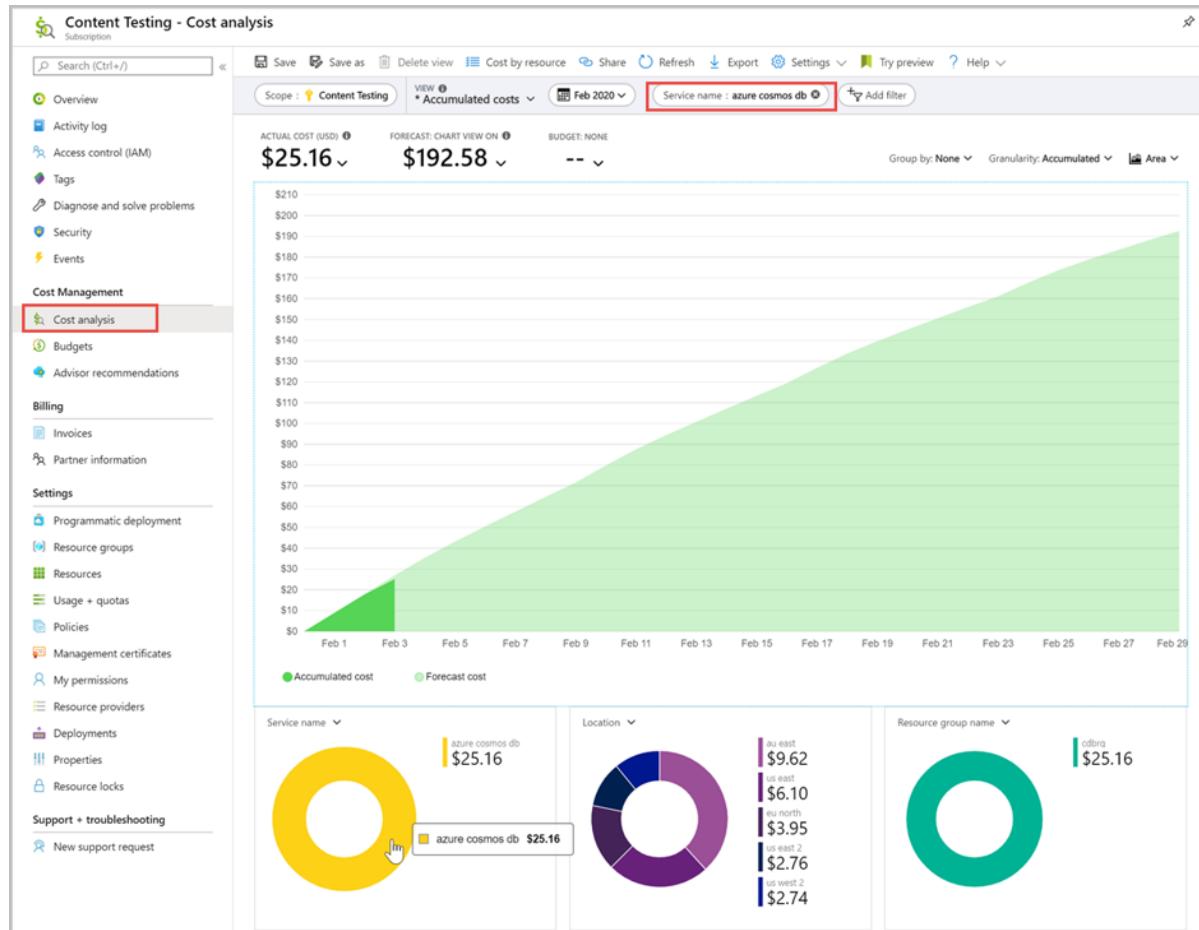
## Monitor costs

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 **Cost Management + Billing** window, select **Cost management** from the menu and then select **Cost analysis**. You can then change the scope for a specific subscription from the **Scope** dropdown.
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.

## 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](#)

# Pricing model in Azure Cosmos DB

1/14/2020 • 5 minutes to read • [Edit Online](#)

The pricing model of Azure Cosmos DB simplifies the cost management and planning. With Azure Cosmos DB, you pay for the throughput provisioned and the storage that you consume.

- **Provisioned Throughput:** Provisioned throughput (also called reserved throughput) guarantees high performance at any scale. You specify the throughput (RU/s) that you need, and Azure Cosmos DB dedicates the resources required to guarantee the configured throughput. 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 provisioned throughput even if you don't run any workloads.

- **Consumed Storage:** You are billed a flat rate for the total amount of storage (GBs) consumed for data and the indexes for a given hour.

Provisioned throughput, specified as [Request Units](#) per second (RU/s), allows you to read from or write data into containers or databases. 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. Azure Cosmos DB pricing is elastic and is proportional to the throughput that you configure on a database or a container. The minimum throughput and storage values and the scale increments provide a full range of price vs. elasticity spectrum to all segments of customers, from small scale to large-scale containers. Each database or a container is billed on an hourly basis for the throughput provisioned in the units of 100 RU/s, with a minimum of 400 RU/s, and storage consumed in GBs. Unlike provisioned throughput, storage is billed on a consumption basis. That is, you don't have to reserve any storage in advance. You are billed only for the storage you consume.

For more information, see the [Azure Cosmos DB pricing page](#) and [Understanding your Azure Cosmos DB bill](#).

The pricing model in Azure Cosmos DB is consistent across all APIs. To learn more, see [How Azure Cosmos DB pricing model is cost-effective for customers](#). There is a minimum throughput required on a database or a container to ensure the SLAs and you can increase or decrease the provisioned throughput by \$6 for each 100 RU/s.

Currently the minimum price for both database and the container-based throughput is \$24/month (see the [Azure Cosmos DB pricing page](#) for latest information). If your workload uses multiple containers, it can be optimized for cost by using database level throughput because database level throughput allows you to have any number of containers in a database sharing the throughput among the containers. The following table summarizes provisioned throughput and the costs for different entities:

ENTITY	MINIMUM THROUGHPUT & COST	SCALE INCREMENTS & COST	PROVISIONING SCOPE
Database	400 RU/s (\$24/month)	100 RU/s (\$6/month)	Throughput is reserved for the database and is shared by containers within the database

ENTITY	MINIMUM THROUGHPUT & COST	SCALE INCREMENTS & COST	PROVISIONING SCOPE
Container	400 RU/s (\$24/month)	100 RU/s (\$6/month)	Throughput is reserved for a specific container

As shown in the previous table, the minimum throughput in Azure Cosmos DB starts at a price of \$24/month. If you start with the minimum throughput and scale up over time to support your production workloads, your costs will rise smoothly, in the increments of \$6/month. The pricing model in Azure Cosmos DB is elastic and there is a smooth increase or decrease in the price as you scale up or down.

## Try Azure Cosmos DB for free

Azure Cosmos DB offers several options for developers to try it for free. These options include:

- **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 RUs 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 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

10/21/2019 • 7 minutes to read • [Edit Online](#)

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](#).

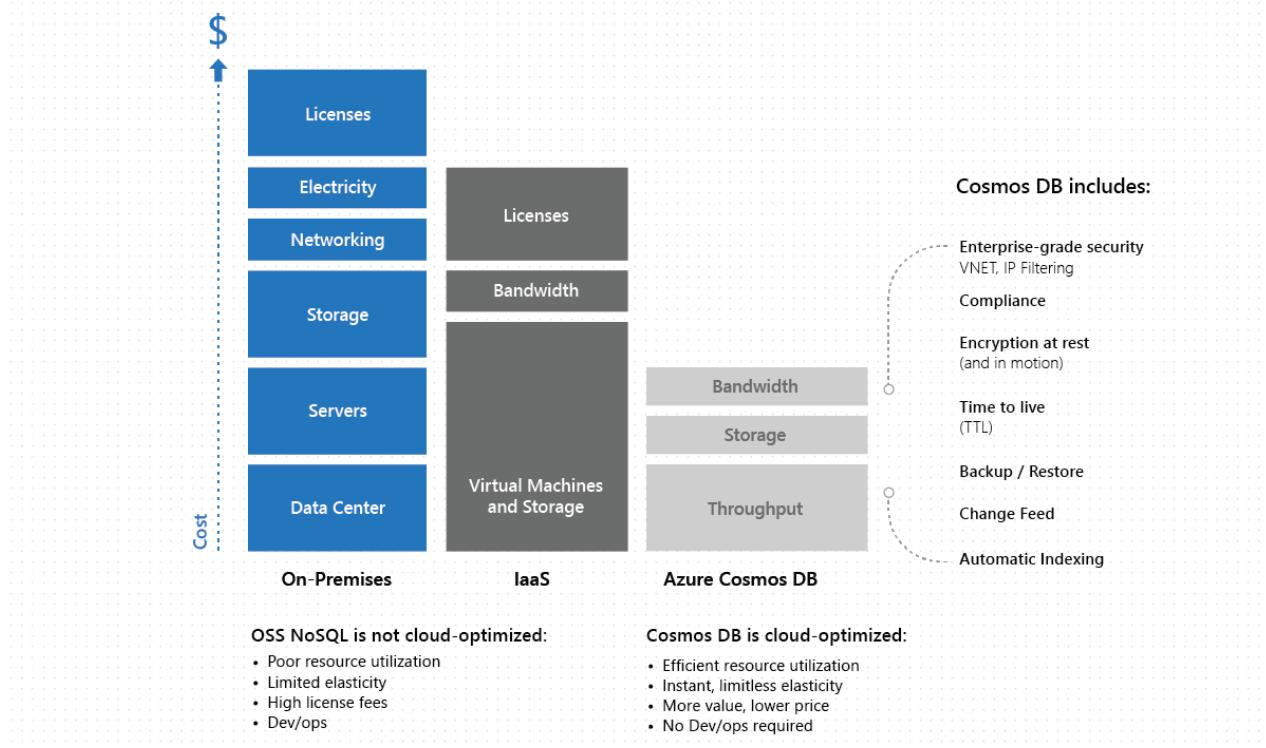
The OSS NoSQL database solutions, such as Apache Cassandra, MongoDB, HBase, engines were designed for on-premises. When offered as a managed service they are equivalent to a Resource Manager template with a tenant database for managing the provisioned clusters and monitoring support. OSS NoSQL architectures 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-master, 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

1/16/2020 • 14 minutes to read • [Edit Online](#)

As a fully managed cloud-native database service, Azure Cosmos DB simplifies billing by charging only for provisioned throughput 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.

With Azure Cosmos DB, you are billed hourly based on the provisioned throughput and the consumed storage. For the provisioned throughput, the unit for billing is 100 RU/sec per hour, charged at \$0.008 per hour, assuming standard public pricing, see the [Pricing page](#). For the consumed storage, you are billed \$0.25 per 1 GB of storage per month, see the [Pricing page](#).

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 period over a month.

## NOTE

Billing is for any portion of a wall-clock hour, not a 60 minute duration.

## Billing examples

### Billing example - 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  $\$0.08 * 24 \text{ hours} * 30 \text{ days} = \$57.60$  for the month.
- The total monthly bill will show 7,200 units (of 100 RUs), which will cost \$57.60.

### Billing example - 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 ( $2,500 \text{ RU/sec} / 100 \text{ RU/sec/unit} * 24 \text{ hours}$ ). The cost will be \$4.80 ( $600 \text{ units} * \$0.008/\text{unit}$ ).
- Total bill for the month will be \$4.80.

### 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.

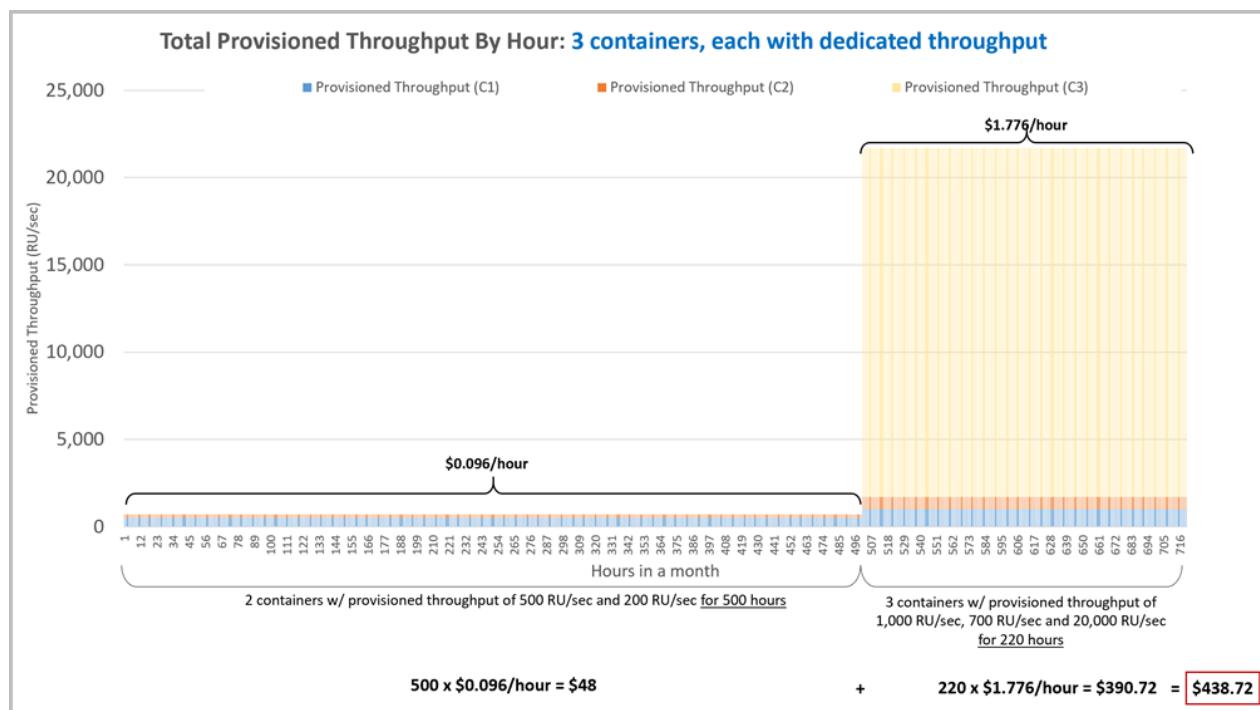
## Billing rate when throughput on a container or database scales up/down

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  $1,200/100 * \$0.008 = \$0.096/\text{hour}$ .
- 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 ( $1,000 \text{ RU/sec} + 1,200 \text{ RU/sec} + 20,000 \text{ RU/sec}$ ).
- Your bill would then change to:  $\$0.008 \times 222 = \$1.776/\text{hour}$ .
- 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:  $500 \times \$0.096/\text{hour} + 220 \times \$1.776/\text{hour} = \$438.72/\text{month}$ .

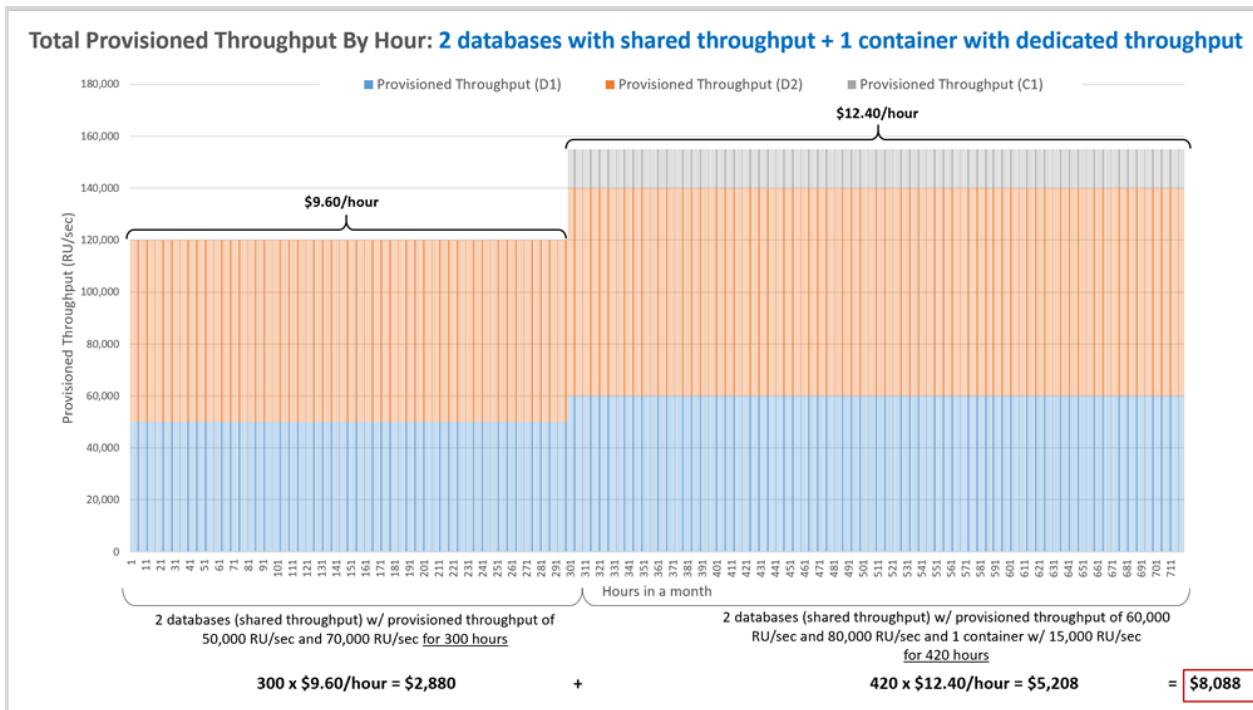


## Billing example: containers with shared throughput mode

- 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  $1200 \times \$0.008 = \$9.60/\text{hour}$ .
- 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:  $1,550 * \$0.008 = \$12.40/\text{hour}$ .
- 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:  $300 \times \$9.60/\text{hour} + 420 \times \$12.40/\text{hour} = \$2,880 + \$5,208 = \$8,088/\text{month}$ .



## Billing examples with geo-replication and multi-master

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, (a) configured with a single write region is equal to  $T \times N$  RU/sec and (b) configured with all regions capable of processing writes is equal to  $T \times (N+1)$  RU/sec, respectively. Provisioned throughput (single write region) costs \$0.008/hour per 100 RU/sec and provisioned throughput with multiple writable regions (multi-master 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	\$0.008 per 100 RU/sec per hour	\$576

ITEM	USAGE (MONTH)	RATE	MONTHLY COST
Throughput bill for 3 additional regions - East US, North Europe, and East Asia	$3 * 10K \text{ RU/sec} * 24 * 30$	\$0.008 per 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	$3 * 250 \text{ GB}$	\$0.25/GB	\$187.50
<b>Total</b>			<b>\$2,554</b>

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)	$10K \text{ RU/sec} * 24 * 30$	\$0.016 per 100 RU/sec per hour	\$1,152
Throughput bill for 3 additional regions - East US, North Europe, and East Asia (all regions are writable)	$(3 + 1) * 10K \text{ RU/sec} * 24 * 30$	\$0.016 per 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	$3 * 250 \text{ GB}$	\$0.25/GB	\$187.50
<b>Total</b>			<b>\$6,010</b>

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-master, 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 (multi-master 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]:

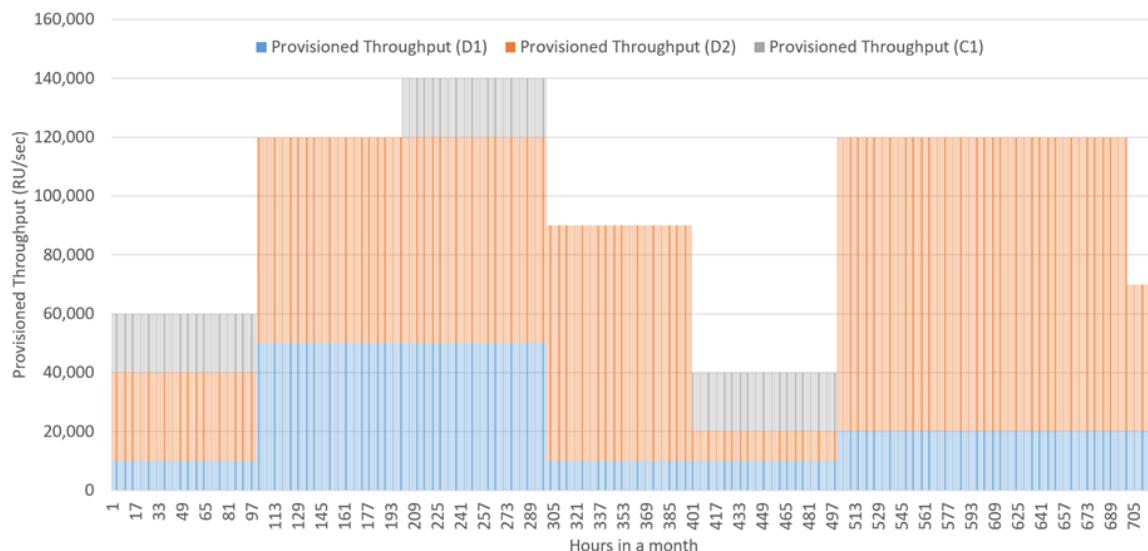
- We scaled up database (D1) to 20 K RU/sec
- We scaled up database (D2) to 100 K RU/sec
- We deleted container (C1) again

[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:

**Total Provisioned Throughput By Hour:**  
**2 databases with shared throughput + 1 container with dedicated throughput – elastically scaling**



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

HOURS	RU/S	ITEM	USAGE (HOURLY)	COST			
[201-300]	D1:50K D2:70K C1:20K	Throughput bill for container in West US (all regions are writable)	<table border="1"> <tr><td>D1: 50 K RU/sec/100 * \$0.016 * 100 hours = \$800</td></tr> <tr><td>D2: 70 K RU/sec/100 * \$0.016 * 100 hours = \$1,120</td></tr> <tr><td>C1: 20 K RU/sec/100 *\$0.016 * 100 hours = \$320</td></tr> </table>	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
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							
		Throughput bill for 2 additional regions: East US, North Europe (all regions are writable)	<table border="1"> <tr><td>(2 + 1) * (140 K RU/sec /100 * \$0.016-) * 100 hours = \$6,720</td></tr> </table>	(2 + 1) * (140 K RU/sec /100 * \$0.016-) * 100 hours = \$6,720	\$6,720		
(2 + 1) * (140 K RU/sec /100 * \$0.016-) * 100 hours = \$6,720							
[301-400]	D1:10K D2:80K C1: --	Throughput bill for container in West US (all regions are writable)	<table border="1"> <tr><td>D1: 10K RU/sec/100 * \$0.016 * 100 hours = \$160</td></tr> <tr><td>D2: 80 K RU/sec/100 * \$0.016 * 100 hours = \$1,280</td></tr> </table>	D1: 10K RU/sec/100 * \$0.016 * 100 hours = \$160	D2: 80 K RU/sec/100 * \$0.016 * 100 hours = \$1,280	\$1,440	
D1: 10K RU/sec/100 * \$0.016 * 100 hours = \$160							
D2: 80 K RU/sec/100 * \$0.016 * 100 hours = \$1,280							
		Throughput bill for 2 additional regions: East US, North Europe (all regions are writable)	<table border="1"> <tr><td>(1 + 1) * (90 K RU/sec /100 * \$0.016) * 100 hours = \$2,880</td></tr> </table>	(1 + 1) * (90 K RU/sec /100 * \$0.016) * 100 hours = \$2,880	\$2,880		
(1 + 1) * (90 K RU/sec /100 * \$0.016) * 100 hours = \$2,880							
[401-500]	D1:10K D2:10K C1:20K	Throughput bill for container in West US (all regions are writable)	<table border="1"> <tr><td>D1: 10K RU/sec/100 * \$0.016 * 100 hours = \$160</td></tr> <tr><td>D2: 10K RU/sec/100 * \$0.016 * 100 hours = \$160</td></tr> <tr><td>C1: 20 K RU/sec/100 *\$0.016 * 100 hours = \$320</td></tr> </table>	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
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							
		Throughput bill for 2 additional regions: East US, North Europe (all regions are writable)	<table border="1"> <tr><td>(1 + 1) * (40 K RU/sec /100 * \$0.016) * 100 hours = \$1,280</td></tr> </table>	(1 + 1) * (40 K RU/sec /100 * \$0.016) * 100 hours = \$1,280	\$1,280		
(1 + 1) * (40 K RU/sec /100 * \$0.016) * 100 hours = \$1,280							
[501-700]	D1:20K D2:100K C1: --	Throughput bill for container in West US (all regions are writable)	<table border="1"> <tr><td>D1: 20 K RU/sec/100 * \$0.016 * 200 hours = \$640</td></tr> <tr><td>D2: 100 K RU/sec/100 * \$0.016 * 200 hours = \$3,200</td></tr> </table>	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	
D1: 20 K RU/sec/100 * \$0.016 * 200 hours = \$640							
D2: 100 K RU/sec/100 * \$0.016 * 200 hours = \$3,200							

HOURS	RU/S	ITEM	USAGE (HOURLY)	COST
		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) * 200 hours = \$448	\$224
<b>Total Monthly Cost</b>				<b>\$38,688</b>

## 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 \$0.008 per 100 RU/sec in that region
- in Japan East: for 50-K RU/sec at the rate of \$0.009 per 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 \$6.40 per hour. See reserved capacity pricing on the [Pricing page](#).

- Cost of throughput (pay-as-you-go):  $100,000 \text{ RU/sec}/100 * \$0.008/\text{hour} * 8760 \text{ hours in a year} = \$70,080$
- Cost of throughput (with reserved capacity) \$70,080 discounted at 20% = \$56,064

What you've effectively purchased is a credit of \$8 per hour, for 100 K RU/sec using the list price in East US, at the price of \$6.40 per 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 \$8.00 worth of provisioned throughput per hour, and will be billed the average of \$0.50 per 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

REGION	HOURLY PRICE PER 100 RU/S	UNITS (RU/S)	BILLED AMOUNT (HOURLY)	BILLED AMOUNT (MONTHLY)
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

2/7/2020 • 14 minutes to read • [Edit Online](#)

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 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

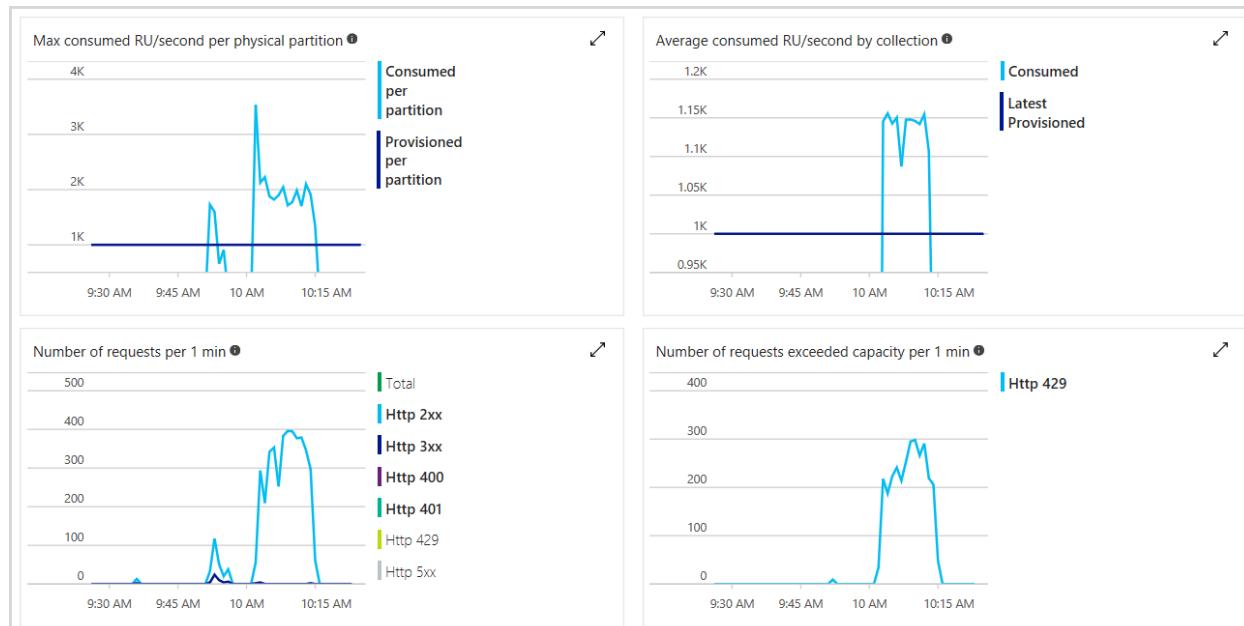
save on storage space, index space, and provisioned throughput and result in lower costs.

## 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 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 portal to 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 query cost in Azure Cosmos DB

12/13/2019 • 5 minutes to read • [Edit Online](#)

Azure Cosmos DB offers a rich set of database operations including relational and hierarchical queries 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. This article describes how to evaluate request unit charges for a query and optimize the query in terms of performance and cost.

Queries in Azure Cosmos DB are typically ordered from fastest/most efficient to slower/less efficient in terms of throughput as follows:

- GET operation on a single partition key and item 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.

Queries that read data from one or more partitions incur higher latency and consume higher number of request units. Since each partition has automatic indexing for all properties, the query can be served efficiently from the index. You can make queries that use multiple partitions faster by using the parallelism options. To learn more about partitioning and partition keys, see [Partitioning in Azure Cosmos DB](#).

## Evaluate request unit charge for a query

Once you have stored some data in your Azure Cosmos containers, you can use the Data Explorer in the Azure portal to construct and run your queries. You can also get the cost of the queries by using the data explorer. This method will give you a sense of the actual charges involved with typical queries and operations that your system supports.

You can also get the cost of queries programmatically by using the SDKs. To measure the overhead of any operation such as create, update, or delete inspect the `x-ms-request-charge` header when using REST API. If you are using the .NET or the Java SDK, the `RequestCharge` property is the equivalent property to get the request charge and this property is present within the `ResourceResponse` or `FeedResponse`.

```
// Measure the performance (request units) of writes
ResourceResponse<Document> response = await client.CreateDocumentAsync(collectionSelfLink, myDocument);

Console.WriteLine("Insert of an item 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);
}
```

# Factors influencing request unit charge for a query

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

The performance and the throughput consumed by queries, user-defined functions (UDFs) mostly depends 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.

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.

## Next steps

Next you can proceed to learn more about cost optimization in Azure Cosmos DB with the following articles:

- Learn more about [How Azure Cosmos pricing works](#)
- 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 multi-region Azure Cosmos accounts](#)
- Learn more about [Azure Cosmos DB reserved capacity](#)

# Optimize storage cost in Azure Cosmos DB

10/21/2019 • 4 minutes to read • [Edit Online](#)

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 reads and writes cost in Azure Cosmos DB

10/21/2019 • 3 minutes to read • [Edit Online](#)

This article describes how the cost required to read and write data from Azure Cosmos DB is calculated. Read operations include get operations on items and write operations include insert, replace, delete, and upsert of items.

## Cost of reads and writes

Azure Cosmos DB guarantees 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.

You don't have to think about differentiating request units between reads and writes. The unified currency model of request units creates efficiencies to interchangeably use the same throughput capacity for both reads and writes. The following table shows the cost of reads and writes in terms of RU/s for items that are 1 KB and 100 KB in size.

ITEM SIZE	COST OF ONE READ	COST OF ONE WRITE
1 KB	1 RU	5 RUs
100 KB	10 RUs	50 RUs

Reading an item that is 1 KB in size costs one RU. Writing an item that is 1-KB costs five RUs. The read and write costs are applicable when using the default session [consistency level](#). The considerations around RUs include: item size, property count, data consistency, indexed properties, indexing, and query patterns.

## Normalized cost for 1 million reads and writes

Provisioning 1,000 RU/s translates to 3.6 million RU/hour and will cost \$0.08 for the hour (in the US and Europe). For a 1-KB item, you can perform 3.6 million reads or 0.72 million writes, (this value is calculated as:  $3.6 \text{ million RU} / 5$ ) per hour with this provisioned throughput. Normalized to a million reads and writes, the cost would be \$0.022 for 1 million reads (this value is calculated as:  $\$0.08/3.6 \text{ million}$ ) and \$0.111 for 1 million writes (this value is calculated as:  $\$0.08/0.72 \text{ million}$ ).

## Number of regions and the request units cost

The cost of writes is constant irrespective of the number of regions associated with the Azure Cosmos account. In other words, a 1 KB write will cost five RUs independent of the number of regions that are associated with the account. There's a non-trivial amount of resources spent in replicating, accepting, and processing the replication traffic on every region. For details about multi-region cost optimization, see [Optimizing the cost of multi-region Cosmos accounts](#) article.

## Optimize the cost of writes and reads

When you perform write operations, you should provision enough capacity to support the number of writes needed per second. You can increase the provisioned throughput by using SDK, portal, CLI before performing the writes and then reduce the throughput after the writes are completed. Your throughput for the write period is the minimum throughput needed for the given data, plus the throughput required for insert workload assuming no other workloads are running.

If you are running other workloads concurrently, for example, query/read/update/delete, you should add the additional request units required for those operations too. If the write operations are rate-limited, you can customize the retry/backoff policy by using Azure Cosmos DB SDKs. For instance, you can increase the load until a small rate of requests gets rate-limited. If rate-limit occurs, the client application should back off on rate-limiting requests for the specified retry interval. Before retrying writes, you should have a minimal amount of time gap between retries. Retry policy support is included in SQL.NET, Java, Node.js, and Python SDKs and all the supported versions of the .NET Core SDKs.

You can also bulk insert data into Azure Cosmos DB or copy data from any supported source data store to Azure Cosmos DB by using [Azure Data Factory](#). Azure Data Factory natively integrates with the Azure Cosmos DB Bulk API to provide the best performance, when you write data.

## 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 queries](#)
- Learn more about [Optimizing the cost of multi-region Azure Cosmos accounts](#)

# Optimize multi-region cost in Azure Cosmos DB

10/21/2019 • 3 minutes to read • [Edit Online](#)

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:

1.  $T \times N$  RU/s if your Azure Cosmos account is configured with a single write region.
2.  $T \times (N+1)$  RU/s if your Azure Cosmos account is configured with all regions capable of processing writes.

Provisioned throughput with single write region costs \$0.008/hour per 100 RU/s and provisioned throughput with multiple writable regions costs \$0.016/hour per 100 RU/s. To learn more, see [Azure Cosmos DB Pricing page](#).

## Costs for multiple write regions

In a multi-master 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 should support conflict resolution. The amount of workload for writers has increased. 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 you have a container in West US provisioned with throughput 10K RU/s and stores 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 four regions from your globally distributed app. Your total monthly bill(assuming 31 days) in a month is as follows:

ITEM	USAGE (MONTHLY)	RATE	MONTHLY COST
Throughput bill for container in West US (multiple write regions)	10K RU/s * 24 * 31	\$0.016per 100 RU/s per hour	\$1,190.40
Throughput bill for 3 additional regions - East US, North Europe, and East Asia (multiple write regions)	(3 + 1) * 10K RU/s * 24 * 31	\$0.016per 100 RU/s per hour	\$4,761.60
Storage bill for container in West US	1 TB (or 1,024 GB)	\$0.25/GB	\$256
Storage bill for 3 additional regions - East US, North Europe, and East Asia	3 * 1 TB (or 3,072 GB)	\$0.25/GB	\$768
<b>Total</b>			<b>\$6,976</b>

## Improve throughput utilization on a per region-basis

If you have inefficient utilization, for example, one or more under-utilized or over-utilized regions, you can take the following steps to improve throughput utilization:

1. Make sure you optimize provisioned throughput (RUs) in the write region first, and then make the maximum use of the RUs in read regions by using change feed from the read-region etc.
2. Multiple write regions reads and writes can be scaled across all regions associated with Azure Cosmos account.
3. Monitor the activity in your regions and you could add and remove regions on demand to scale your read and write throughput.

## 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

10/21/2019 • 2 minutes to read • [Edit Online](#)

This article describes the different options to use Azure Cosmos DB for development and testing for free of cost.

## 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.

## Try Azure Cosmos DB for free

[Try Azure Cosmos DB for free](#) is a free of charge experience that allows you to create database and collections, and experiment with Azure Cosmos DB in the cloud. You don't have to sign-up for Azure or pay any cost. 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 by using the Quickstarts or tutorials. You can create a demo or perform unit testing without incurring any costs. By using try Azure Cosmos DB for free accounts, you can evaluate Azure Cosmos DB's premium capabilities for free, including turnkey global distribution, SLAs, and consistency models. You can create a database with a maximum of 25 Azure Cosmos containers and 10,000 RU/s of throughput. You can run the sample application without subscribing to an Azure account or using your credit card. With Try Azure Cosmos DB for free, you can create a multi-region Azure Cosmos account and run an app on it in just a few minutes. 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.

## Next steps

You can get started with using the emulator or the free Azure Cosmos DB accounts 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](#)
- Learn more about [Optimizing the cost of multi-region Azure Cosmos accounts](#)

# Optimize cost with reserved capacity in Azure Cosmos DB

2/20/2020 • 6 minutes to read • [Edit Online](#)

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-master 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:

The screenshot shows a dialog box titled "Select the product you want to purchase". At the top, there is a note about Cosmos DB Reserved Capacity savings and a "Learn More" link. Below this are dropdown menus for "Scope" (set to "Shared") and "Billing subscription". A red box highlights the "Recommended" tab, which is selected over "All Products". There are also filter options for "Filter by name...", "Term" (set to "Three Years"), "Billing frequency" (set to "Select a value"), "Add Filter", and "Reset filters". A message below the filters says "Showing recommendations based on your usage over the last 30 d... Learn more". The main table lists two recommendations:

Reserved Capacity Units	Throughput Type	Term	Billing frequency	Recommended Quant...
30,000 RU/s	100 RU/s / Hour	Three Years	Upfront	1
30,000 RU/s	100 RU/s / Hour	Three Years	Monthly	1

At the bottom left are "Select" and "Cancel" buttons. On the right, it says "Upfront price for reserved capacity units: <Total Cost>".

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:

Select the product you want to purchase

Cosmos DB Reserved Capacity offers significant savings over the normal price of Cosmos DB provisioned throughput capacity. You pay a one-time upfront fee and commit to paying for a minimum provisioned throughput level at the following hourly rates for the duration of the reserved capacity term. [Learn More](#)

Scope \* Shared Billing subscription \* Pay-As-You-Go (ae34073c-52a8-4c2a-b5d0-7e40fe...)

Recommended All Products

Filter by name... Term : Three Years Billing frequency : Monthly Throughput Type : 100 RU/s / Hour Reset filters

Showing recommendations based on your usage over the last 60 d... Learn more

Reserved Capacity Units	Throughput Type	Term	Billing frequency	Recommended Quantity
5,000 RU/s	100 RU/s / Hour	Three Years	Monthly	0
10,000 RU/s	100 RU/s / Hour	Three Years	Monthly	0
20,000 RU/s	100 RU/s / Hour	Three Years	Monthly	0
30,000 RU/s	100 RU/s / Hour	Three Years	Monthly	0
40,000 RU/s	100 RU/s / Hour	Three Years	Monthly	0
50,000 RU/s	100 RU/s / Hour	Three Years	Monthly	0
60,000 RU/s	100 RU/s / Hour	Three Years	Monthly	0
70,000 RU/s	100 RU/s / Hour	Three Years	Monthly	0
80,000 RU/s	100 RU/s / Hour	Three Years	Monthly	0
90,000 RU/s	100 RU/s / Hour	Three Years	Monthly	0
100,000 RU/s	100 RU/s / Hour	Three Years	Monthly	0
120,000 RU/s	100 RU/s / Hour	Three Years	Monthly	0
130,000 RU/s	100 RU/s / Hour	Three Years	Monthly	0
140,000 RU/s	100 RU/s / Hour	Three Years	Monthly	0
150,000 RU/s	100 RU/s / Hour	Three Years	Monthly	0
160,000 RU/s	100 RU/s / Hour	Three Years	Monthly	0
170,000 RU/s	100 RU/s / Hour	Three Years	Monthly	0
180,000 RU/s	100 RU/s / Hour	Three Years	Monthly	0
190,000 RU/s	100 RU/s / Hour	Three Years	Monthly	0
200,000 RU/s	100 RU/s / Hour	Three Years	Monthly	0
210,000 RU/s	100 RU/s / Hour	Three Years	Monthly	0
220,000 RU/s	100 RU/s / Hour	Three Years	Monthly	0
230,000 RU/s	100 RU/s / Hour	Three Years	Monthly	0
240,000 RU/s	100 RU/s / Hour	Three Years	Monthly	0
250,000 RU/s	100 RU/s / Hour	Three Years	Monthly	0
260,000 RU/s	100 RU/s / Hour	Three Years	Monthly	0
270,000 RU/s	100 RU/s / Hour	Three Years	Monthly	0
280,000 RU/s	100 RU/s / Hour	Three Years	Monthly	0
290,000 RU/s	100 RU/s / Hour	Three Years	Monthly	0
300,000 RU/s	100 RU/s / Hour	Three Years	Monthly	0
310,000 RU/s	100 RU/s / Hour	Three Years	Monthly	0
320,000 RU/s	100 RU/s / Hour	Three Years	Monthly	0
330,000 RU/s	100 RU/s / Hour	Three Years	Monthly	0
340,000 RU/s	100 RU/s / Hour	Three Years	Monthly	0
350,000 RU/s	100 RU/s / Hour	Three Years	Monthly	0
360,000 RU/s	100 RU/s / Hour	Three Years	Monthly	0
370,000 RU/s	100 RU/s / Hour	Three Years	Monthly	0
380,000 RU/s	100 RU/s / Hour	Three Years	Monthly	0
390,000 RU/s	100 RU/s / Hour	Three Years	Monthly	0
400,000 RU/s	100 RU/s / Hour	Three Years	Monthly	0
410,000 RU/s	100 RU/s / Hour	Three Years	Monthly	0
420,000 RU/s	100 RU/s / Hour	Three Years	Monthly	0
430,000 RU/s	100 RU/s / Hour	Three Years	Monthly	0
440,000 RU/s	100 RU/s / Hour	Three Years	Monthly	0
450,000 RU/s	100 RU/s / Hour	Three Years	Monthly	0
460,000 RU/s	100 RU/s / Hour	Three Years	Monthly	0
470,000 RU/s	100 RU/s / Hour	Three Years	Monthly	0
480,000 RU/s	100 RU/s / Hour	Three Years	Monthly	0
490,000 RU/s	100 RU/s / Hour	Three Years	Monthly	0
500,000 RU/s	100 RU/s / Hour	Three Years	Monthly	0
510,000 RU/s	100 RU/s / Hour	Three Years	Monthly	0
520,000 RU/s	100 RU/s / Hour	Three Years	Monthly	0
530,000 RU/s	100 RU/s / Hour	Three Years	Monthly	0
540,000 RU/s	100 RU/s / Hour	Three Years	Monthly	0
550,000 RU/s	100 RU/s / Hour	Three Years	Monthly	0
560,000 RU/s	100 RU/s / Hour	Three Years	Monthly	0
570,000 RU/s	100 RU/s / Hour	Three Years	Monthly	0
580,000 RU/s	100 RU/s / Hour	Three Years	Monthly	0
590,000 RU/s	100 RU/s / Hour	Three Years	Monthly	0
600,000 RU/s	100 RU/s / Hour	Three Years	Monthly	0
610,000 RU/s	100 RU/s / Hour	Three Years	Monthly	0
620,000 RU/s	100 RU/s / Hour	Three Years	Monthly	0
630,000 RU/s	100 RU/s / Hour	Three Years	Monthly	0
640,000 RU/s	100 RU/s / Hour	Three Years	Monthly	0
650,000 RU/s	100 RU/s / Hour	Three Years	Monthly	0
660,000 RU/s	100 RU/s / Hour	Three Years	Monthly	0
670,000 RU/s	100 RU/s / Hour	Three Years	Monthly	0
680,000 RU/s	100 RU/s / Hour	Three Years	Monthly	0
690,000 RU/s	100 RU/s / Hour	Three Years	Monthly	0
700,000 RU/s	100 RU/s / Hour	Three Years	Monthly	0
710,000 RU/s	100 RU/s / Hour	Three Years	Monthly	0
720,000 RU/s	100 RU/s / Hour	Three Years	Monthly	0
730,000 RU/s	100 RU/s / Hour	Three Years	Monthly	0
740,000 RU/s	100 RU/s / Hour	Three Years	Monthly	0
750,000 RU/s	100 RU/s / Hour	Three Years	Monthly	0
760,000 RU/s	100 RU/s / Hour	Three Years	Monthly	0
770,000 RU/s	100 RU/s / Hour	Three Years	Monthly	0
780,000 RU/s	100 RU/s / Hour	Three Years	Monthly	0
790,000 RU/s	100 RU/s / Hour	Three Years	Monthly	0
800,000 RU/s	100 RU/s / Hour	Three Years	Monthly	0
810,000 RU/s	100 RU/s / Hour	Three Years	Monthly	0
820,000 RU/s	100 RU/s / Hour	Three Years	Monthly	0
830,000 RU/s	100 RU/s / Hour	Three Years	Monthly	0
840,000 RU/s	100 RU/s / Hour	Three Years	Monthly	0
850,000 RU/s	100 RU/s / Hour	Three Years	Monthly	0
860,000 RU/s	100 RU/s / Hour	Three Years	Monthly	0
870,000 RU/s	100 RU/s / Hour	Three Years	Monthly	0
880,000 RU/s	100 RU/s / Hour	Three Years	Monthly	0
890,000 RU/s	100 RU/s / Hour	Three Years	Monthly	0
900,000 RU/s	100 RU/s / Hour	Three Years	Monthly	0
910,000 RU/s	100 RU/s / Hour	Three Years	Monthly	0
920,000 RU/s	100 RU/s / Hour	Three Years	Monthly	0
930,000 RU/s	100 RU/s / Hour	Three Years	Monthly	0
940,000 RU/s	100 RU/s / Hour	Three Years	Monthly	0
950,000 RU/s	100 RU/s / Hour	Three Years	Monthly	0
960,000 RU/s	100 RU/s / Hour	Three Years	Monthly	0
970,000 RU/s	100 RU/s / Hour	Three Years	Monthly	0
980,000 RU/s	100 RU/s / Hour	Three Years	Monthly	0
990,000 RU/s	100 RU/s / Hour	Three Years	Monthly	0
1,000,000 RU/s	100 RU/s / Hour	Three Years	Monthly	0

Select Cancel Monthly price for reserved capacity units: <Monthly Cost>

FIELD	DESCRIPTION
Scope	<p>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.</p> <p>If you select <b>Shared</b>, 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.</p> <p>If you select <b>Single subscription</b>, the reservation discount is applied to Azure Cosmos DB instances in the selected subscription.</p> <p>If you select <b>Single resource group</b>, the reservation discount is applied to Azure Cosmos DB instances in the selected subscription and the selected resource group within that subscription.</p> <p>You can change the reservation scope after you buy the reserved capacity.</p>

FIELD	DESCRIPTION
Subscription	<p>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:</p> <ul style="list-style-type: none"> <li>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.</li> <li>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.</li> </ul>
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-master 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.

- 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**
- 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.

The screenshot shows the 'Purchase reservations' pane with the following details:

Purchase reservations						
Products		Review + buy				
Azure Cosmos DB						
PRODUCT NAME	REGION	TERM LENGTH	SCOPE	UNIT PRICE	SUBTOTAL	ESTIMATED SAV...
20,000 Multi-master RU/s	Global	One Year	Single resource grou...	<Unit Price>	<Subtotal>	<Estimated Savings>
<b>Subtotal:</b> <Total Cost>						

- 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](#).

# Security in Azure Cosmos DB - overview

2/21/2020 • 7 minutes to read • [Edit Online](#)

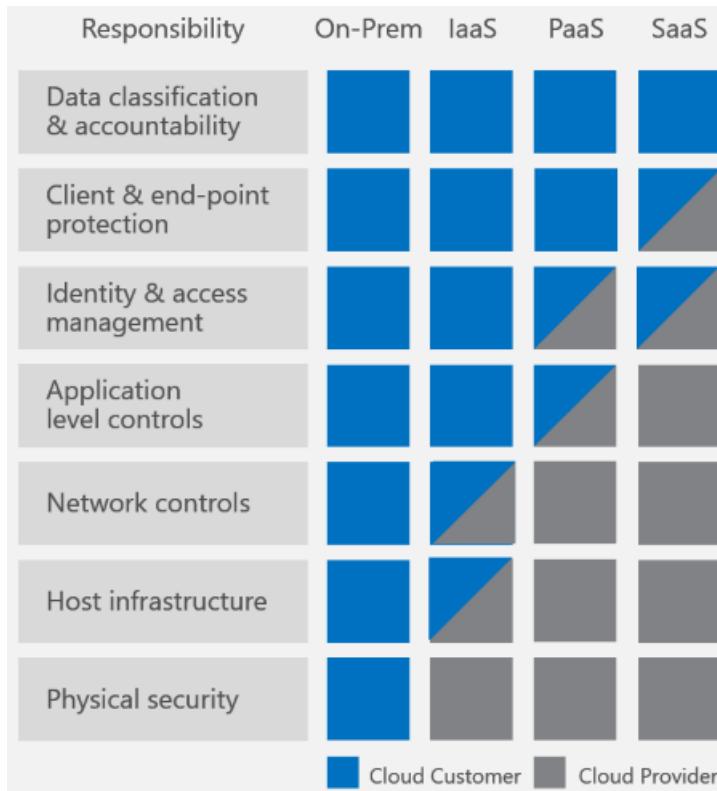
This article discusses database security best practices and key features offered by Azure Cosmos DB to help you prevent, detect, and respond to database breaches.

## 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/SSL 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	<p>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.</p> <p>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.</p> <p>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.</p> <p>Learn more in <a href="#">Azure Cosmos DB firewall support</a>.</p>

SECURITY REQUIREMENT	AZURE COSMOS DB'S SECURITY APPROACH
Authorization	<p>Azure Cosmos DB uses hash-based message authentication code (HMAC) for authorization.</p> <p>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.</p> <p>You can use either a <a href="#">master key</a>, or a <a href="#">resource token</a> allowing fine-grained access to a resource such as a document.</p> <p>Learn more in <a href="#">Securing access to Azure Cosmos DB resources</a>.</p>
Users and permissions	<p>Using the master 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.</p> <p>Learn more in <a href="#">Securing access to Azure Cosmos DB resources</a>.</p>
Active directory integration (RBAC)	<p>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 <a href="#">Active Directory integration</a> article for more information.</p>
Global replication	<p>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.</p> <p>In the context of security, global replication ensures data protection against regional failures.</p> <p>Learn more in <a href="#">Distribute data globally</a>.</p>
Regional failovers	<p>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.</p> <p>Learn more in <a href="#">Regional Failovers in Azure Cosmos DB</a>.</p>

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 <a href="#">consistency levels</a> . This replication guarantees a 99.99% <a href="#">availability SLA</a> 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	<p>Azure Cosmos databases are backed up regularly and stored in a geo redundant store.</p> <p>Learn more in <a href="#">Automatic online backup and restore with Azure Cosmos DB</a>.</p>
Restore deleted data	<p>The automated online backups can be used to recover data you may have accidentally deleted up to ~30 days after the event.</p> <p>Learn more in <a href="#">Automatic online backup and restore with Azure Cosmos DB</a></p>
Protect and isolate sensitive data	<p>All data in the regions listed in What's new? is now encrypted at rest.</p> <p>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.</p>
Monitor for attacks	<p>By using <a href="#">audit logging and activity logs</a>, 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.</p>
Respond to attacks	<p>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.</p> <p>Learn more in <a href="#">Microsoft Azure Security Response in the Cloud</a>.</p>
Geo-fencing	Azure Cosmos DB ensures data governance for sovereign regions (for example, Germany, China, US Gov).
Protected facilities	<p>Data in Azure Cosmos DB is stored on SSDs in Azure's protected data centers.</p> <p>Learn more in <a href="#">Microsoft global datacenters</a></p>
HTTPS/SSL/TLS encryption	<p>All connections to Azure Cosmos DB support HTTPS. Azure Cosmos DB also supports TLS 1.2.</p> <p>It is possible to enforce a minimum TLS version server-side. To do so, please contact <a href="mailto:azuracosmosdbtls@service.microsoft.com">azuracosmosdbtls@service.microsoft.com</a>.</p>

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 <a href="#">Azure Cosmos DB encryption at rest</a>
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 SSL 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 <a href="#">Azure Compliance site</a> as well as the latest <a href="#">Azure Compliance Document</a> 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:

Configure logs to view DocumentDB account changes

See who is accessing your keys and making changes to the account

View logs that contain timestamps, IP addresses, and usernames

Microsoft Azure Azure Cosmos DB > testacct - Activity log

testacct - Activity log

Search (Ctrl+)

Overview

Activity log

Access control (IAM)

Tags

Diagnose and solve problems

Quick start

Data Explorer

SETTINGS

Replicate data globally

Default consistency

Firewall

Columns Export Log Analytics

Select query ...

Subscription: SQL DB Content Resource group: testacct

Timestamp: Last 6 hours Resource: testacct

Event category: All categories Operation: 0 selected

Event severity: 4 selected Event initiated by: Search

Alerts fired: 0 Outage notifications: 0

Query returned 11 items. Click here to download all the items as csv.

OPERATION NAME	STATUS	TIME	TIME STAMP	SUBSCRIPTION	EVENT INITIATED BY
ListKeys	Started	27 min ago	Wed Dec 13...	SQL DB Content	jfastl@contoso.com
ListDocumentCollections	Started	36 min ago	Wed Dec 13...	SQL DB Content	jfastl@contoso.com
ListDatabases	Succeeded	36 min ago	Wed Dec 13...	SQL DB Content	jfastl@contoso.com
ListDocumentCollections	Succeeded	36 min ago	Wed Dec 13...	SQL DB Content	jfastl@contoso.com
ListDatabases	Started	36 min ago	Wed Dec 13...	SQL DB Content	jfastl@contoso.com

## Next steps

For more information about master 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

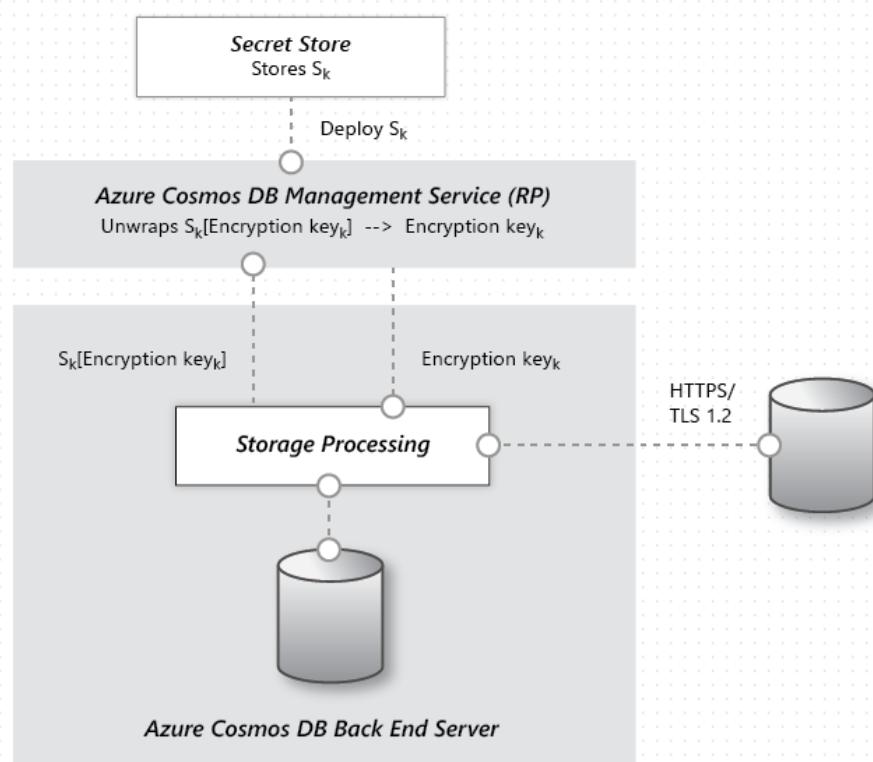
1/23/2020 • 3 minutes to read • [Edit Online](#)

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, Cosmos DB is very easy to use. 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. We provide this feature while we continue to meet our [availability and performance SLAs](#).

## 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 now this feature is available for the new cosmos accounts and this should be done at 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

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](#).

# Secure access to data in Azure Cosmos DB

1/24/2020 • 7 minutes to read • [Edit Online](#)

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
Master 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

## Master keys

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

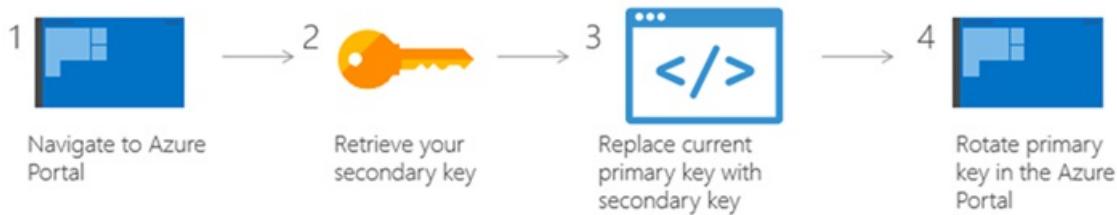
- Provide access to accounts, databases, users, and permissions.
- Cannot be used to provide granular access to containers and documents.
- Are created during the creation of an account.
- Can be regenerated at any time.

Each account consists of two Master 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 master 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 master keys can be retrieved and regenerated using the Azure portal. For instructions, see [View, copy, and regenerate access keys](#).

The process of rotating your master key is simple. Navigate to the Azure portal to retrieve your secondary key, then replace your primary key with your secondary key in your application, then rotate the primary key in the Azure portal.



### Code sample to use a master key

The following code sample illustrates how to use a Cosmos DB account endpoint and master 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);
```

## 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 master key.
- Enable clients to read, write, and delete resources in the Cosmos DB account according to the permissions they've been granted.

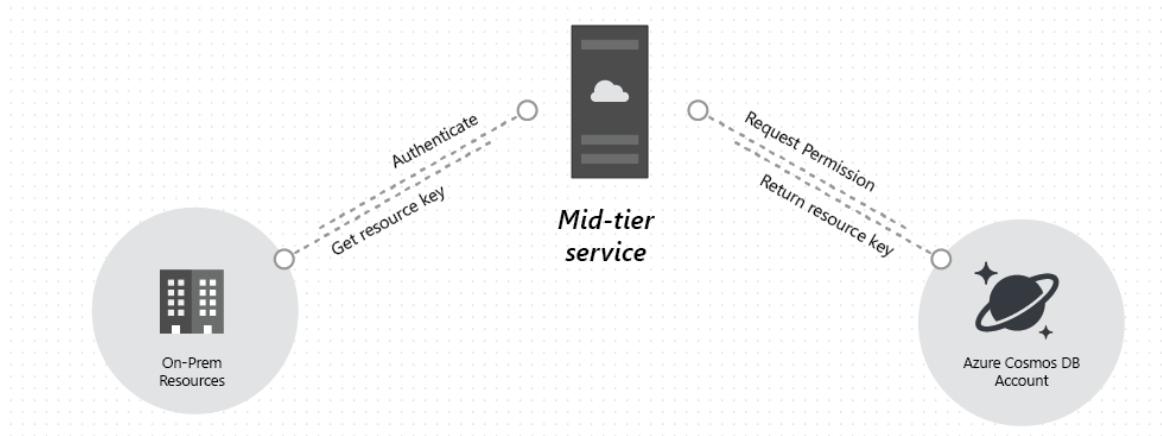
You can use a resource token (by creating Cosmos DB users and permissions) when you want to provide access to resources in your Cosmos DB account to a client that cannot be trusted with the master key.

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 master 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 master 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.

- 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.
- 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 is 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 [Nodejs 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.

### 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: "permissionUser10rds",
        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("permissionUser10rds")

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

Azure Cosmos DB enables you to search, select, modify and delete any personal data located in database or collections. Azure Cosmos DB provides APIs to find and delete personal data however, it's your responsibility to use the APIs and define logic required to erase the personal data. Each multi-model API (SQL, MongoDB, Gremlin, Cassandra, Table) provides different language SDKs that contain methods to search and delete personal data. 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](#)

# IP firewall in Azure Cosmos DB

5/23/2019 • 2 minutes to read • [Edit Online](#)

To secure 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 overview

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.

## Next steps

Next you can configure IP firewall or VNET service endpoint for your account by using the following docs:

- [How to configure IP firewall for your Azure Cosmos account](#)
- [Access Azure Cosmos DB resources from virtual networks](#)
- [How to configure virtual network service endpoint for your Azure Cosmos account](#)

# Access Azure Cosmos DB from virtual networks (VNet)

12/13/2019 • 4 minutes to read • [Edit Online](#)

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.

## Frequently asked questions

Here are some frequently asked questions about configuring access from virtual networks:

### **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 (aka 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 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 64 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**

- [How to limit Azure Cosmos account access to subnet\(s\) within virtual networks](#)
- [How to configure IP firewall for your Azure Cosmos account](#)

# Role-based access control in Azure Cosmos DB

12/13/2019 • 2 minutes to read • [Edit Online](#)

Azure Cosmos DB provides built-in role-based access control (RBAC) for common management scenarios in Azure Cosmos DB. An individual who has a profile in Azure Active Directory can assign these RBAC 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.
Cosmos DB Operator	Can provision Azure Cosmos accounts, databases, and containers but cannot access the keys that are required to access the data.

### IMPORTANT

RBAC support in Azure Cosmos DB applies to control plane operations only. Data plane operations are secured using master 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 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 (RBAC) using access control (IAM) in the Azure portal:

A screenshot of the Azure portal's Access control (IAM) blade for a specific Cosmos DB account. The left sidebar shows various management options like Overview, Activity log, and Access control (IAM). The main area is titled 'Check access' and shows a table of role assignments. The table has columns for Name, Type, Role, and Scope. There are three entries: one user with 'DocumentDB Account Contributor' role assigned, another user with 'Reader' role assigned, and a group named 'Subscription admins' with 'Owner' role inherited from the subscription.

## 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 RBAC 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](#)

## Preventing changes from Cosmos SDK

The Cosmos resource provider can be locked down to prevent any changes to resources including Cosmos account, databases, containers and throughput from any client connecting via account keys (i.e. applications connecting via Cosmos SDK). When set, changes to any resource must be from a user with the proper RBAC role and credentials. This capability is set with `disableKeyBasedMetadataWriteAccess` property value in the Cosmos resource provider. An example of an Azure Resource Manager template with this property setting is below.

```
{  
  "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",  
    "disableKeyBasedMetadataWriteAccess": true  
  }  
}
```

## Next steps

- [What is role-based access control \(RBAC\) for Azure resources](#)
- [Custom roles for Azure resources](#)
- [Azure Cosmos DB resource provider operations](#)

# Advanced Threat Protection for Azure Cosmos DB (Preview)

2/25/2020 • 3 minutes to read • [Edit Online](#)

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

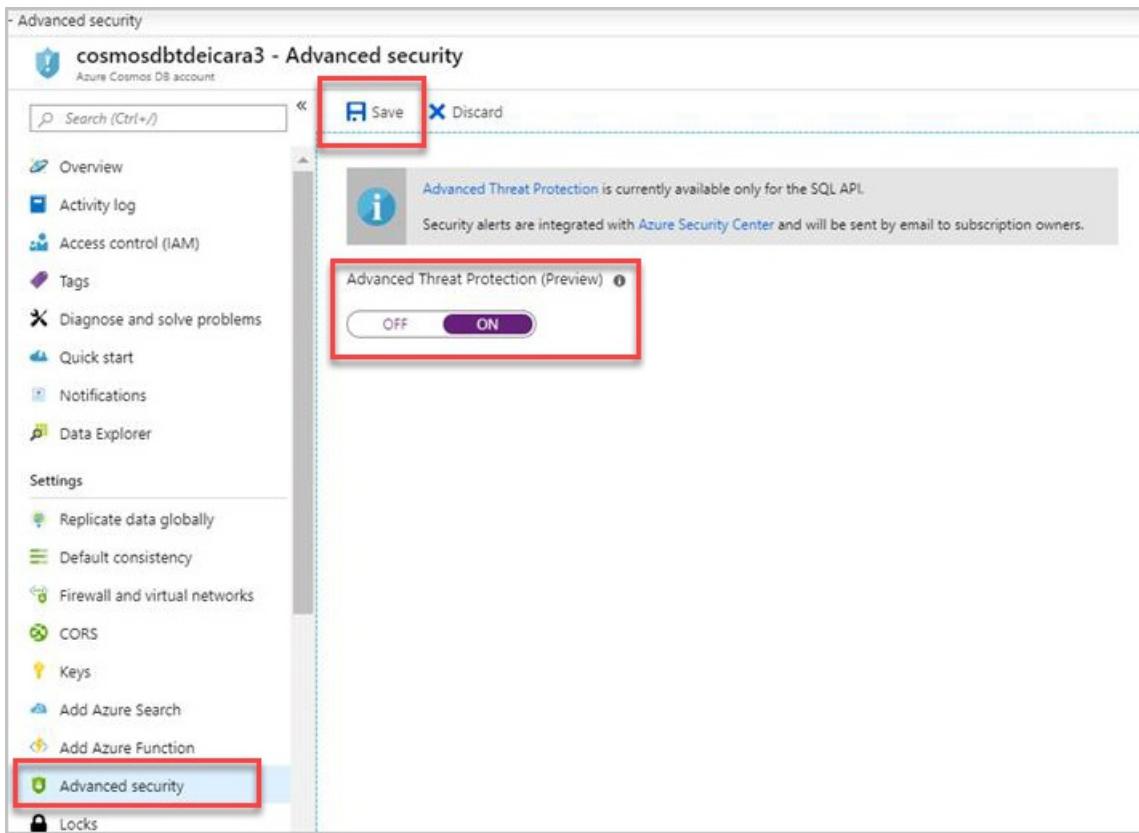
Advanced Threat Protection for Azure Cosmos DB detects anomalous activities indicating unusual and potentially harmful attempts to access or exploit databases. It can currently trigger the following alerts:

- **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.

## Set up Advanced Threat Protection

### Set up ATP using the portal

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.

### Set up ATP using REST API

Use Rest API commands to create, update, or get the Advanced Threat Protection setting for a specific Azure Cosmos DB account.

- [Advanced Threat Protection - Create](#)
- [Advanced Threat Protection - Get](#)

### Set up ATP using Azure PowerShell

Use the following PowerShell cmdlets:

- [Enable Advanced Threat Protection](#)
- [Get Advanced Threat Protection](#)
- [Disable Advanced Threat Protection](#)

### Using Azure Resource Manager templates

Use an Azure Resource Manager template to set up Cosmos DB with Advanced Threat Protection enabled. For more information, see [Create a CosmosDB Account with Advanced Threat Protection](#).

### Using Azure Policy

Use an Azure Policy to enable Advanced Threat Protection for Cosmos DB.

1. Launch the Azure **Policy - Definitions** page, and search for the **Deploy Advanced Threat Protection for Cosmos DB** policy.

The screenshot shows the 'Policy - Definitions' blade in the Azure portal. On the left, there's a navigation menu with items like Overview, Getting started, Join Preview, Compliance, Remediation, Authoring, Assignments, and Definitions. The 'Definitions' item is selected. The main area displays a table with one row for the policy definition. The columns include NAME, DEFINITION LOCATION, POLICIES, TYPE, DEFINITION TYPE, and CATEGORY. The policy name is 'Deploy Advanced Threat Protection for Cosmos DB Accounts', located in 'Cosmos DB' and is of type 'Policy'.

- Click on the **Deploy Advanced Threat Protection for CosmosDB** policy, and then click **Assign**.

This screenshot shows the details of the 'Deploy Advanced Threat Protection for Cosmos DB Accounts' policy. It includes fields for Name (Deploy Advanced Threat Protection for Cosmos DB Accounts), Description (This policy enables Advanced Threat Protection across Cosmos DB accounts.), Effect ([parameters('effect')]), and Category (Cosmos DB). The 'Assign' button is highlighted with a red box.

- From the **Scope** field, click the three dots, select an Azure subscription or resource group, and then click **Select**.

This screenshot shows the 'Assign policy' blade for the selected policy. It has sections for SCOPE, BASICS, PARAMETERS, and MANAGED IDENTITY. The 'SCOPE' section is expanded, showing a 'Scope' dropdown with a three-dot ellipsis button highlighted with a red box. A separate 'Scope' selection dialog is shown on the right, also with a red box around its header. This dialog has 'Subscription' and 'Resource Group' dropdowns, both currently empty. At the bottom of the dialog are 'Select', 'Cancel', and 'Clear All Selections' buttons, with the 'Select' button highlighted with a red box.

- Enter the other parameters, and click **Assign**.

## Manage ATP security alerts

When Azure Cosmos DB activity anomalies occur, a security alert is triggered with information about the

suspicious security event.

From Azure Security Center, you can review and manage your current [security alerts](#). Click on a specific alert in [Security Center](#) to view possible causes and recommended actions to investigate and mitigate the potential threat. The following image shows an example of alert details provided in Security Center.

**Access from an unusual location to a Cosmos DB account**

[cosmosdbtdeicara3](#)

[Learn more](#)

### General information

DESCRIPTION	Someone has accessed your Azure Cosmos DB account 'cosmosdbtdeicara3' from an unusual location.
ACTIVITY TIME	Thursday, July 18, 2019, 3:00:00 PM
SEVERITY	<span style="color: red;">!</span> High
STATE	Active
ATTACKED RESOURCE	[REDACTED]
SUBSCRIPTION	[REDACTED]
DETECTED BY	Microsoft
ENVIRONMENT	Azure
RESOURCE TYPE	 Cosmos DB
ALERT ID	824f4a35-76b3-467e-86b8-3acec0eba83e
CLIENT IP ADDRESS	[REDACTED]
CLIENT LOCATION	[REDACTED]
USER AGENT	dummyAgent
DATABASE ID	[REDACTED]
COLLECTION ID	[REDACTED]
POTENTIAL CAUSES	<p>This alert indicates that this account has been accessed successfully from an IP address that is unfamiliar and unexpected compared to recent access patterns.</p> <p>Potential causes:</p> <ul style="list-style-type: none"><li>An attacker has accessed your Cosmos DB account.</li><li>A legitimate user has accessed your Cosmos DB account from a new location.</li></ul>

### Remediation steps

INVESTIGATION STEPS	<p>See how to <a href="#">configure Cosmos DB Analytics logging</a>.</p> <ul style="list-style-type: none"><li>Limit access to your Cosmos DB account, following the 'least privilege' principle: <a href="https://go.microsoft.com/fwlink/?linkid=2097419">https://go.microsoft.com/fwlink/?linkid=2097419</a></li><li>Revoke all Cosmos DB access tokens that may be compromised and ensure that your access tokens are only shared with authorized users.</li><li>Ensure that Cosmos DB access tokens are stored in a secured location such as Azure Key Vault. Avoid storing or sharing Cosmos DB access tokens in the source code, documentation, and email.</li></ul>
REMEDIATION STEPS	

An email notification is also sent with the alert details and recommended actions. The following image shows an example of an alert email.

 Microsoft Azure

**HIGH SEVERITY**

Azure Security Center has detected a potential security threat on your environment



## Access from an unusual location to a Cosmos DB account

Someone has accessed your Azure Cosmos DB account  
[REDACTED] from an unusual location.

July 18, 2019 10:00 UTC



Attacked resource [REDACTED]



Detected by Microsoft

[Explore in Azure Security Center >](#)

### Activity details

Subscription ID [REDACTED]

Subscription name [REDACTED]

Resource type CosmosDB

Database ID [REDACTED]

Collection ID [REDACTED]

User agent dummyAgent

Client IP address [REDACTED]

Client location [REDACTED]

Potential causes This alert indicates that this account has been accessed successfully from an IP address that is unfamiliar and unexpected compared to recent access patterns. Potential causes:

- An attacker has accessed your Cosmos DB account.
- A legitimate user has accessed your Cosmos DB account from a new location.

Investigation steps View related Cosmos DB activity using Cosmos DB Analytics Logging. See [how to configure Cosmos DB Analytics logging](#).

Remediation steps

- Limit access to your Cosmos DB account, following the 'least privilege' principle.
- Revoke all Cosmos DB access tokens that may be compromised and ensure that your access tokens are only shared with authorized users.
- Ensure that Cosmos DB access tokens are stored in a secured location such as Azure Key Vault. Avoid storing or sharing Cosmos DB access tokens in the source code, documentation, and email.

Alert ID dcf8e8ca-a94b-11e9-a716-000d3a6a0922

## Cosmos DB ATP alerts

To see a list of the alerts generated when monitoring Azure Cosmos DB accounts, see the [Cosmos DB alerts](#) section in the Azure Security Center documentation.

## Next steps

- Learn more about [Diagnostic logging in Azure Cosmos DB](#)
- Learn more about [Azure Security Center](#)

# Security controls for Azure Cosmos DB

1/26/2020 • 2 minutes to read • [Edit Online](#)

This article documents the security controls built into Azure Cosmos DB.

A security control is a quality or feature of an Azure service that contributes to the service's ability to prevent, detect, and respond to security vulnerabilities.

For each control, we use "Yes" or "No" to indicate whether it is currently in place for the service, "N/A" for a control that is not applicable to the service. We might also provide a note or links to more information about an attribute.

## Network

SECURITY CONTROL	YES/NO	NOTES
Service endpoint support	Yes	
VNet injection support	Yes	With VNet service endpoint, you can configure an Azure Cosmos DB account to allow access only from a specific subnet of a virtual network (VNet). You can also combine VNet access with firewall rules. To learn more, see <a href="#">Access Azure Cosmos DB from virtual networks</a> .
Network Isolation and Firewall support	Yes	With firewall support, you can configure your Azure Cosmos account to allow access only from an approved set of IP addresses, a range of IP addresses and/or cloud services. To learn more, see <a href="#">Configure IP firewall in Azure Cosmos DB</a> .
Forced tunneling support	Yes	Can be configured at the client side on the VNet where the virtual machines are located.

## Monitoring & logging

SECURITY CONTROL	YES/NO	NOTES
Azure monitoring support (Log analytics, App insights, etc.)	Yes	All requests that are sent to Azure Cosmos DB are logged. <a href="#">Azure Monitoring</a> , Azure Metrics, Azure Audit Logging are supported. You can log information corresponding to data plane requests, query runtime statistics, query text, MongoDB requests. You can also set up alerts.

SECURITY CONTROL	YES/NO	NOTES
Control and management plane logging and audit	Yes	Azure Activity log for account level operations such as Firewalls, VNets, Keys access, and IAM.
Data plane logging and audit	Yes	Diagnostics monitoring logging for container level operations such as create container, provision throughput, indexing policies, and CRUD operations on documents.

## Identity

SECURITY CONTROL	YES/NO	NOTES
Authentication	Yes	Yes at the Database Account Level; at the data plane level, Cosmos DB uses resource tokens and key access.
Authorization	Yes	Supported at the Azure Cosmos account with Master keys (primary and secondary) and Resource tokens. You can get read/write or read only access to data with master keys. Resource tokens allow limited time access to resources such as documents and containers.

## Data protection

SECURITY CONTROL	YES/NO	NOTES
Server-side encryption at rest: Microsoft-managed keys	Yes	All Azure Cosmos databases and backups are encrypted by default; see <a href="#">Data encryption in Azure Cosmos DB</a> .
Server-side encryption at rest: customer-managed keys (BYOK)	Yes	See <a href="#">Configure customer-managed keys for your Azure Cosmos DB account</a>
Column level encryption (Azure Data Services)	Yes	Only in the Tables API Premium. Not all APIs support this feature. See <a href="#">Introduction to Azure Cosmos DB: Table API</a> .
Encryption in transit (such as ExpressRoute encryption, in VNet encryption, and VNet-VNet encryption )	Yes	All Azure Cosmos DB data is encrypted at transit.

SECURITY CONTROL	YES/NO	NOTES
API calls encrypted	Yes	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 <a href="mailto:azurecosmosdbtls@service.microsoft.com">azurecosmosdbtls@service.microsoft.com</a> .

## Configuration management

SECURITY CONTROL	YES/NO	NOTES
Configuration management support (versioning of configuration, etc.)	No	

## Additional security controls for Cosmos DB

SECURITY CONTROL	YES/NO	NOTES
Cross Origin Resource Sharing (CORS)	Yes	See <a href="#">Configure Cross-Origin Resource Sharing (CORS)</a> .

## Next steps

- Learn more about the [built-in security controls across Azure services](#).

# Online backup and on-demand data restore in Azure Cosmos DB

12/16/2019 • 5 minutes to read • [Edit Online](#)

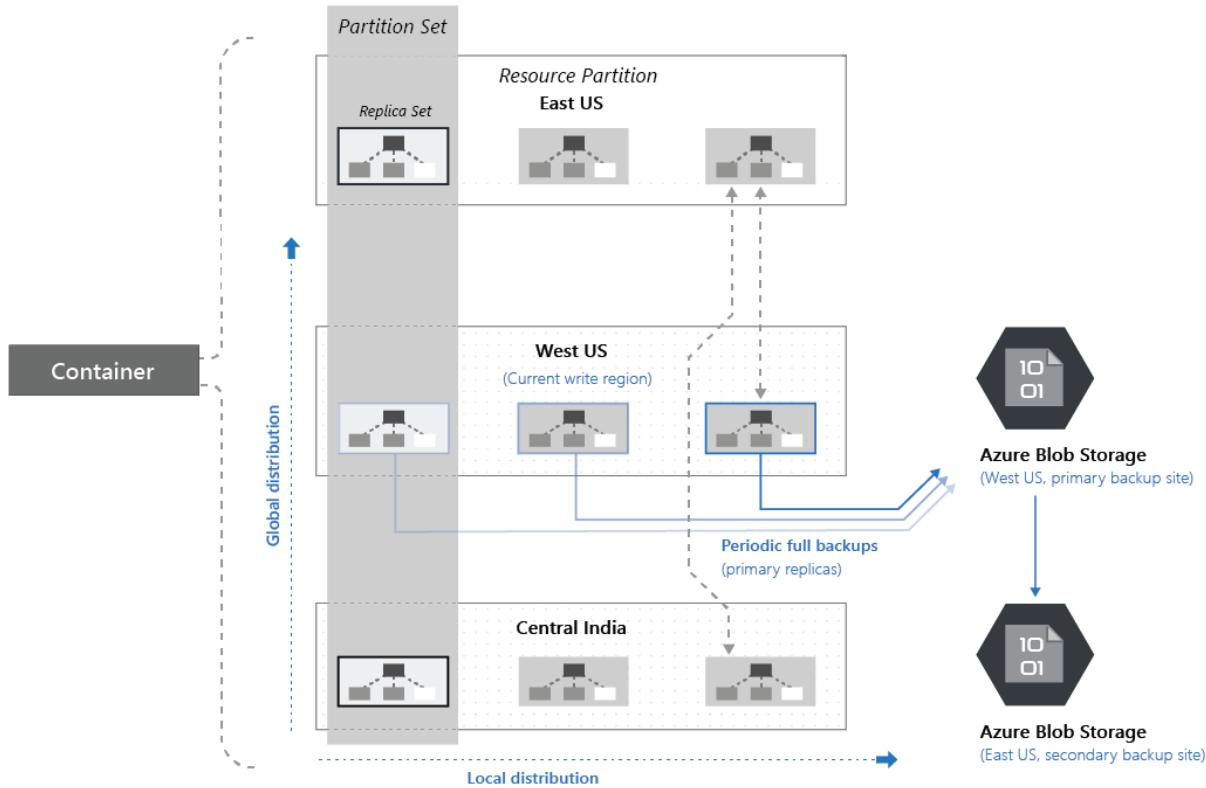
Azure Cosmos DB automatically takes backups of your data at regular intervals. The automatic backups are taken without affecting the performance or availability of the database operations. All the backups are stored separately in a storage service, and those backups are globally replicated for resiliency 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 backup of your database every 4 hours and at any point of time, only the latest 2 backups are stored. However, 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-master configuration) of your Azure Cosmos database account. 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 will use this backup only if a backup restore is initiated.
- 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.
- 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.

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:



## 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, as well as for incremental changes, and store it in your own storage.

## Backup retention period

Azure Cosmos DB takes snapshots of your data every four hours. 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.

## Restoring data from online backups

Accidental deletion or modification of data can happen in one of the following scenarios:

- The entire Azure Cosmos account is deleted
- One or more Azure Cosmos databases are deleted
- One or more Azure Cosmos containers are deleted
- Azure Cosmos items (for example, documents) within a container are deleted or modified. 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. The restore process always creates a new Azure Cosmos account to hold the restored data. The name of the new account, if not specified, will have the format `<Azure_Cosmos_account_original_name>-restored1`. The last digit is incremented, if multiple restores are attempted. You can't restore data to a pre-created Azure Cosmos account.

When an Azure Cosmos account is deleted, we can restore the data into an account with the same name, provided that the account name is not in use. In such cases, it's recommended to not re-create the account after deletion, because it not only prevents the restored data to use the same name, but also makes discovering the right account to restore from more difficult.

When an Azure Cosmos database is deleted, it is possible to restore the whole database or a subset of the containers within that database. It is also possible to select containers across databases and restore them and all the restored data is placed in a new Azure Cosmos account.

When one or more items within a container are accidentally deleted or changed (the data corruption case), you will need to specify the time to restore to. Time is of essence for this case. Since 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 the case of deletes, your data is no longer stored because they won't be overwritten by the backup cycle. Backups for deleted databases or containers are saved for 30 days.

If you provision throughput at the database level (that is, where a set of containers shares the provisioned throughput), the backup and restore process in this case happen at the entire database level, and not at the individual containers level. In such cases, selecting a subset of containers to restore is not an option.

## Migrating data to the original account

The primary goal of the data restore is to provide a way to recover any data that you delete or modify accidentally. So, we recommend that you first inspect the content of the recovered data to ensure it contains what you are expecting. Then work on migrating the data back to the primary account. Although it is possible to use the restored account as the live account, it's not a recommended option if you have production workloads.

The following are different ways to migrate data back to the original Azure Cosmos account:

- Using [Cosmos DB Data Migration Tool](#)
- Using [Azure Data Factory](#)
- Using [change feed](#) in Azure Cosmos DB
- Write custom code

Delete the restored accounts as soon as you are done migrating, because they will incur ongoing charges.

## 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](#)
- [How to restore data from an Azure Cosmos account](#)
- [Use Cosmos DB change feed](#) to move data to Azure Cosmos DB.
- [Use Azure Data Factory](#) to move data to Azure Cosmos DB.

# Compliance in Azure Cosmos DB

12/5/2019 • 2 minutes to read • [Edit Online](#)

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'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:

GLOBALY 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			
ISO 27018:2014			
ISO 9001:2015			

GLOBALY APPLICABLE	US GOVERNMENT	INDUSTRY SPECIFIC	REGION OR COUNTRY SPECIFIC
SOC 1, 2, 3			

To learn more about each of these compliance offerings and how they benefit you, see [Overview of Microsoft Azure compliance](#) page.

The following table lists the certifications supported by Azure Cosmos DB in Azure Government:

GLOBALY 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 the [Overview of Azure compliance](#).
- For an overview of Azure Cosmos DB security and the latest improvements, see [Azure Cosmos database security](#) article.
- For more information about Microsoft certifications, see the [Azure Trust Center](#).

# Azure Cosmos DB service quotas

2/28/2020 • 8 minutes to read • [Edit Online](#)

This article provides an overview of the default quotas offered to different resources in the Azure Cosmos DB.

## Storage and throughput

After you create an Azure Cosmos account under your subscription, you can manage data in your account by [creating databases, containers, and items](#). 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 ( <a href="#">dedicated throughput provisioned mode</a> )	1,000,000 by default. You can increase it by <a href="#">filing an Azure support ticket</a>
Maximum RUs per database ( <a href="#">shared throughput provisioned mode</a> )	1,000,000 by default. You can increase it by <a href="#">filing an Azure support ticket</a>
Maximum RUs per (logical) partition key	10,000
Maximum storage across all items per (logical) partition key	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 RUs required per 1 GB	10 RU/s

### 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](#).

A Cosmos container (or shared throughput database) must have a minimum throughput of 400 RUs. As the container grows, the minimum supported throughput also depends on the following factors:

- The minimum throughput that you can set on a container depends on the maximum throughput ever provisioned on the container. For example, if your throughput was increased to 10000 RUs, then the lowest possible provisioned throughput would be 1000 RUs
- The minimum throughput on a shared throughput database also depends on the total number of containers that you have ever created in a shared throughput database, measured at 100 RUs per container. For example, if you have created five containers within a shared throughput database, then the throughput must be at least 500 RUs

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](#).

**NOTE**

In some cases, you may be able to lower throughput to lesser than 10%. Use the API to get the exact minimum RUs per container.

In summary, here are the minimum provisioned RU limits.

RESOURCE	DEFAULT LIMIT
Minimum RUs per container ( <a href="#">dedicated throughput provisioned mode</a> )	400
Minimum RUs per database ( <a href="#">shared throughput provisioned mode</a> )	400
Minimum RUs per container within a shared throughput database	100

Cosmos DB supports elastic scaling of throughput (RUs) per container or database via the SDKs or portal. Each container can scale synchronously and immediately within a scale range of 10 to 100 times, between minimum and maximum values. 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.

## 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 <a href="#">filing an Azure support ticket</a>
Maximum number of regional failovers	1/hour by default. You can increase it by <a href="#">filing an Azure support ticket</a>

**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

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)

## 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 *

\* You can increase any of these per-container limits by contacting Azure Support.

## 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 nesting depth	No practical limit
Maximum length of property name	No practical limit

RESOURCE	DEFAULT LIMIT
Maximum length of property value	No practical limit
Maximum length of string property value	No practical limit
Maximum length of numeric property value	IEEE754 double-precision 64-bit

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 master 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 master 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 <a href="#">filing an Azure support ticket</a>
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.

## Autopilot mode limits

See the [Autopilot](#) article for the throughput and storage limits in autopilot mode.

## 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 ANDs per query	2000 *
Maximum ORs per query	2000 *
Maximum UDFs per query	10 *
Maximum arguments per IN expression	6000 *
Maximum points per polygon	4096 *

\* You can increase any of these SQL query limits by contacting Azure Support.

## 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	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 (can be renewed any number of times)
Maximum containers per subscription (SQL, Gremlin, Table API)	1

RESOURCE	DEFAULT LIMIT
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.

## 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](#)

# Introduction to the Azure Cosmos DB Cassandra API

2/25/2020 • 2 minutes to read • [Edit Online](#)

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.

## 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.

**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 guaranteed 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.

## 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

2/26/2020 • 5 minutes to read • [Edit Online](#)

Azure Cosmos DB is Microsoft's globally distributed multi-model database service. You can communicate with the Azure Cosmos DB Cassandra API through Cassandra Query Language (CQL) 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 CQL version **v4**. 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:

- ascii
- bigint
- blob
- boolean
- counter
- date
- decimal
- double
- float
- frozen
- inet
- int

- list
- set
- smallint
- text
- time
- timestamp
- timeuuid
- tinyint
- tuple
- uuid
- varchar
- varint
- tuples
- udts
- map

## CQL functions

Azure Cosmos DB Cassandra API supports the following CQL functions:

- Token
- Aggregate functions
  - min, max, avg, count
- Blob conversion functions
  - typeAsBlob(value)
  - blobAsType(value)
- UUID and timeuuid functions
  - dateOf()
  - now()
  - minTimeuuid()
  - unixTimestampOf()
  - toDate(timeuuid)
  - toTimestamp(timeuuid)
  - toUnixTimestamp(timeuuid)
  - toDate(timestamp)
  - toUnixTimestamp(timestamp)
  - toTimestamp(date)
  - toUnixTimestamp(date)

## 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.

## CQL Shell

CQLSH command-line utility comes with Apache Cassandra 3.1.1 and works out of box by setting some environment variables.

### 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://www.apache.org/dist/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
```

## CQL commands

Azure Cosmos DB supports the following database commands on Cassandra API accounts.

- CREATE KEYSPACE (The replication settings for this command are ignored)
- CREATE TABLE
- ALTER TABLE
- USE
- INSERT
- SELECT
- UPDATE
- BATCH - Only unlogged commands are supported
- DELETE

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 role-based access control (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 ;
```

## 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 rate limitation, the [spark](#) and the [Java](#) extensions are provided. If you use

other SDKs to access Cassandra API in Azure Cosmos DB, create a connection policy to retry on these exceptions.

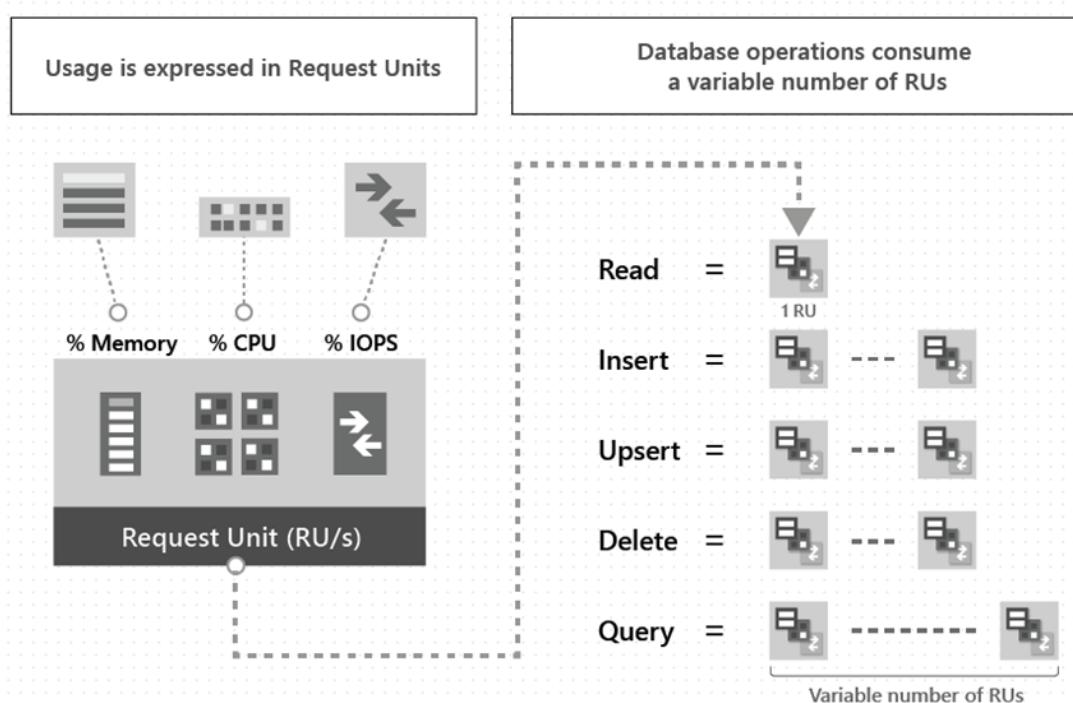
## Next steps

- Get started with [creating a Cassandra API account, database, and a table](#) by using a Java application

# Elastically scale an Azure Cosmos DB Cassandra API account

2/19/2020 • 3 minutes to read • [Edit Online](#)

There are a variety of options to explore the elastic nature of the Azure Cosmos DB's 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.



## 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 Autopilot](#)

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 [Autopilot](#).

## 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 Autopilot. The disadvantage of this approach is that it can be quite complex to implement retries while rate limiting may increase latency.

## Use Autopilot

In addition to manual or programmatic way of provisioning throughput, you can also configure Azure cosmos containers in Autopilot mode. Autopilot mode 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 autopilot mode](#) article.

The advantage of this approach is that it is the easiest way to manage the scaling needs in your system. It guarantees not to apply rate-limiting **within the configured RU ranges**. The disadvantage is that, if the scaling needs in your system are predictable, Autopilot may be a less cost-effective way of handling your scaling needs than using the bespoke control plane or SDK level approaches mentioned above.

## 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

5/17/2019 • 6 minutes to read • [Edit Online](#)

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.

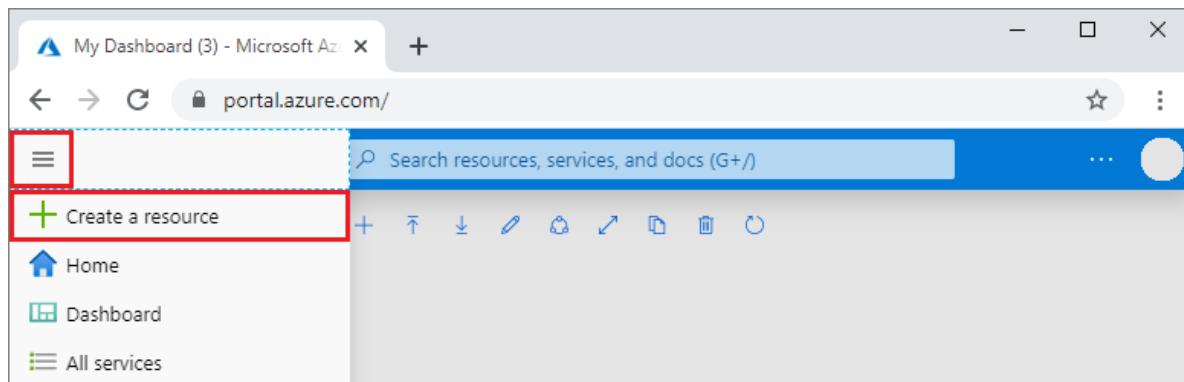
In addition, you need:

- 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**.

New

Search the Marketplace

Azure Marketplace [See all](#)

- Get started
- Recently created
- AI + Machine Learning
- Analytics
- Blockchain
- Compute
- Containers
- Databases**
- Developer Tools
- DevOps
- Identity
- Integration
- Internet of Things
- Media
- Mixed Reality

Featured [See all](#)

	Azure SQL Managed Instance <a href="#">Quickstart tutorial</a>
	SQL Database <a href="#">Quickstart tutorial</a>
	Azure Synapse Analytics (formerly SQL DW) <a href="#">Quickstart tutorial</a>
	Azure Database for MariaDB <a href="#">Learn more</a>
	Azure Database for MySQL <a href="#">Quickstart tutorial</a>
	Azure Database for PostgreSQL <a href="#">Quickstart tutorial</a>
	<b>Azure Cosmos DB</b> <a href="#">Quickstart tutorial</a>

- 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 <b>Create new</b> . 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 <code>cassandra.cosmos.azure.com</code> 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	<p>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.</p> <p>Select <b>Cassandra</b>, because in this quickstart you are creating a table that works with the Cassandra API.</p> <p><a href="#">Learn more about the Cassandra API.</a></p>
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.

Select **Review + Create**. You can skip the **Network** and **Tags** section.

### Create Azure Cosmos DB Account

[Basics](#) [Networking](#) [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 *	A Subscription
Resource Group *	<input type="button" value="Select existing..."/> <input type="button" value="Create new"/>

**Instance Details**

Account Name *	Enter account name
API * ⓘ	Cassandra
Apache Spark ⓘ	<input type="radio" value="Notebooks (preview)"/> Notebooks (preview) <input type="radio" value="Notebooks with Apache Spark (preview)"/> Notebooks with Apache Spark (preview) <input type="radio" value="None"/> None <a href="#">Sign up for Apache Spark preview</a>
Location *	(US) West US
Geo-Redundancy ⓘ	<input type="button" value="Enable"/> <input type="button" value="Disable"/>
Multi-region Writes ⓘ	<input type="button" value="Enable"/> <input type="button" value="Disable"/>

\*Up to 33% off multi-region writes is available to qualifying new accounts only. Accounts must be created between December 1, 2019 and February 29, 2020. 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 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.

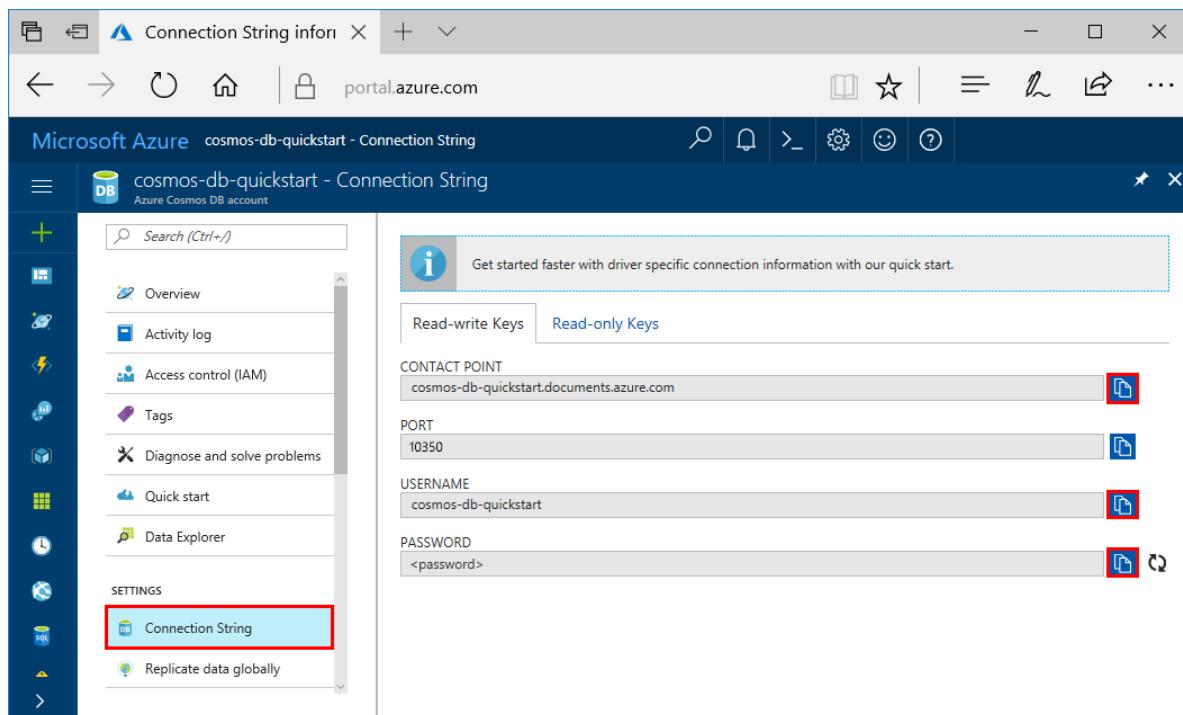
```
mapper.FirstOrDefault<User>("Select * from user where user_id = ?", 3);
```

## 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**.

Use the  button on the right side of the screen to copy the USERNAME value.



2. In Visual Studio, open the Program.cs file.

3. 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";
```

4. 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...==";
```

5. 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
```

6. 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.

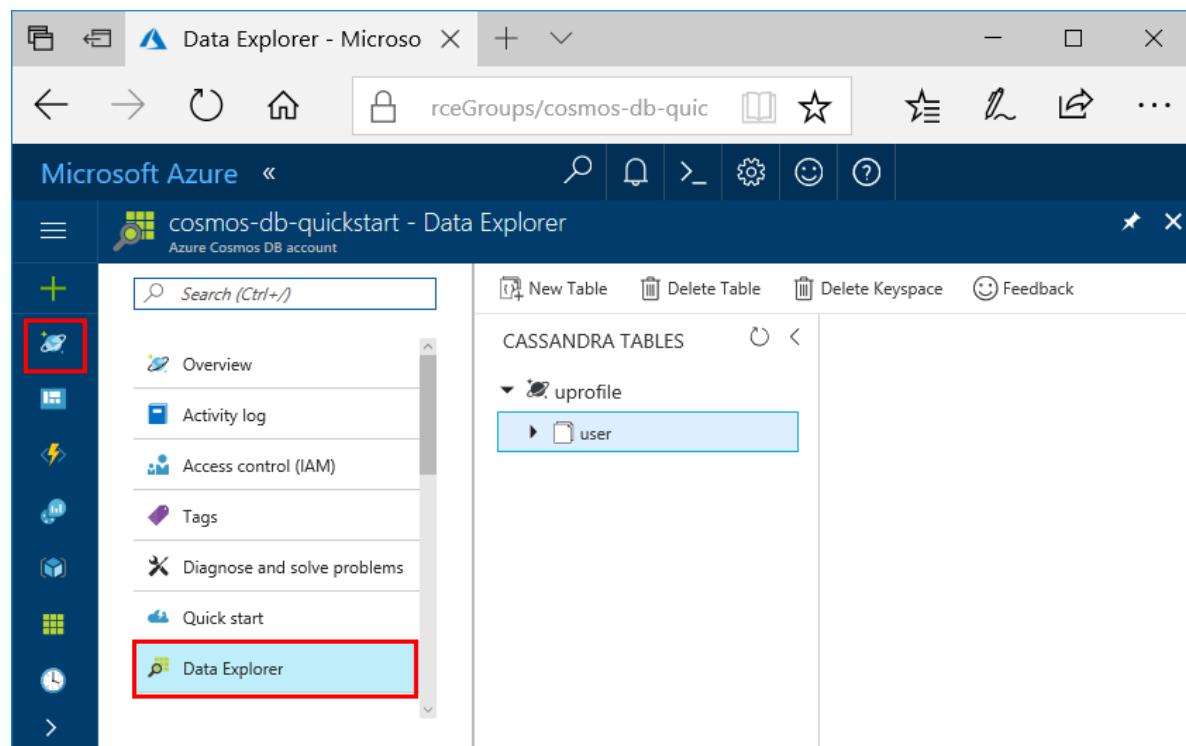
```
Install-Package CassandraCSharpDriver
```

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.

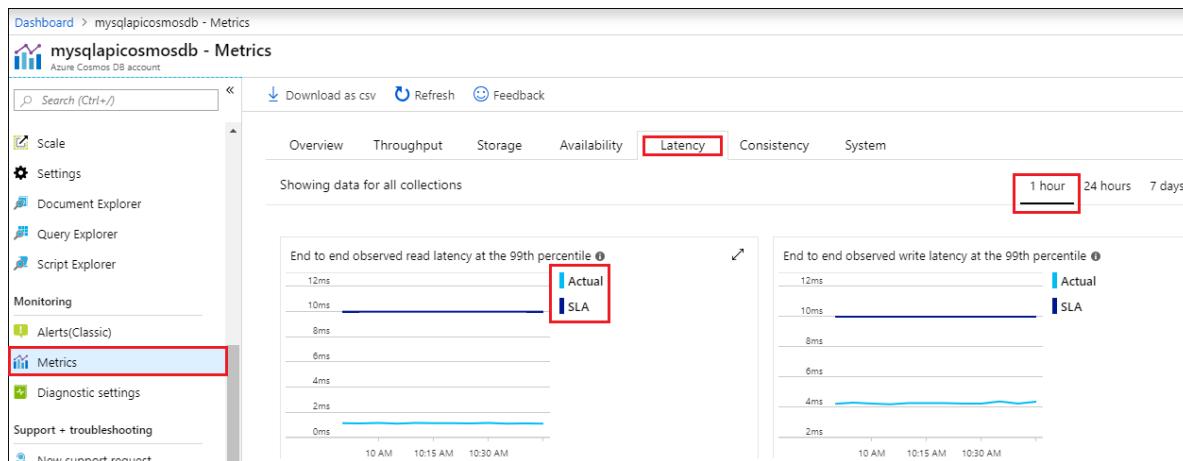


## Review SLAs in the Azure portal

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.

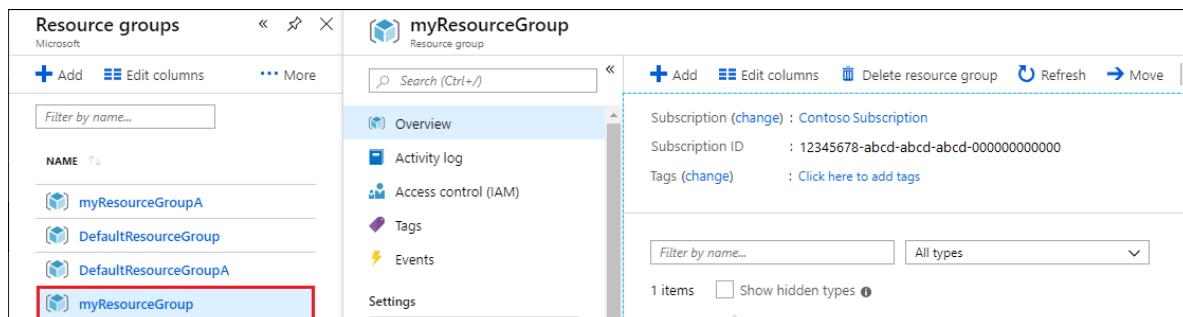


3. Review the metrics on the other tabs.

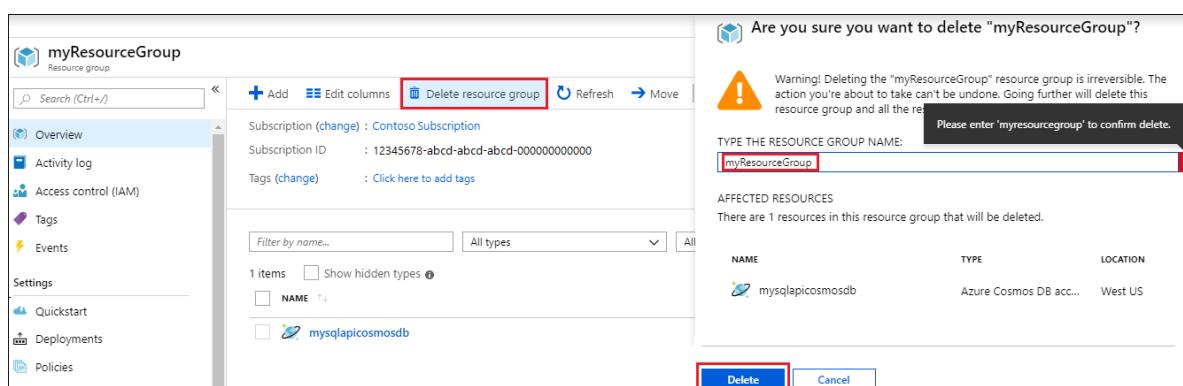
## Clean up resources

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 Java app to manage Azure Cosmos DB Cassandra API data

2/11/2020 • 7 minutes to read • [Edit Online](#)

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. 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.

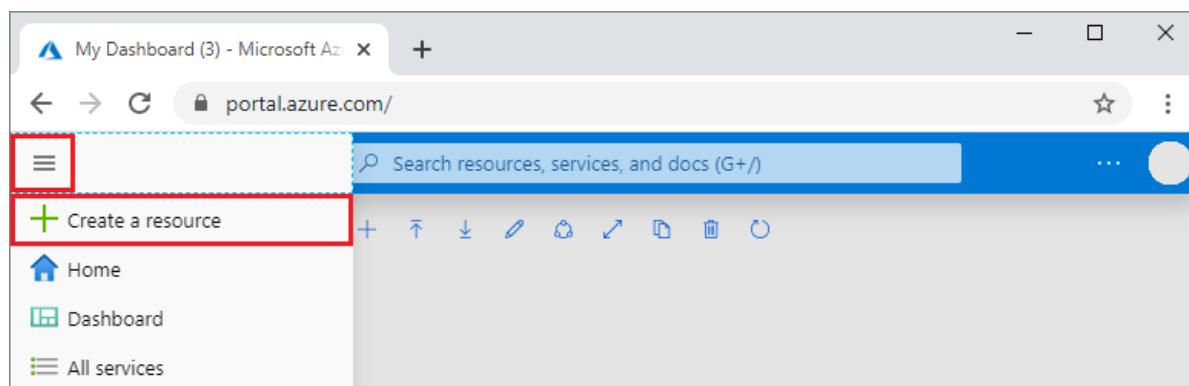
## Prerequisites

- 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**.

New

Search the Marketplace

Azure Marketplace See all

Featured See all

Get started	 Azure SQL Managed Instance <a href="#">Quickstart tutorial</a>
Recently created	 SQL Database <a href="#">Quickstart tutorial</a>
AI + Machine Learning	 Azure Synapse Analytics (formerly SQL DW) <a href="#">Quickstart tutorial</a>
Analytics	
Blockchain	 Azure Database for MariaDB <a href="#">Learn more</a>
Compute	 Azure Database for MySQL <a href="#">Quickstart tutorial</a>
Containers	 Azure Database for PostgreSQL <a href="#">Quickstart tutorial</a>
Databases	 Azure Cosmos DB <a href="#">Quickstart tutorial</a>
Developer Tools	
DevOps	
Identity	
Integration	
Internet of Things	
Media	
Mixed Reality	

- 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 <b>Create new</b> . 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	<p>Enter a unique name to identify your Azure Cosmos DB account. Your account URI will be <i>cassandra.cosmos.azure.com</i> appended to your unique account name.</p> <p>The account name can use only lowercase letters, numbers, and hyphens (-), and must be between 3 and 31 characters long.</p>

SETTING	VALUE	DESCRIPTION
API	Cassandra	<p>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.</p> <p>Select <b>Cassandra</b>, because in this quickstart you are creating a table that works with the Cassandra API.</p> <p><a href="#">Learn more about the Cassandra API.</a></p>
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.

Select **Review+Create**. You can skip the **Network** and **Tags** section.

**Create Azure Cosmos DB Account**

Basics Networking 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 \* A Subscription

Resource Group \* Select existing... [Create new](#)

**Instance Details**

Account Name \* Enter account name

API \* **Cassandra**

Apache Spark [Notebooks \(preview\)](#) [Notebooks with Apache Spark \(preview\)](#) **None** [Sign up for Apache Spark preview](#)

Location \* (US) West US

Geo-Redundancy **Enable** **Disable**

Multi-region Writes **Enable** **Disable**

\*Up to 33% off multi-region writes is available to qualifying new accounts only. Accounts must be created between December 1, 2019 and February 29, 2020. 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

- 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 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(cassandraUserName, 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.

- In your Azure Cosmos DB account in the [Azure portal](#), select **Connection String**.

The screenshot shows the Azure Cosmos DB portal with the title "cosmos-db-quickstart - Connection String". On the left sidebar, under "Connection String", there is a red box highlighting the "Connection String" option. The main pane displays connection details for a Cassandra account:

- CONTACT POINT:** cosmos-db-quickstart.cassandra.cosmos.azure.com
- PORT:** 10350
- USERNAME:** cosmos-db-quickstart
- PRIMARY PASSWORD:** yDSiPeqBtLweW4JR3t7705kr57tiWEDK7t4bQY6gpzaW1yacpSQnbafuEVVz3UiPoD6... (with a copy icon)
- SECONDARY PASSWORD:** ARZ7ZZRpW8WzCzsP1tNq67MUEli6tcf5IGx0kSCqvCNAWck9MLR4ohRsZ1RADzaRx... (with a copy icon)
- PRIMARY CONNECTION STRING:** AccountEndpoint=cosmos-db-quickstart.cassandra.cosmos.azure.com;AccountKey=... (with a copy icon)
- SECONDARY CONNECTION STRING:** AccountEndpoint=cosmos-db-quickstart.cassandra.cosmos.azure.com;AccountKey=... (with a copy icon)

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. Paste 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 SSL certificate, then replace `<SSL key store file location>` with the location of the 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 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\java-examples` folder.

```
cd "C:\git-samples\azure-cosmosdb-cassandra-java-getting-started\java-examples"
```

- In the git terminal window, use the following command to generate the `cosmosdb-cassandra-examples.jar` file.

```
mvn clean install
```

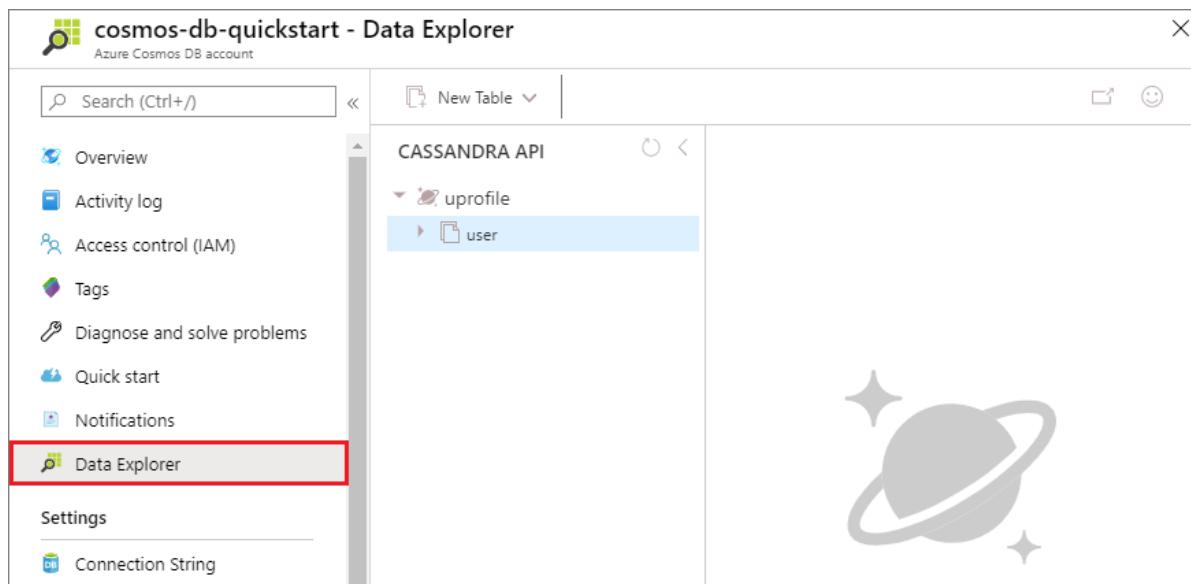
- In the git terminal window, run the following command to start the Java application.

```
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.

- In the Azure portal, open **Data Explorer** to query, modify, and work with this new data.

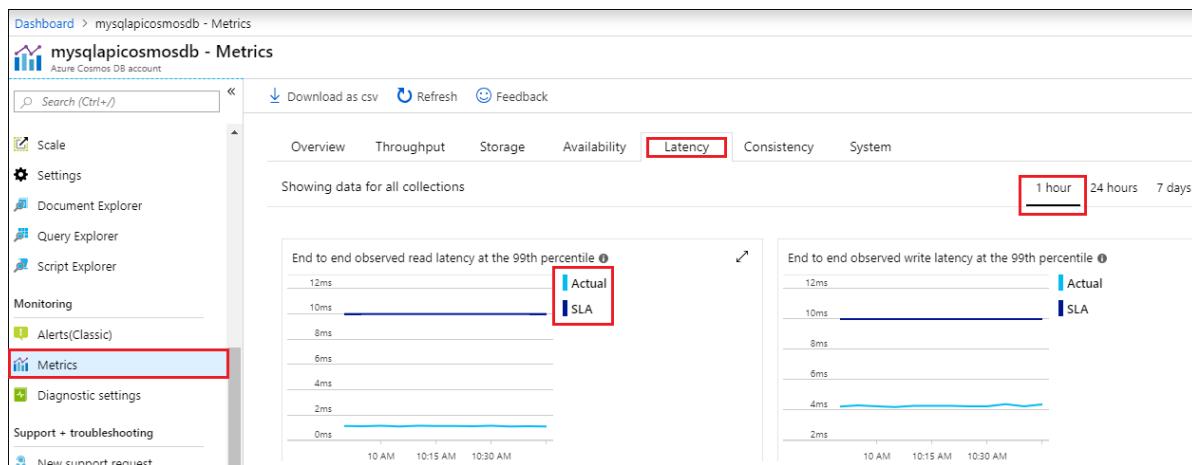


## Review SLAs in the Azure portal

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:

- Select **Metrics** in your Cosmos DB account's navigation menu.
- Select a tab such as **Latency**, and select a timeframe on the right. Compare the **Actual** and **SLA** lines on the charts.



- 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:

- In the Azure portal Search bar, search for and select **Resource groups**.
- From the list, select the resource group you created for this quickstart.

- On the resource group **Overview** page, select **Delete resource group**.

- 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

2/14/2020 • 7 minutes to read • [Edit Online](#)

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.

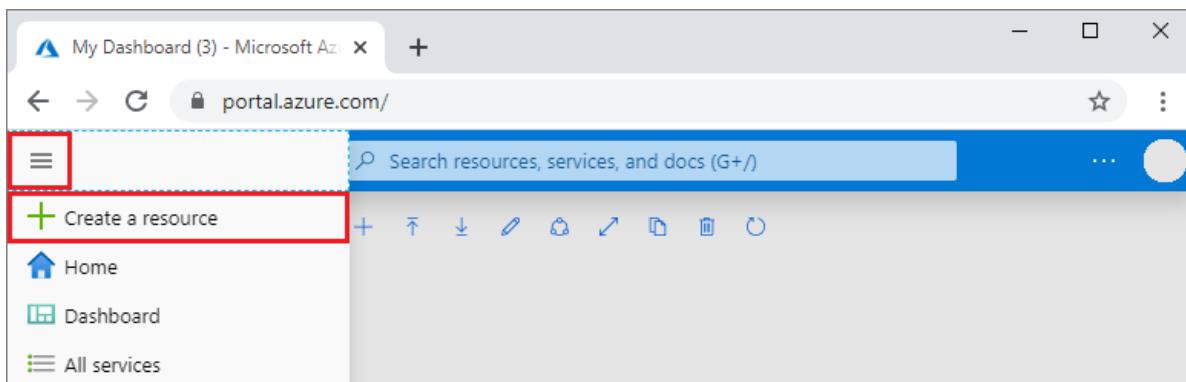
In addition, you need:

- [Nodejs](#) 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**.

New

Search the Marketplace

Azure Marketplace [See all](#)

Featured [See all](#)

Get started	 Azure SQL Managed Instance <a href="#">Quickstart tutorial</a>
Recently created	 SQL Database <a href="#">Quickstart tutorial</a>
AI + Machine Learning	 Azure Synapse Analytics (formerly SQL DW) <a href="#">Quickstart tutorial</a>
Analytics	
Blockchain	
Compute	
Containers	 Azure Database for MariaDB <a href="#">Learn more</a>
Databases	
Developer Tools	 Azure Database for MySQL <a href="#">Quickstart tutorial</a>
DevOps	
Identity	 Azure Database for PostgreSQL <a href="#">Quickstart tutorial</a>
Integration	
Internet of Things	 Azure Cosmos DB <a href="#">Quickstart tutorial</a>
Media	
Mixed Reality	 Azure Cosmos DB <a href="#">Quickstart tutorial</a>

- 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 <b>Create new</b> . 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	<p>Enter a unique name to identify your Azure Cosmos DB account. Your account URI will be <code>cassandra.cosmos.azure.com</code> appended to your unique account name.</p> <p>The account name can use only lowercase letters, numbers, and hyphens (-), and must be between 3 and 31 characters long.</p>

SETTING	VALUE	DESCRIPTION
API	Cassandra	<p>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.</p> <p>Select <b>Cassandra</b>, because in this quickstart you are creating a table that works with the Cassandra API.</p> <p><a href="#">Learn more about the Cassandra API.</a></p>
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.

Select **Review + Create**. You can skip the **Network** and **Tags** section.

**Create Azure Cosmos DB Account**

[Basics](#) [Networking](#) [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 \*

Resource Group \*  [Create new](#)

**Instance Details**

Account Name \*

API \*  Cassandra

Apache Spark [Notebooks \(preview\)](#) [Notebooks with Apache Spark \(preview\)](#) [None](#) [Sign up for Apache Spark preview](#)

Location \*

Geo-Redundancy [Enable](#) [Disable](#)

Multi-region Writes [Enable](#) [Disable](#)

\*Up to 33% off multi-region writes is available to qualifying new accounts only. Accounts must be created between December 1, 2019 and February 29, 2020. 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 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();
    });
}

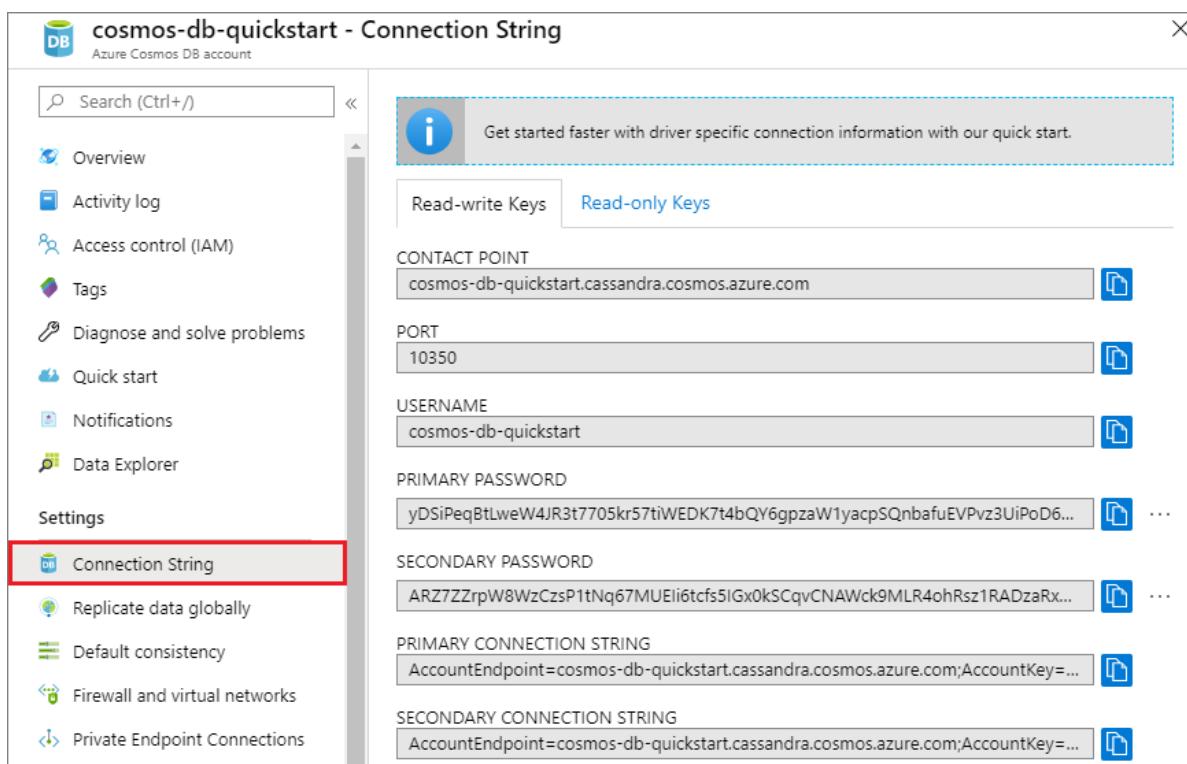
```

## 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**.

Use the  button on the right side of the screen to copy the top value, the CONTACT POINT.



2. Open the `config.js` file.

3. 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"
```

4. 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';
```

5. 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==';
```

6. 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

```
d4:00:20:d0:5e:66:fc:53:fe:1a:50:88:2c:78:db:28:52:ca:e4:74
```

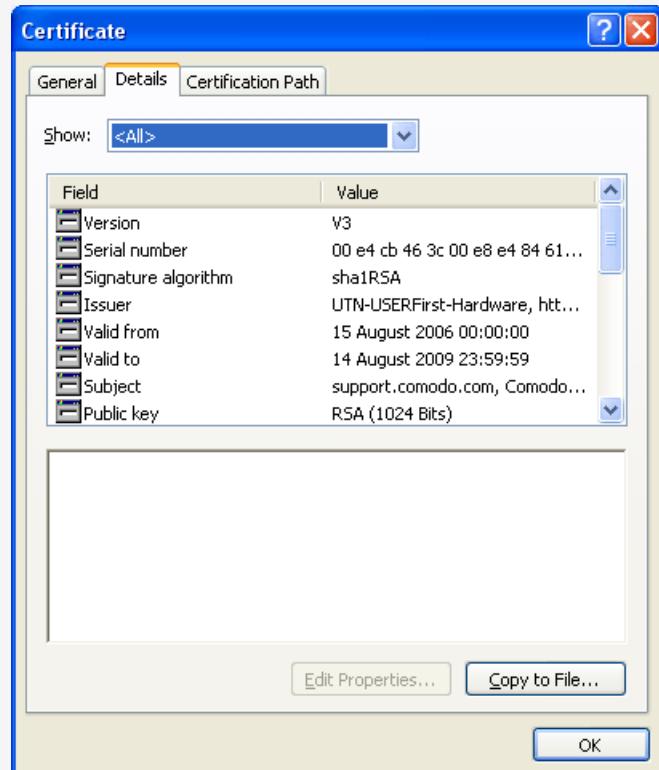
2. Open `uprofile.js` and change the `path\to\cert` to point to your new certificate.

3. Save `uprofile.js`.

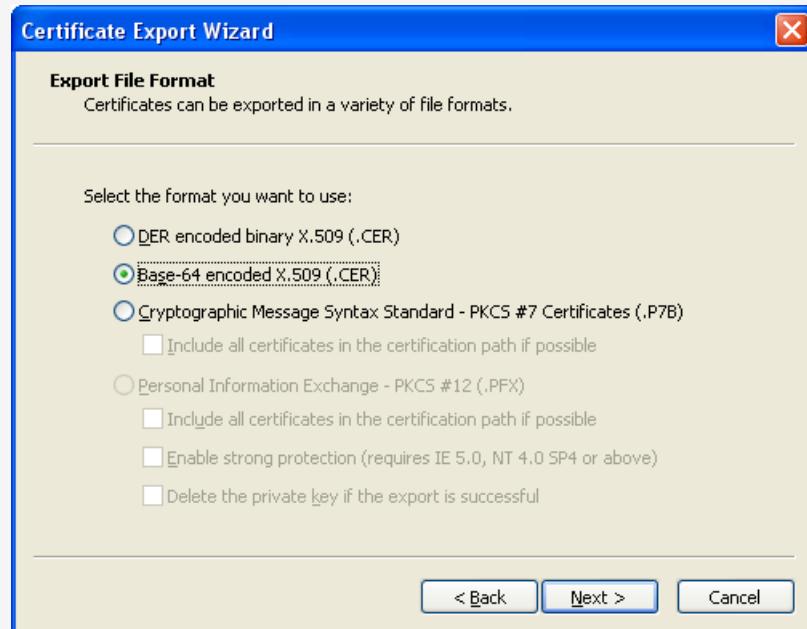
## NOTE

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:

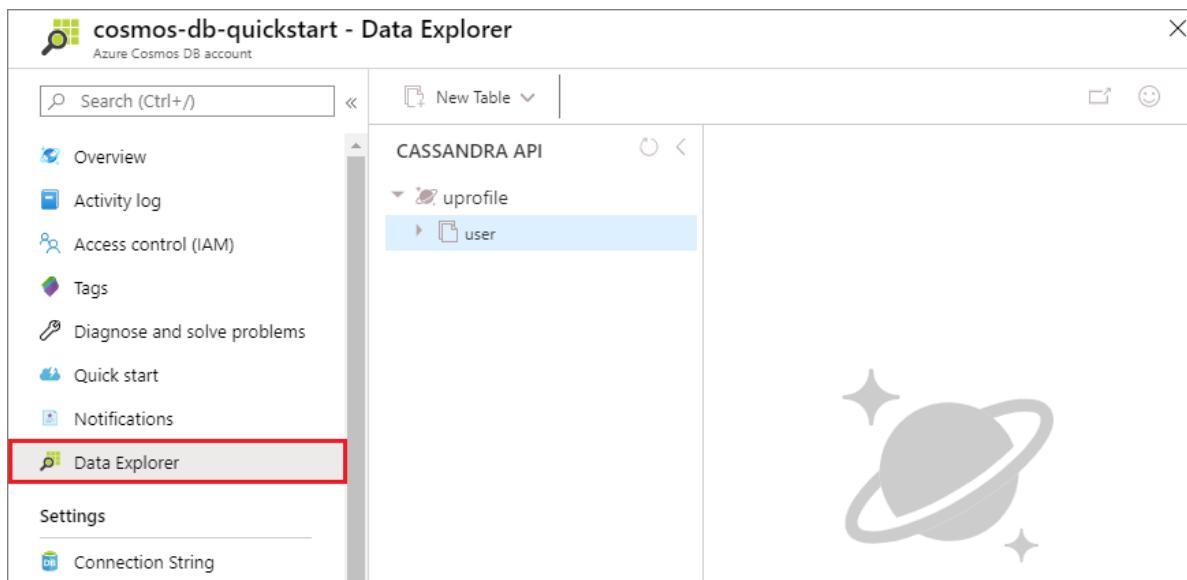
```
cd azure-cosmos-db-cassandra-nodejs-getting-started
```

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.

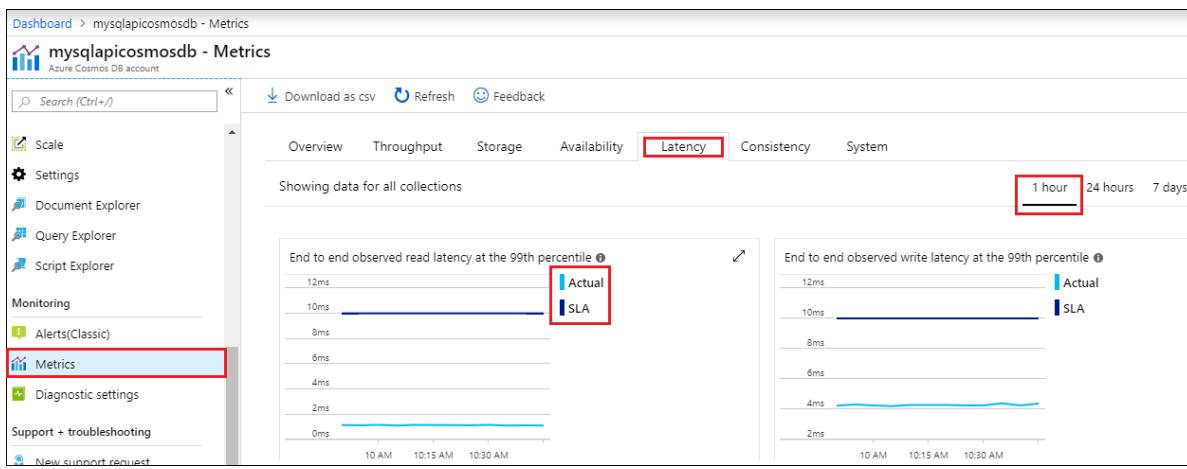


## Review SLAs in the Azure portal

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.



- 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:

- In the Azure portal Search bar, search for and select **Resource groups**.
- From the list, select the resource group you created for this quickstart.

- On the resource group **Overview** page, select **Delete resource group**.

- 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

2/11/2020 • 6 minutes to read • [Edit Online](#)

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.

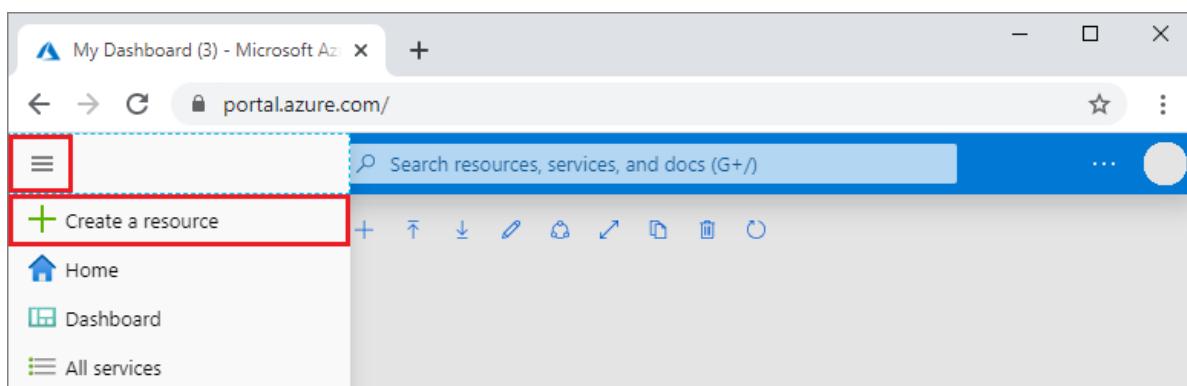
## Prerequisites

- 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**.

New

Search the Marketplace

Azure Marketplace [See all](#)

Get started  
Recently created  
AI + Machine Learning  
Analytics  
Blockchain  
Compute  
Containers  
**Databases**  
Developer Tools  
DevOps  
Identity  
Integration  
Internet of Things  
Media  
Mixed Reality

Featured [See all](#)

	Azure SQL Managed Instance <a href="#">Quickstart tutorial</a>
	SQL Database <a href="#">Quickstart tutorial</a>
	Azure Synapse Analytics (formerly SQL DW) <a href="#">Quickstart tutorial</a>
	Azure Database for MariaDB <a href="#">Learn more</a>
	Azure Database for MySQL <a href="#">Quickstart tutorial</a>
	Azure Database for PostgreSQL <a href="#">Quickstart tutorial</a>
	<b>Azure Cosmos DB</b> <a href="#">Quickstart tutorial</a>

- 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 <b>Create new</b> . 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	<p>Enter a unique name to identify your Azure Cosmos DB account. Your account URI will be <i>cassandra.cosmos.azure.com</i> appended to your unique account name.</p> <p>The account name can use only lowercase letters, numbers, and hyphens (-), and must be between 3 and 31 characters long.</p>

SETTING	VALUE	DESCRIPTION
API	Cassandra	<p>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.</p> <p>Select <b>Cassandra</b>, because in this quickstart you are creating a table that works with the Cassandra API.</p> <p><a href="#">Learn more about the Cassandra API.</a></p>
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.

Select **Review+Create**. You can skip the **Network** and **Tags** section.

**Create Azure Cosmos DB Account**

[Basics](#) [Networking](#) [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 \*

Resource Group \*  [Create new](#)

**Instance Details**

Account Name \*

API \*  [Notebooks \(preview\)](#) [Notebooks with Apache Spark \(preview\)](#) [None](#) [Sign up for Apache Spark preview](#)

Apache Spark

Location \*

Geo-Redundancy

Multi-region Writes

\*Up to 33% off multi-region writes is available to qualifying new accounts only. Accounts must be created between December 1, 2019 and February 29, 2020. 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 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 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.

```
ssl_opts = {
    'ca_certs': 'path\to\cert',
    'ssl_version': ssl.PROTOCOL_TLSv1_2
}
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_options=ssl_opts)
session = cluster.connect()
```

- The `cluster` is initialized with contactPoint information. The contactPoint is retrieved from the Azure portal.

```
cluster = Cluster([cfg.config['contactPoint']], port = cfg.config['port'], auth_provider=auth_provider)
```

- The `cluster` connects to the Azure Cosmos DB Cassandra API.

```
session = cluster.connect()
```

- A new keyspace is created.

```
session.execute('CREATE KEYSPACE IF NOT EXISTS uprofile WITH replication = {\\"class\\":\n    \\"NetworkTopologyStrategy\\\", \\"datacenter1\\\" : \\"1\\\" }')
```

- A new table is created.

```
session.execute('CREATE TABLE IF NOT EXISTS uprofile.user (user_id int PRIMARY KEY, user_name text,\nuser_bcity text);')
```

- Key/value entities are inserted.

```
insert_data = session.prepare("INSERT INTO uprofile.user (user_id, user_name , user_bcity) VALUES\n(?, ?, ?)")\nsession.execute(insert_data, [1,'Lybkov','Seattle'])\nsession.execute(insert_data, [2,'Doniv','Dubai'])\nsession.execute(insert_data, [3,'Keviv','Chennai'])\nsession.execute(insert_data, [4,'Ehtevs','Pune'])\nsession.execute(insert_data, [5,'Dnivog','Belgaum'])\n....
```

- Query to get all key values.

```
rows = session.execute('SELECT * FROM uprofile.user')
```

- Query to get a key-value.

```
rows = session.execute('SELECT * FROM uprofile.user where user_id=1')
```

## 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**.

Use the  button on the right side of the screen to copy the top value, the CONTACT POINT.

2. Open the *config.py* file.
3. 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'
```

4. 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',
```

5. Copy the PASSWORD value from the portal and paste it over <FILLME> on line 8.

Line 8 should now look similar to

```
'password' = '2Ggkr662ifxz2Mg==';
```

6. Save the *config.py* file.

## Use the X509 certificate

1. Download the Baltimore CyberTrust Root certificate locally from <https://cacert.omniroute.com/bc2025.crt>. Rename the file using the file extension .cer.

The certificate has serial number 02:00:00:b9 and SHA1 fingerprint  
d4:00:20: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:

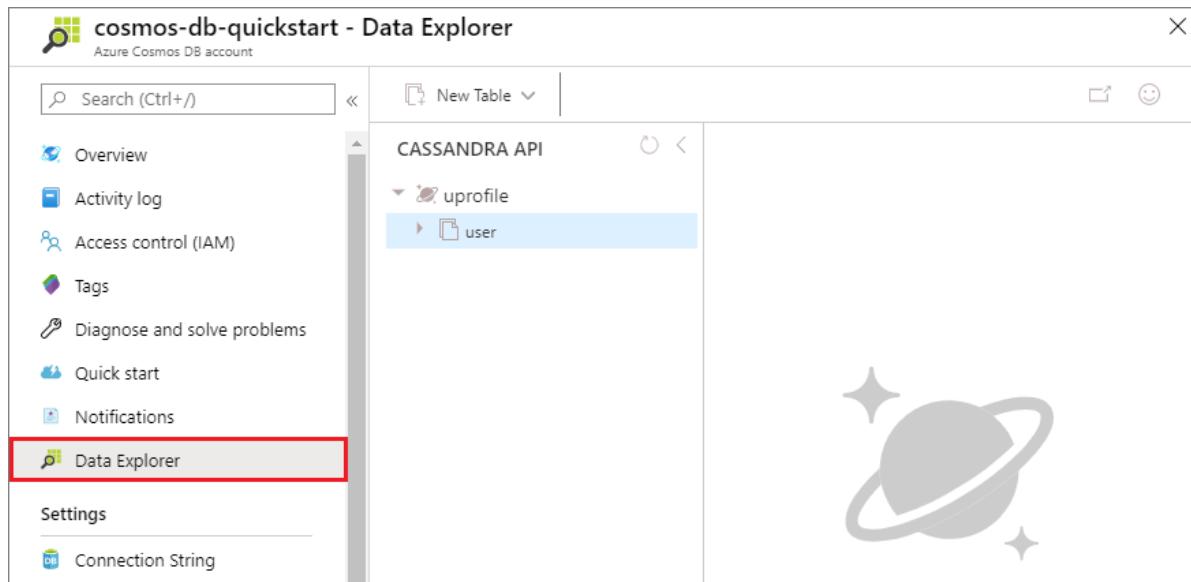
```
python pyquickstart.py
```

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.

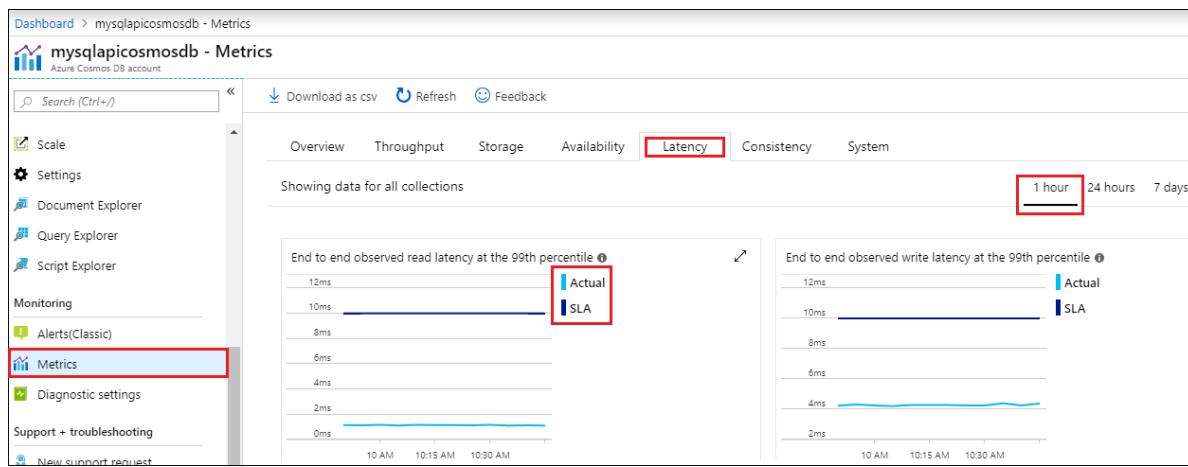


## Review SLAs in the Azure portal

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.

This screenshot shows the Azure portal interface. On the left, the 'Resource groups' blade lists several resource groups, with 'myResourceGroup' highlighted and a red box around it. On the right, the 'myResourceGroup' details page is open, showing its subscription information (Contoso Subscription), tags, and settings. A red box highlights the 'myResourceGroup' name in the list.

3. On the resource group **Overview** page, select **Delete resource group**.

This screenshot shows the 'Delete resource group' confirmation dialog. It asks if you're sure you want to delete 'myResourceGroup'. It includes a warning message about the irreversibility of the action, a field to enter the resource group name ('myResourceGroup'), and a list of affected resources (one Azure Cosmos DB account named 'mysqlapicosmosdb'). A red box highlights the 'Delete' button at the bottom of the dialog.

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](#)

# Tutorial: Create a Cassandra API account in Azure Cosmos DB by using a Java application to store key/value data

12/13/2019 • 5 minutes to read • [Edit Online](#)

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

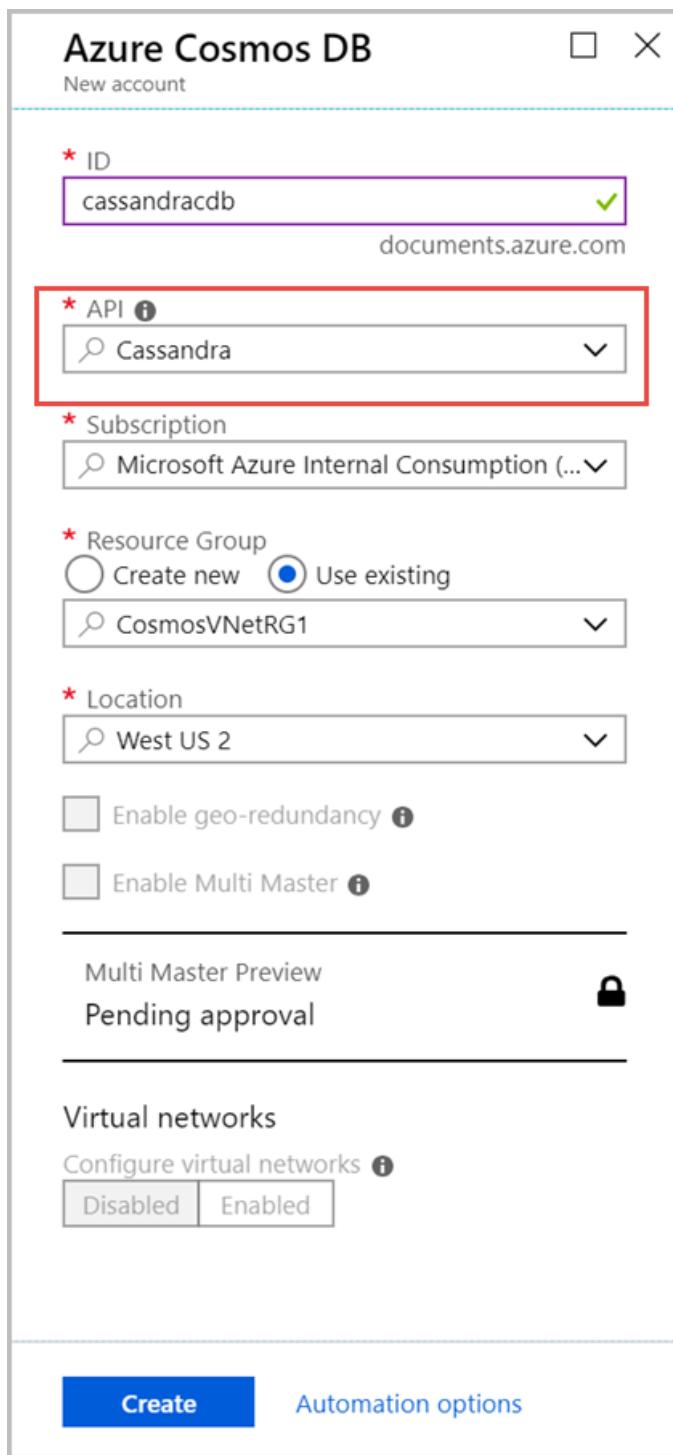
- If you don't have an Azure subscription, create a [free account](#) before you begin.
- Get the latest version of [Java Development Kit \(JDK\)](#).
- [Download](#) and [install](#) the [Maven](#) binary archive.
  - On Ubuntu, you can run `apt-get install maven` to install Maven.

## 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	<p>Enter a unique name to identify this Azure Cosmos account.</p> <p>Because <code>cassandra.cosmosdb.azure.com</code> is appended to the ID that you provide to create your contact point, use a unique but identifiable ID.</p>

SETTING	SUGGESTED VALUE	DESCRIPTION
API	Cassandra	The API determines the type of account to create. Select <b>Cassandra</b> , 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 <b>Create New</b> , 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

12/13/2019 • 2 minutes to read • [Edit Online](#)

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 PreparedStatement 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

1/4/2019 • 2 minutes to read • [Edit Online](#)

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

12/13/2019 • 4 minutes to read • [Edit Online](#)

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 SSL:** Azure Cosmos DB has strict security requirements and standards. Be sure to enable SSL when you interact with your account. When you use CQL with SSH, you have an option to provide SSL 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](#)

# Change feed in the Azure Cosmos DB API for Cassandra

12/2/2019 • 2 minutes to read • [Edit Online](#)

[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 is guaranteed 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 [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.

```

//set initial start time for pulling the change feed
DateTime timeBegin = DateTime.UtcNow;

//initialise variable to store the continuation token
byte[] pageState = null;
while (true)
{
    try
    {

        //Return the latest change for all rows in 'user' table
        IStatement changeFeedQueryStatement = new SimpleStatement(
            $"SELECT * FROM uprofile.user where COSMOS_CHANGEFEED_START_TIME() = '{timeBegin.ToString("yyyy-MM-ddTHH:mm:ss.ffffZ", CultureInfo.InvariantCulture)}'");

        if (pageState != null)
        {
            changeFeedQueryStatement = changeFeedQueryStatement.SetPagingState(pageState);
        }
        Console.WriteLine("getting records from change feed at last page state....");
        RowSet rowSet = session.Execute(changeFeedQueryStatement);

        //store the continuation token here
        pageState = rowSet.PagingState;

        List<Row> rowList = rowSet.ToList();
        if (rowList.Count != 0)
        {
            for (int i = 0; i < rowList.Count; i++)
            {
                string value = rowList[i].GetValue<string>("user_name");
                int key = rowList[i].GetValue<int>("user_id");
                // do something with the data - e.g. compute, forward to another event, function, etc.
                // here, we just print the user name field
                Console.WriteLine("user_name: " + value);
            }
        }
        else
        {
            Console.WriteLine("zero documents read");
        }
    }
    catch (Exception e)
    {
        Console.WriteLine("Exception " + e);
    }
}

```

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"

```

//Return the latest change for all row in 'user' table where user_id = 1
IStatement changeFeedQueryStatement = new SimpleStatement(
    $"SELECT * FROM uprofile.user where user_id = 1 AND COSMOS_CHANGEFEED_START_TIME() = '{timeBegin.ToString("yyyy-MM-ddTHH:mm:ss.ffffZ", CultureInfo.InvariantCulture)}'");

```

## 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](#)

# Connect to Azure Cosmos DB Cassandra API from Spark

9/3/2019 • 4 minutes to read • [Edit Online](#)

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
---------------	---------------	-------------

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. $10 \times n$ 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", "60000000")
```

## 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

1/4/2019 • 2 minutes to read • [Edit Online](#)

This article details how to work with 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

1/10/2020 • 3 minutes to read • [Edit Online](#)

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

2/26/2020 • 2 minutes to read • [Edit Online](#)

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

```
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 , 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

12/13/2019 • 2 minutes to read • [Edit Online](#)

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

1/4/2019 • 2 minutes to read • [Edit Online](#)

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

Currently predicate pushdown is not supported, the samples below reflect client-side filtering.

```

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")
  .filter("book_pub_year > 1891")
//.filter("book_name IN ('A sign of four','A study in scarlet')")
//.filter("book_name='A sign of four' OR book_name='A study in scarlet'")
//.filter("book_author='Arthur Conan Doyle' AND book_pub_year=1890")
//.filter("book_pub_year=1903")

readBooksDF.printSchema
readBooksDF.explain
readBooksDF.show

```

# 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:

- [Up insert operations](#)
- [Delete operations](#)
- [Aggregation operations](#)
- [Table copy operations](#)

# Upsert data into Azure Cosmos DB Cassandra API from Spark

1/4/2019 • 2 minutes to read • [Edit Online](#)

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

12/13/2019 • 5 minutes to read • [Edit Online](#)

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

4/2/2019 • 3 minutes to read • [Edit Online](#)

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 serialized 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 serialized 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 serialized 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

12/13/2019 • 2 minutes to read • [Edit Online](#)

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

2/24/2020 • 4 minutes to read • [Edit Online](#)

This article describes how to perform different operations to automate management of your Azure Cosmos DB accounts, databases and containers using Azure Resource Manager templates. This article has examples for SQL 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.

## Create Azure Cosmos account, keyspace and table

Create Azure Cosmos DB resources using an Azure Resource Manager template. This template will create an Azure Cosmos account for Cassandra API with two tables that share 400 RU/s throughput at the keyspace-level. Copy the template and deploy as shown below or visit [Azure Quickstart Gallery](#) and deploy from the Azure portal. You can also download the template to your local computer or create a new template and specify the local path with the `--template-file` parameter.

### NOTE

Account names must be lowercase and 44 or fewer characters. To update RU/s, resubmit the template with updated throughput property values.

```
{  
  "$schema": "https://schema.management.azure.com/schemas/2015-01-01/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",  
      "metadata": {  
        "description": "The default consistency level for the Cosmos DB account."  
      }  
    }  
  },  
  "resources": [  
    {  
      "type": "Microsoft.DocumentDB/databaseAccounts",  
      "name": "[parameters('accountName')]",  
      "apiVersion": "2017-02-28",  
      "location": "[parameters('location')]",  
      "properties": {  
        "consistencyPolicy": {  
          "consistentPrefix": true  
        },  
        "locations": [ {  
          "name": "[parameters('primaryRegion')]",  
          "readEndpoint": true,  
          "OfferType": "Standard",  
          "OfferThroughput": 400  
        }, {  
          "name": "[parameters('secondaryRegion')]",  
          "OfferType": "Standard",  
          "OfferThroughput": 400  
        } ]  
      }  
    },  
    {  
      "type": "Microsoft.DocumentDB/databaseAccounts/apis",  
      "name": "[concat(parameters('accountName'), '/cassandra')]",  
      "apiVersion": "2017-02-28",  
      "dependsOn": [ "[resourceId('Microsoft.DocumentDB/databaseAccounts', parameters('accountName'))]" ],  
      "properties": {  
        "key": "Cassandra",  
        "keyType": "Cassandra",  
        "offerType": "Standard",  
        "throughput": 400  
      }  
    },  
    {  
      "type": "Microsoft.DocumentDB/databaseAccounts/apis/containers",  
      "name": "[concat(parameters('accountName'), '/cassandra/cassandra')]",  
      "apiVersion": "2017-02-28",  
      "dependsOn": [ "[resourceId('Microsoft.DocumentDB/databaseAccounts/apis', concat(parameters('accountName'), '/cassandra'))]" ],  
      "properties": {  
        "key": "cassandra",  
        "keyType": "Cassandra",  
        "offerType": "Standard",  
        "throughput": 400  
      }  
    },  
    {  
      "type": "Microsoft.DocumentDB/databaseAccounts/apis/containers/partitions",  
      "name": "[concat(parameters('accountName'), '/cassandra/cassandra/partition')]",  
      "apiVersion": "2017-02-28",  
      "dependsOn": [ "[resourceId('Microsoft.DocumentDB/databaseAccounts/apis/containers', concat(parameters('accountName'), '/cassandra/cassandra'))]" ],  
      "properties": {  
        "key": "partition",  
        "keyType": "Cassandra",  
        "offerType": "Standard",  
        "throughput": 400  
      }  
    }  
  ]  
}
```

```
"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."
  }
},
"multipleWriteLocations": {
  "type": "bool",
  "defaultValue": false,
  "allowedValues": [ true, false ],
  "metadata": {
    "description": "Enable multi-master to make all regions writable."
  }
},
"automaticFailover": {
  "type": "bool",
  "defaultValue": false,
  "allowedValues": [ true, false ],
  "metadata": {
    "description": "Enable automatic failover for regions. Ignored when Multi-Master is enabled"
  }
},
"keyspaceName": {
  "type": "string",
  "metadata": {
    "description": "The name for the Cassandra Keyspace"
  }
},
"table1Name": {
  "type": "string",
  "metadata": {
    "description": "The name for the first Cassandra table"
  }
},
"table2Name": {
  "type": "string",
  "metadata": {
    "description": "The name for the second Cassandra table"
  }
},
"throughput": {
  "type": "int",
  "defaultValue": 400,
  "minValue": 400,
  "maxValue": 1000000,
  "metadata": {
    "description": "The throughput for both Cassandra tables"
  }
}
},
```

```

"variables": {
  "accountName": "[toLowerCase(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": "2019-08-01",
    "location": "[parameters('location')]",
    "kind": "GlobalDocumentDB",
    "properties": {
      "capabilities": [{ "name": "EnableCassandra" }],
      "consistencyPolicy": "[variables('consistencyPolicy')[parameters('defaultConsistencyLevel')]]",
      "locations": "[variables('locations')]",
      "databaseAccountOfferType": "Standard",
      "enableAutomaticFailover": "[parameters('automaticFailover')]",
      "enableMultipleWriteLocations": "[parameters('multipleWriteLocations')]"
    }
  },
  {
    "type": "Microsoft.DocumentDB/databaseAccounts/cassandraKeyspaces",
    "name": "[concat(variables('accountName'), '/', parameters('keyspaceName'))]",
    "apiVersion": "2019-08-01",
    "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('table1Name'))]",
    "apiVersion": "2019-08-01",
    "dependsOn": [ "[resourceId('Microsoft.DocumentDB/databaseAccounts/cassandraKeyspaces', "

```

```

variables('accountName'), parameters('keyspaceName'))]] ],
  "properties":
  {
    "resource":{
      "id": "[parameters('table1Name')]",
      "schema": {
        "columns": [
          { "name": "userid", "type": "uuid" },
          { "name": "posted_month", "type": "int" },
          { "name": "posted_time", "type": "uuid" },
          { "name": "body", "type": "text" },
          { "name": "posted_by", "type": "text" }
        ],
        "partitionKeys": [
          { "name": "userid" },
          { "name": "posted_month" },
          { "name": "posted_time" }
        ]
      },
      "options": { "throughput": "[parameters('throughput')]" }
    }
  }
},
{
  "type": "Microsoft.DocumentDb/databaseAccounts/cassandraKeyspaces/tables",
  "name": "[concat(variables('accountName'), '/', parameters('keyspaceName'), '/', parameters('table2Name'))]",
  "apiVersion": "2019-08-01",
  "dependsOn": [ "[resourceId('Microsoft.DocumentDB/databaseAccounts/cassandraKeyspaces', variables('accountName'), parameters('keyspaceName'))]" ],
  "properties":
  {
    "resource":{
      "id": "[parameters('table2Name')]",
      "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')]" }
    }
  }
}
]
}

```

## Deploy with the Azure CLI

To deploy the Azure Resource Manager template using the Azure CLI, **Copy** the script and select **Try it** to open Azure Cloud Shell. To paste the script, right-click the shell, and then select **Paste**:

```
read -p 'Enter the Resource Group name: ' resourceGroupName
read -p 'Enter the location (i.e. westus2): ' location
read -p 'Enter the account name: ' accountName
read -p 'Enter the primary region (i.e. westus2): ' primaryRegion
read -p 'Enter the secondary region (i.e. eastus2): ' secondaryRegion
read -p 'Enter the keyset name: ' keysetName
read -p 'Enter the first table name: ' table1Name
read -p 'Enter the second table name: ' table2Name

az group create --name $resourceGroupName --location $location
az group deployment create --resource-group $resourceGroupName \
    --template-uri https://raw.githubusercontent.com/azure/azure-quickstart-templates/master/101-cosmosdb-
cassandra/azuredeploy.json \
    --parameters accountName=$accountName primaryRegion=$primaryRegion secondaryRegion=$secondaryRegion
keysetName=$keysetName \
    table1Name=$table1Name table2Name=$table2Name

az cosmosdb show --resource-group $resourceGroupName --name accountName --output tsv
```

The `az cosmosdb show` command shows the newly created Azure Cosmos account after it has been provisioned. If you choose to use a locally installed version of the Azure CLI instead of using Cloud Shell, see the [Azure CLI](#) article.

## 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

12/5/2019 • 2 minutes to read • [Edit Online](#)

The following table includes links to sample Azure PowerShell scripts for Azure Cosmos DB for Cassandra API.

<a href="#">Create an account, keyspace and table</a>	Creates an Azure Cosmos account, keyspace and table.
<a href="#">List or get keyspaces or tables</a>	List or get keyspaces or tables.
<a href="#">Get RU/s</a>	Get RU/s for a keyspace or table.
<a href="#">Update RU/s</a>	Update RU/s for a keyspace or table.
<a href="#">Update an account or add a region</a>	Add a region to a Cosmos account. Can also be used to modify other account properties but these must be separate from changes to regions.
<a href="#">Change failover priority or trigger failover</a>	Change the regional failover priority of an Azure Cosmos account or trigger a manual failover.
<a href="#">Account keys or connection strings</a>	Get primary and secondary keys, connection strings or regenerate an account key of an Azure Cosmos account.
<a href="#">Create a Cosmos Account with IP Firewall</a>	Create an Azure Cosmos account with IP Firewall enabled.

# Azure CLI samples for Azure Cosmos DB Cassandra API

9/25/2019 • 2 minutes to read • [Edit Online](#)

The following table includes links to sample Azure CLI scripts for Azure Cosmos DB Cassandra API. Reference pages for all Azure Cosmos DB CLI commands are available in the [Azure CLI Reference](#). All Azure Cosmos DB CLI script samples can be found in the [Azure Cosmos DB CLI GitHub Repository](#).

<a href="#">Create an Azure Cosmos account, keyspace and table</a>	Creates an Azure Cosmos DB account, keyspace, and table for Cassandra API.
<a href="#">Change throughput</a>	Update RU/s on a keyspace and table.
<a href="#">Add or failover regions</a>	Add a region, change failover priority, trigger a manual failover.
<a href="#">Account keys and connection strings</a>	List account keys, read-only keys, regenerate keys and list connection strings.
<a href="#">Secure with IP firewall</a>	Create a Cosmos account with IP firewall configured.
<a href="#">Secure new account with service endpoints</a>	Create a Cosmos account and secure with service-endpoints.
<a href="#">Secure existing account with service endpoints</a>	Update a Cosmos account to secure with service-endpoints when the subnet is eventually configured.

# Azure Cosmos DB's API for MongoDB

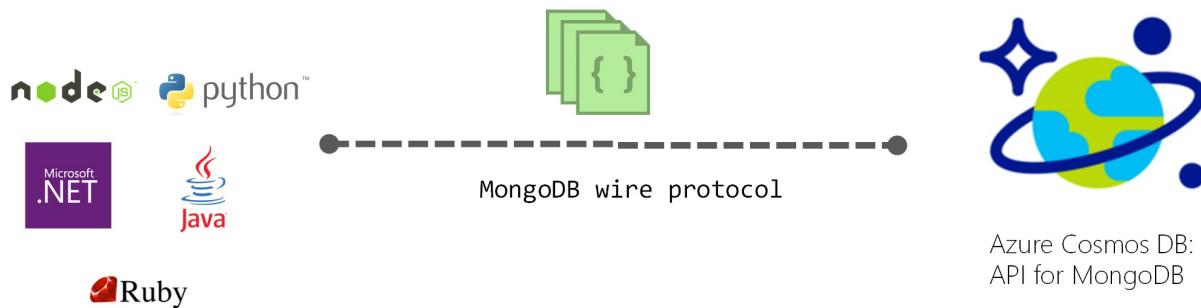
10/21/2019 • 2 minutes to read • [Edit Online](#)

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. By default, you can interact with Cosmos DB using SQL API. Additionally, the 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.

## Wire protocol compatibility

Azure Cosmos DB implements wire protocols of common NoSQL databases including Cassandra, MongoDB, Gremlin, and Azure Tables Storage. By providing a native implementation of the wire protocols directly and efficiently inside Cosmos DB, it allows existing client SDKs, drivers, and tools of the NoSQL databases to interact with Cosmos DB transparently. Cosmos DB does not use any source code of the databases for providing wire-compatible APIs for any of the NoSQL databases.

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-master 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](#).

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

10/21/2019 • 7 minutes to read • [Edit Online](#)

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.

## 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	
\$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
<code>\$all</code>	<code>{ "Location.coordinates": { \$all: [-121.758, 46.87] } }</code>	
<code>\$elemMatch</code>	<code>{ "Location.coordinates": { \$elemMatch: { \$lt: 0 } } }</code>	
<code>\$size</code>	<code>{ "Location.coordinates": { \$size: 2 } }</code>	
<code>\$comment</code>	<code>{ "Location.coordinates": { \$elemMatch: { \$lt: 0 } }, \$comment: "Negative values"}</code>	
<code>\$text</code>		Not supported. Use <code>\$regex</code> 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
<code>cursor.sort()</code>	<code>cursor.sort({ "Elevation": -1 })</code>	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 role based access control (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.

# Azure Cosmos DB's API for MongoDB (3.6 version): supported features and syntax

2/18/2020 • 7 minutes to read • [Edit Online](#)

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
parallelCollectionScan	Yes

COMMAND	SUPPORTED
resetError	No
update	Yes
<a href="#">Change streams</a>	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
killOp	No

COMMAND	SUPPORTED
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	No
explain: executionStats	No
features	No
hostInfo	No
listDatabases	Yes
listCommands	No
profiler	No
serverStatus	No
top	No
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	No
\$bucket	No
\$bucketAuto	No
\$sortByCount	Yes

COMMAND	SUPPORTED
\$addFields	Yes
\$replaceRoot	Yes
\$count	Yes
\$currentOp	No
\$listLocalSessions	No
\$listSessions	No
\$graphLookup	No

## 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

COMMAND	SUPPORTED
\$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
\$trunc	Yes

## String expressions

COMMAND	SUPPORTED
\$concat	Yes

COMMAND	SUPPORTED
\$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
\$objectToArray	Yes
\$range	Yes
\$reverseArray	Yes
\$reduce	Yes

COMMAND	SUPPORTED
\$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
\$dayOfMonth	Yes
\$dayOfWeek	Yes
\$year	Yes
\$month	Yes

COMMAND	SUPPORTED
\$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

COMMAND	SUPPORTED
\$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
Timestamp	Yes
64-bit Integer (long)	Yes
MinKey	Yes
MaxKey	Yes

COMMAND	SUPPORTED
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
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

COMMAND	SUPPORTED
\$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

COMMAND	SUPPORTED
\$[]	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
\$near	Yes
\$nearSphere	Yes

OPERATOR	SUPPORTED
\$geometry	Yes
\$minDistance	Yes
\$maxDistance	Yes
\$center	Yes
\$centerSphere	Yes
\$box	Yes
\$polygon	Yes

## 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
cursor maxScan()	Yes
cursor maxTimeMS()	Yes

COMMAND	SUPPORTED
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()	Nes
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 custom indexes in Cosmos DB by using the `createIndex` command, including the 'unique' constraint.

## 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 role-based access control (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.

# Quickstart: Build a .NET web app using Azure Cosmos DB's API for MongoDB

10/21/2019 • 5 minutes to read • [Edit Online](#)

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 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

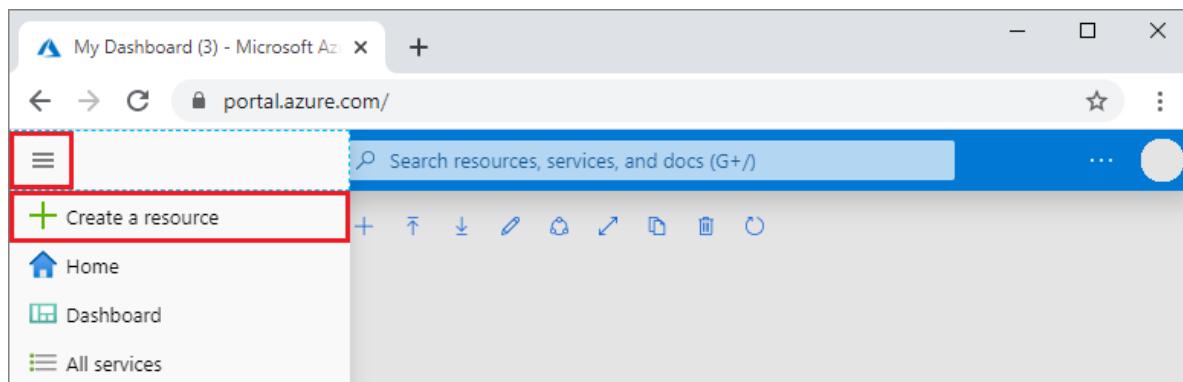
To run the sample, you'll need [Visual Studio](#) and a valid Azure Cosmos DB account.

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**.

New

Search the Marketplace

Azure Marketplace [See all](#)

Get started  
Recently created  
AI + Machine Learning  
Analytics  
Blockchain  
Compute  
Containers  
**Databases**  
Developer Tools  
DevOps  
Identity  
Integration  
Internet of Things  
Media  
Mixed Reality

Featured [See all](#)

 Azure SQL Managed Instance  
[Quickstart tutorial](#)

 SQL Database  
[Quickstart tutorial](#)

 Azure Synapse Analytics (formerly SQL DW)  
[Quickstart tutorial](#)

 Azure Database for MariaDB  
[Learn more](#)

 Azure Database for MySQL  
[Quickstart tutorial](#)

 Azure Database for PostgreSQL  
[Quickstart tutorial](#)

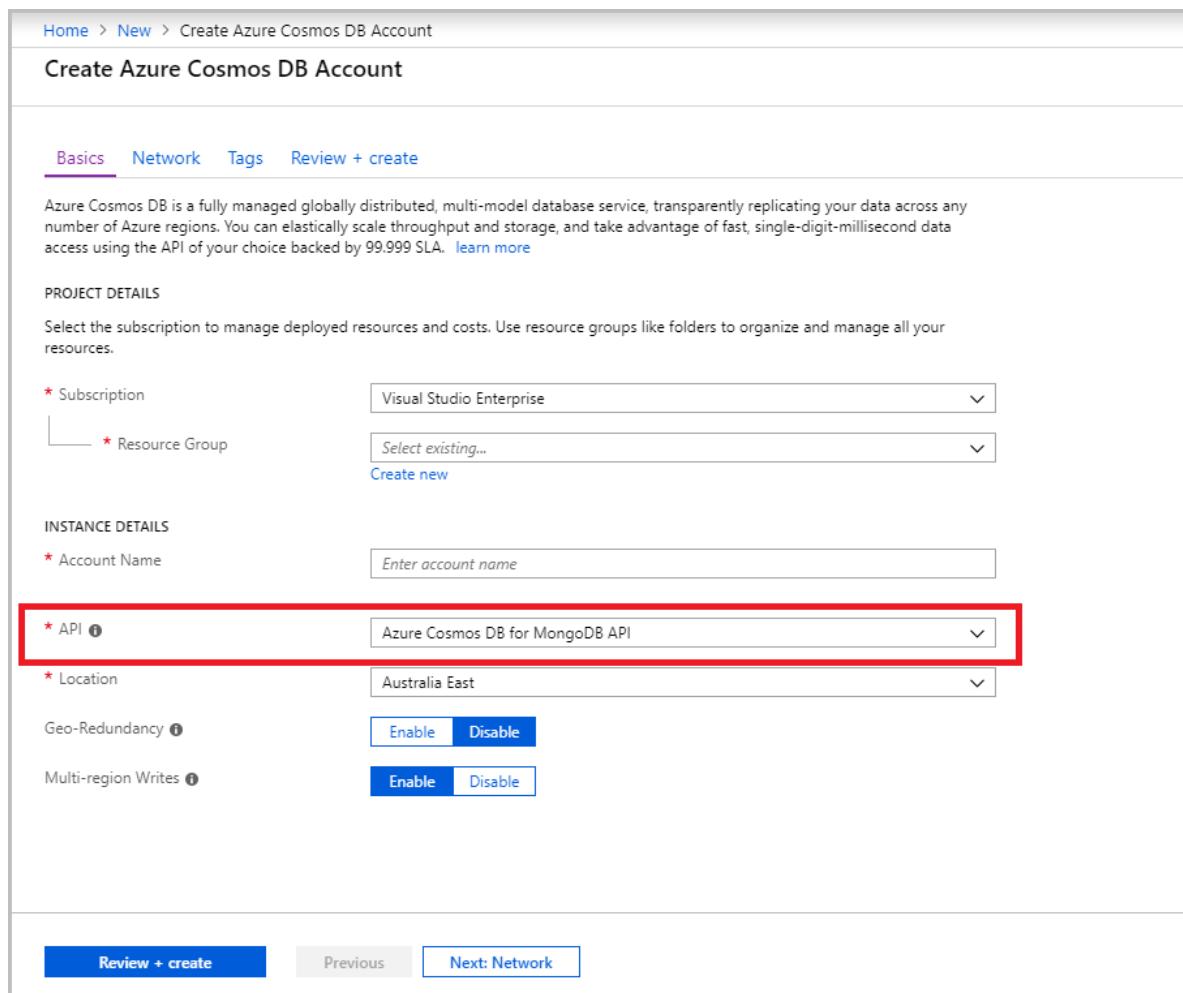
 Azure Cosmos DB  
[Quickstart tutorial](#)

- 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 <b>Create new</b> . 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	<p>Enter a unique name to identify your Azure Cosmos DB account. Your account URI will be <i>mongo.cosmos.azure.com</i> appended to your unique account name.</p> <p>The account name can use only lowercase letters, numbers, and hyphens (-), and must be between 3 and 31 characters long.</p>

SETTING	VALUE	DESCRIPTION
API	Azure Cosmos DB for Mongo DB API	<p>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.</p> <p>Select <b>Azure Cosmos DB for Mongo DB API</b> because in this quickstart you are creating a collection that works with MongoDB.</p> <p><a href="#">Learn more about Azure Cosmos DB for MongoDB API.</a></p>
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.

Select **Review+Create**. You can skip the **Network** and **Tags** section.



Home > New > Create Azure Cosmos DB Account

## Create Azure Cosmos DB Account

[Basics](#) [Network](#) [Tags](#) [Review + create](#)

Azure Cosmos DB is a fully managed globally distributed, multi-model database service, transparently replicating your data across any number of Azure regions. You can elastically scale throughput and storage, and take advantage of fast, single-digit-millisecond data access using the API of your choice backed by 99.99% SLA. [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: Select existing... or Create new

**INSTANCE DETAILS**

\* Account Name: Enter account name

\* API: Azure Cosmos DB for MongoDB API (highlighted with a red box)

\* Location: Australia East

Geo-Redundancy: Enable or Disable

Multi-region Writes: Enable or Disable

[Review + create](#) [Previous](#) [Next: Network](#)

- 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.

Home > myaccount - Quick start

## myaccount - Quick start

Azure Cosmos DB account

Search (Ctrl+ /)

Overview

Activity log

Access control (IAM)

Tags

Diagnose and solve problems

Quick start

Notifications

Data Explorer

Settings

Connection String

Preview Features

Replicate data globally

Default consistency

Firewall and virtual networks

Locks

Automation script

Collections

Browse

Scale

Monitoring

Alerts(Classic)

Metrics

Congratulations! Your Azure Cosmos DB for MongoDB API account is ready.

Now, let's connect your existing MongoDB app to it:

Choose a platform

.NET Node.js MongoDB Shell Java Python Others

**1 Connect your existing MongoDB .NET app**

You can use your existing MongoDB .NET driver to work with Azure Cosmos DB. Make sure to enable SSL. Here is an example:

```
string connectionString =
    @"mongodb://rnagpal-mongo:4IropKPu9DmlqUw0iQBxLTCftOufYwfe1m41SB7Xu7033QYq3VyRkCq5jNT2RNhCcx
MongoClientSettings settings = MongoClientSettings.FromUrl(
    new MongoUrl(connectionString)
);
settings.SslSettings =
    new SslSettings() { EnabledSslProtocols = SslProtocols.Tls12 };
var mongoClient = new MongoClient(settings);
```

PRIMARY CONNECTION STRING

```
mongodb://rnagpal-mongo:4IropKPu9DmlqUw0iQBxLTCftOufYwfe1m41SB7Xu7033QYq3VyRkCq5jNT2RNhCcxF...
```

For more details on configuring .NET driver to use SSL, follow [this article](#).

Questions? [Contact us](#)

**2 Learn More**

Code Samples

Migrating to Azure Cosmos DB

Documentation

Using Robomongo

Using MongoChef

Pricing

Forum

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.

```
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-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

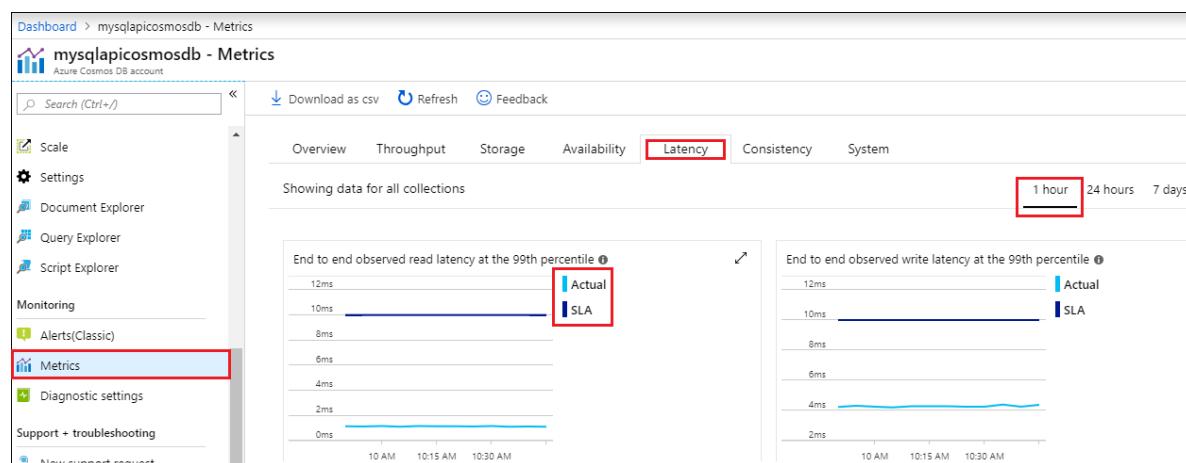
1. In Visual Studio, right-click on the project in **Solution Explorer** and then click **Manage NuGet Packages**.
2. In the NuGet **Browse** box, type *MongoDB.Driver*.
3. From the results, install the **MongoDB.Driver** library. This installs the MongoDB.Driver package as well as all dependencies.
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.

The screenshot shows the Azure portal's Resource groups blade. On the left, there's a list of resource groups: 'myResourceGroupA', 'DefaultResourceGroup', 'DefaultResourceGroupA', and 'myResourceGroup'. The 'myResourceGroup' item is selected and highlighted with a red box. On the right, the 'Overview' tab is selected in the navigation menu. The main content area displays subscription information: 'Subscription (change) : Contoso Subscription', 'Subscription ID : 12345678-abcd-abcd-000000000000', and 'Tags (change) : Click here to add tags'. Below this is a search bar and a filter dropdown. At the bottom, it says '1 items' and has a checkbox for 'Show hidden types'.

3. On the resource group **Overview** page, select **Delete resource group**.

The screenshot shows a confirmation dialog titled 'Are you sure you want to delete "myResourceGroup"?'. It contains a warning message: 'Warning! Deleting the "myResourceGroup" resource group is irreversible. The action you're about to take can't be undone. Going further will delete this resource group and all the resources it contains.' Below this is a text input field with placeholder text 'Please enter "myresourcegroup" to confirm delete.' The input field contains the text 'myResourceGroup'. On the right, it says 'TYPE THE RESOURCE GROUP NAME:' and shows a preview of the affected resources: 'mySqlApicoCosmosDb' (Azure Cosmos DB account, West US). At the bottom are two buttons: 'Delete' (highlighted with a red box) and 'Cancel'.

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

2/6/2020 • 6 minutes to read • [Edit Online](#)

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

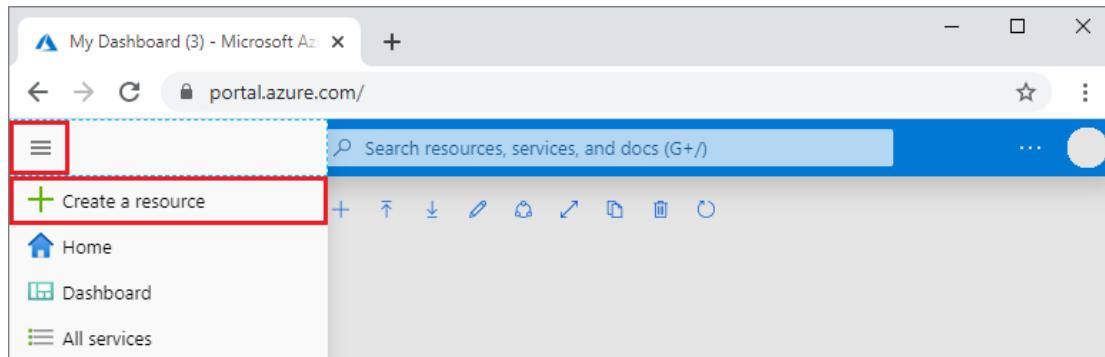
```
.mongodb://localhost:C2y6yDjf5/R+ob0N8A7Cgv30VRDJlWEHLM+4QDU5DE2nQ9nDuVTqobD4b8mGyPMbIZnqyMsEcaGQy67XIw/Jw==@localhost:10255/admin?ssl=true
```

- [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**.

The screenshot shows the Azure Marketplace interface with a search bar at the top. Below it, there are two tabs: 'Azure Marketplace' and 'See all'. Under 'Azure Marketplace', there is a list of categories: Get started, Recently created, AI + Machine Learning, Analytics, Blockchain, Compute, Containers, Databases (which is highlighted with a blue box), Developer Tools, DevOps, Identity, Integration, Internet of Things, Media, and Mixed Reality. To the right of this list, there is a 'Featured' section with several items, each with an icon and a title. One item, 'Azure Cosmos DB', has a red box drawn around its icon and title.

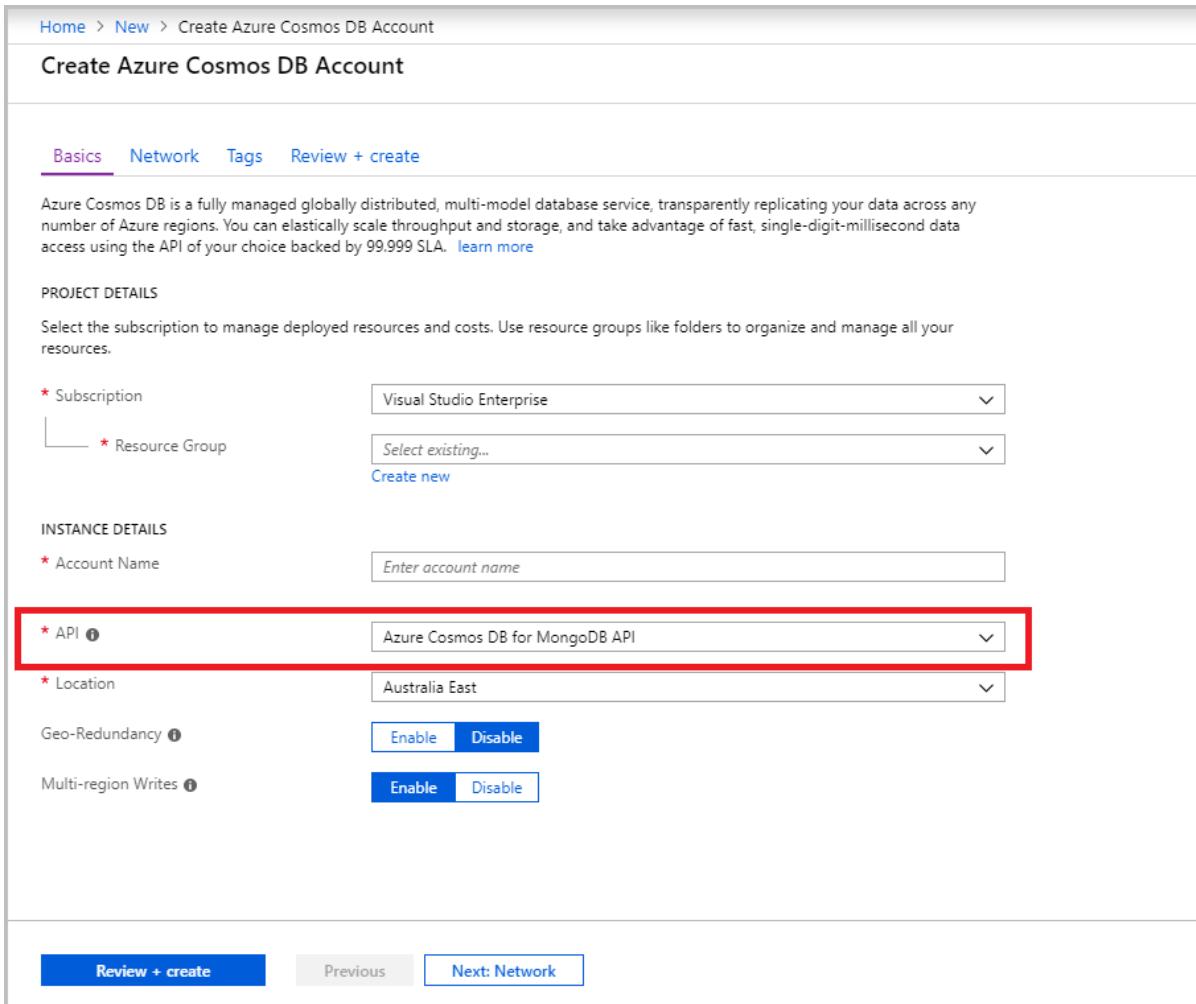
Setting	Value	Description
Subscription	Your subscription	Select the Azure subscription that you want to use for this Azure Cosmos DB account.
Resource Group	Create new Then enter the same name as Account Name	Select <b>Create new</b> . Then enter a new resource group name for your account. For simplicity, 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 <i>mongo.cosmos.azure.com</i> 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.

- 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 <b>Create new</b> . 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 <i>mongo.cosmos.azure.com</i> 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	<p>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.</p> <p>Select <b>Azure Cosmos DB for Mongo DB API</b> because in this quickstart you are creating a collection that works with MongoDB.</p> <p><a href="#">Learn more about Azure Cosmos DB for MongoDB API.</a></p>
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.

Select **Review+Create**. You can skip the **Network** and **Tags** section.



Home > New > Create Azure Cosmos DB Account

## Create Azure Cosmos DB Account

[Basics](#) [Network](#) [Tags](#) [Review + create](#)

Azure Cosmos DB is a fully managed globally distributed, multi-model database service, transparently replicating your data across any number of Azure regions. You can elastically scale throughput and storage, and take advantage of fast, single-digit-millisecond data access using the API of your choice backed by 99.999 SLA. [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: Select existing... or Create new

**INSTANCE DETAILS**

\* Account Name: Enter account name

\* API: Azure Cosmos DB for MongoDB API (highlighted with a red box)

\* Location: Australia East

Geo-Redundancy: Enable or Disable

Multi-region Writes: Enable or Disable

[Review + create](#) [Previous](#) [Next: Network](#)

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.



Search (Ctrl+ /) «

- [Overview](#)
- [Activity log](#)
- [Access control \(IAM\)](#)
- [Tags](#)
- [Diagnose and solve problems](#)
- [Quick start](#)
- [Notifications](#)
- [Data Explorer](#)

---

Settings

- [Connection String](#)
- [Preview Features](#)
- [Replicate data globally](#)
- [Default consistency](#)
- [Firewall and virtual networks](#)
- [Locks](#)
- [Automation script](#)

---

Collections

- [Browse](#)
- [Scale](#)

---

Monitoring

- [Alerts \(Classic\)](#)
- [Metrics](#)

Congratulations! Your Azure Cosmos DB for MongoDB API account is ready.

Now, let's connect your existing MongoDB app to it:

#### Choose a platform



Others

## 1 Connect your existing MongoDB .NET app

You can use your existing MongoDB .NET driver to work with Azure Cosmos DB. Make sure to enable SSL. Here is an example:

```
string connectionString =
    @"mongodb://rnagpal-mongo:4IropKPu9DmlqUw0iQBxLTCFtOufYwfe1m4ISB7Xu7033QYq3VyRkCq5jNT2RNhCcxF";
MongoClientSettings settings = MongoClientSettings.FromUrl(
    new MongoUrl(connectionString));
settings.SslSettings =
    new SslSettings() { EnabledSslProtocols = SslProtocols.Tls12 };
var mongoClient = new MongoClient(settings);
```

### PRIMARY CONNECTION STRING

```
mongodb://rnagpal-mongo:4IropKPu9DmlqUw0iQBxLTCFtOufYwfe1m4ISB7Xu7033QYq3VyRkCq5jNT2RNhCcxF...
```



For more details on configuring .NET driver to use SSL, follow [this article](#).

Questions? [Contact us](#)

## 2 Learn More

[Code Samples](#)

[Migrating to Azure Cosmos DB](#)

[Documentation](#)

[Using Robomongo](#)

[Using MongoChef](#)

[Pricing](#)

[Forum](#)

## 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 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.

The screenshot shows the Azure Cosmos DB Data Explorer interface. On the left, there's a sidebar with various options like Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Quick start, Notifications, and Data Explorer. The Data Explorer option is highlighted with a red box. At the top, there's a search bar and a 'New Container' button, which is also highlighted with a red box. The main area is titled 'SQL API' and contains a form for adding a container. It includes fields for 'Database id' (set to 'Tasks'), 'Throughput' (set to 400), 'Container id' (set to 'Items'), and 'Partition key' (/category). There are also sections for 'Tasks' and 'Autopilot (preview) / Manual'. A note at the bottom says 'Estimated spend (USD): \$<price>hourly / \$<price>daily / \$<price>monthly (2 regions, 400RU/s, \$<price>/RU)'.

2. In the **Add container** page, enter the settings for the new container.

SETTING	SUGGESTED VALUE	DESCRIPTION
<b>Database ID</b>	Tasks	Enter <i>Tasks</i> 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 <b>Provision database throughput</b> 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.
<b>Throughput</b>	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.
<b>Container ID</b>	Items	Enter <i>Items</i> as the name for your new container. Container IDs have the same character requirements as database names.
<b>Partition key</b>	/category	The sample described in this article uses <i>/category</i> 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.

## 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.

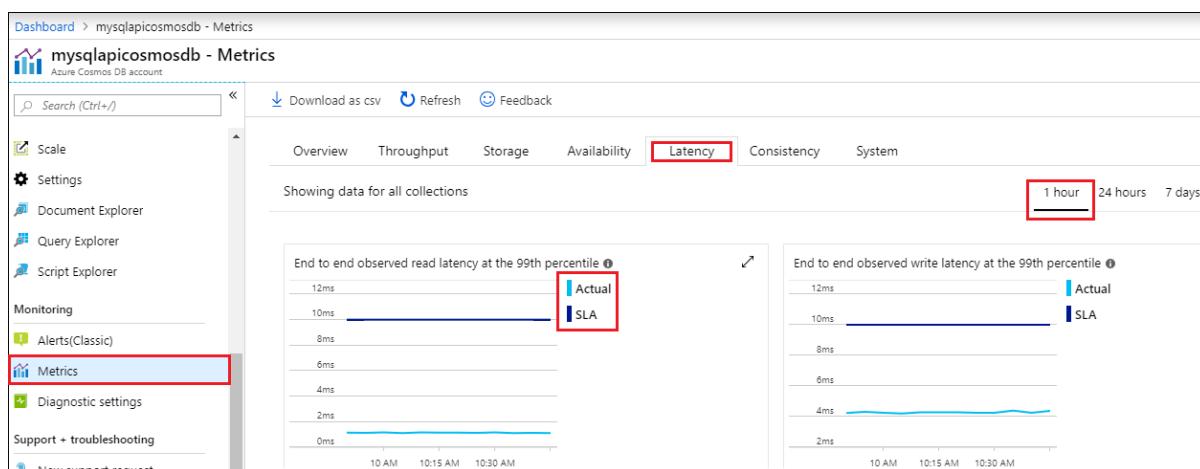
You can now use [Robomongo](#) / [Studio 3T](#) to query, modify, and work with this new data.

## 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.

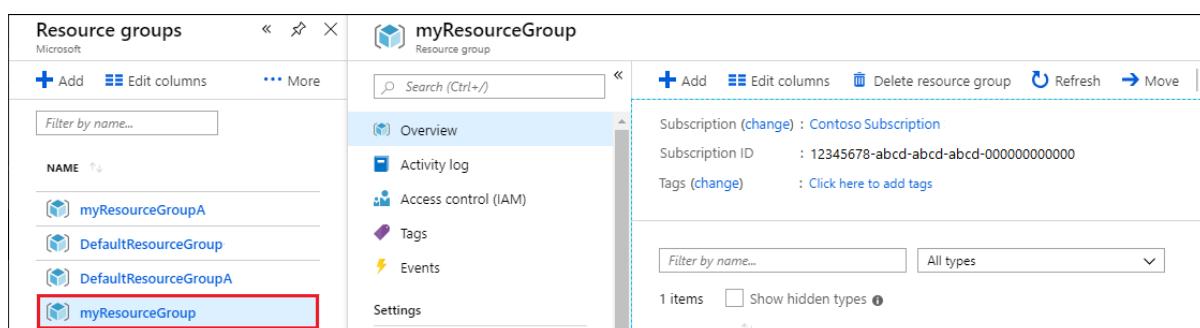


3. Review the metrics on the other tabs.

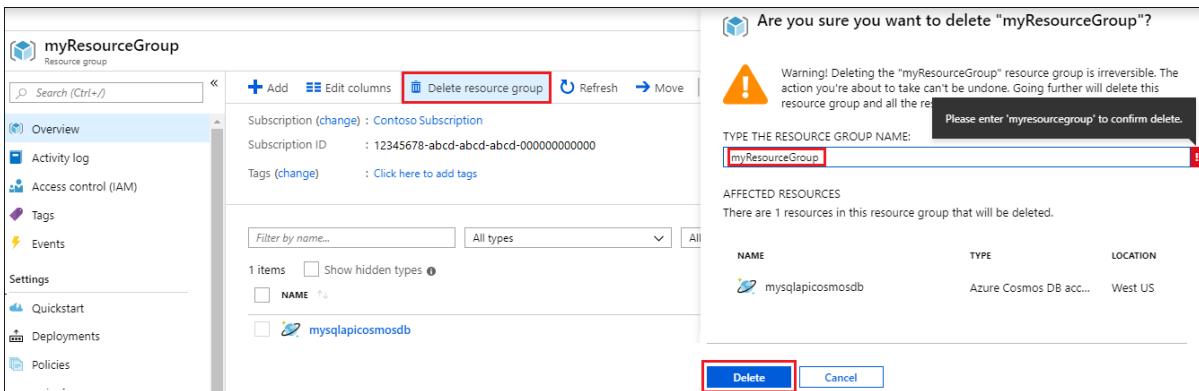
## Clean up resources

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 MongoDB 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

2/6/2020 • 7 minutes to read • [Edit Online](#)

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

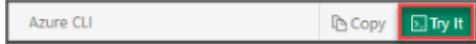
- An Azure account with an active subscription. [Create one for free](#). Or [try Azure Cosmos DB for free](#) without an Azure subscription. You can also use the [Azure Cosmos DB Emulator](#) with the connection string  

```
.mongodb://localhost:C2y6yDjf5/R+ob0N8A7Cgv30VRDJWEHLM+4QDU5DE2nQ9nDuVTqobD4b8mGyPMbIZnqyMsEcaGQy67XIw/Jw==@localhost:10255/admin?ssl=true
```
- [Node.js](#), and a working knowledge of Node.js.
- [Git](#).
- If you don't want to use Azure Cloud Shell, [Azure CLI 2.0+](#).

## 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 <b>Try It</b> in the upper-right corner of a code block. Selecting <b>Try It</b> doesn't automatically copy the code to Cloud Shell.	
Go to <a href="https://shell.azure.com">https://shell.azure.com</a> , or select the <b>Launch Cloud Shell</b> button to open Cloud Shell in your browser.	
Select the <b>Cloud Shell</b> button on the menu bar at the upper right in the <a href="#">Azure portal</a> .	

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.

## 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 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.DocumentDB/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.

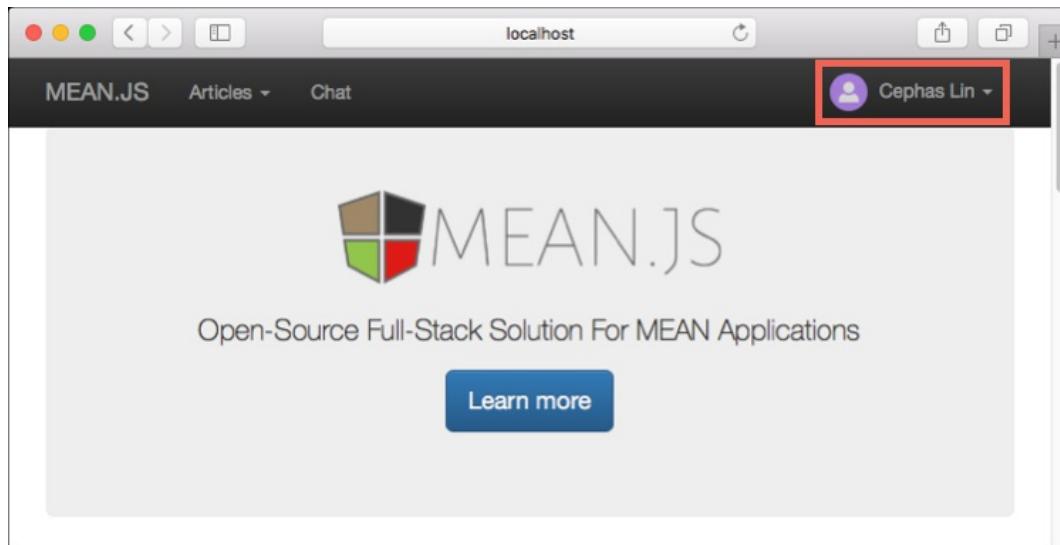
Run `npm start` again.

```
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.

id	region
1	
2	
3	

## 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 shown 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 [Cosmos DB requires SSL](#).

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.

The screenshot shows the Azure Resource Groups blade. On the left, there's a list of resource groups: 'myResourceGroupA', 'DefaultResourceGroup', 'DefaultResourceGroupA', and 'myResourceGroup'. The 'myResourceGroup' item is highlighted with a red border. On the right, the details for 'myResourceGroup' are shown under the 'Overview' tab. It includes information like 'Subscription (change) : Contoso Subscription', 'Subscription ID : 12345678-abcd-abcd-000000000000', and 'Tags (change) : Click here to add tags'. Below this, there's a list of resources: 'mysqlapiccosmosdb' (Azure Cosmos DB account, West US).

3. On the resource group **Overview** page, select **Delete resource group**.

This screenshot shows a confirmation dialog box titled 'Are you sure you want to delete "myResourceGroup"?'. It contains a warning message: 'Warning! Deleting the "myResourceGroup" resource group is irreversible. The action you're about to take can't be undone. Going further will delete this resource group and all the resources it contains.' Below this, a text input field says 'Please enter 'myresourcegroup' to confirm delete.' with 'myResourceGroup' typed in. The 'Affected Resources' section lists one resource: 'mysqlapiccosmosdb' (Azure Cosmos DB account, West US). At the bottom are 'Delete' and 'Cancel' buttons, with 'Delete' highlighted with a red box.

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

2/6/2020 • 6 minutes to read • [Edit Online](#)

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' +
'+ob0N8A7Cgv30VRDJWEHLM+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"
```

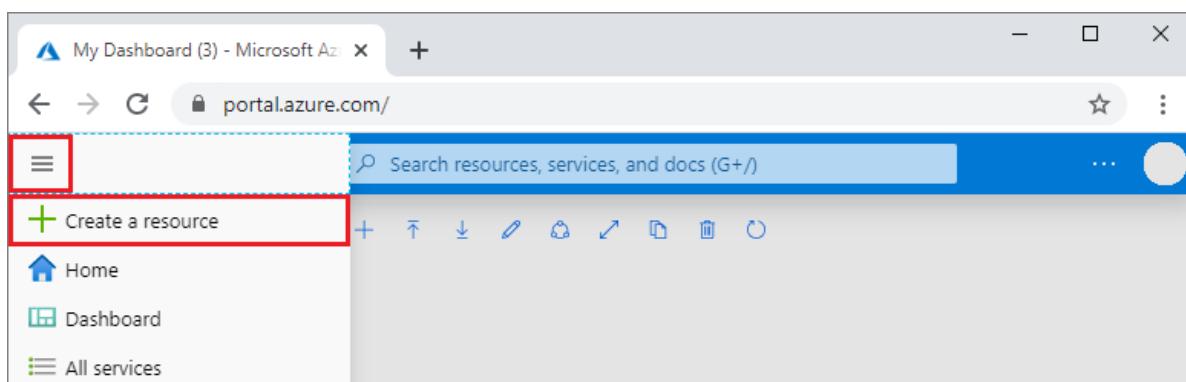
## Run the web app

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**.

New

Search the Marketplace

Azure Marketplace [See all](#)

- Get started
- Recently created
- AI + Machine Learning
- Analytics
- Blockchain
- Compute
- Containers
- Databases**
- Developer Tools
- DevOps
- Identity
- Integration
- Internet of Things
- Media
- Mixed Reality

Featured [See all](#)

-  Azure SQL Managed Instance  
[Quickstart tutorial](#)
-  SQL Database  
[Quickstart tutorial](#)
-  Azure Synapse Analytics (formerly SQL DW)  
[Quickstart tutorial](#)
-  Azure Database for MariaDB  
[Learn more](#)
-  Azure Database for MySQL  
[Quickstart tutorial](#)
-  Azure Database for PostgreSQL  
[Quickstart tutorial](#)
-  Azure Cosmos DB  
[Quickstart tutorial](#)

- 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 <b>Create new</b> . 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 <i>mongo.cosmos.azure.com</i> 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	<p>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.</p> <p>Select <b>Azure Cosmos DB for Mongo DB API</b> because in this quickstart you are creating a collection that works with MongoDB.</p> <p><a href="#">Learn more about Azure Cosmos DB for MongoDB API.</a></p>
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.

Select **Review+Create**. You can skip the **Network** and **Tags** section.

Home > New > Create Azure Cosmos DB Account

## Create Azure Cosmos DB Account

[Basics](#) [Network](#) [Tags](#) [Review + create](#)

Azure Cosmos DB is a fully managed globally distributed, multi-model database service, transparently replicating your data across any number of Azure regions. You can elastically scale throughput and storage, and take advantage of fast, single-digit-millisecond data access using the API of your choice backed by 99.99% SLA. [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: Select existing... or Create new

**INSTANCE DETAILS**

\* Account Name: Enter account name

\* API: Azure Cosmos DB for MongoDB API (highlighted with a red box)

\* Location: Australia East

Geo-Redundancy: Enable or Disable

Multi-region Writes: Enable or Disable

[Review + create](#) [Previous](#) [Next: Network](#)

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 screenshot shows the 'myaccount - Quick start' page for an Azure Cosmos DB account. The left sidebar contains navigation links for Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Quick start (which is selected), Notifications, Data Explorer, Settings (with Connection String, Preview Features, Replicate data globally, Default consistency, Firewall and virtual networks, Locks, and Automation script), Collections (with Browse and Scale), and Monitoring (with Alerts(Classic) and Metrics). The main content area displays a success message: 'Congratulations! Your Azure Cosmos DB for MongoDB API account is ready.' It also provides instructions to connect existing MongoDB apps and lists supported platforms (.NET, Node.js, MongoDB Shell, Java, Python, Others). A code sample for .NET is shown:

```

string connectionString =
    @"mongodb://rnagpal-mongo:4IropKPu9DmlqUw0iQbxLTCftOufYwfe1m4ISB7Xu7033QYq3VyRkCq5jNT2RNhCcxF";
MongoClientSettings settings = MongoClientSettings.FromUrl(
    new MongoUrl(connectionString)
);
settings.SslSettings =
    new SslSettings() { EnabledSslProtocols = SslProtocols.Tls12 };
var mongoClient = new MongoClient(settings);

```

Below the code, there's a note about PRIMARY CONNECTION STRING and a copy button. Further down, there are links for Learn More (Code Samples, Migrating to Azure Cosmos DB, Documentation, Using Robomongo, Using MongoChef, Pricing, Forum), Contact us, and a link to this article.

## 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.



#### NOTE

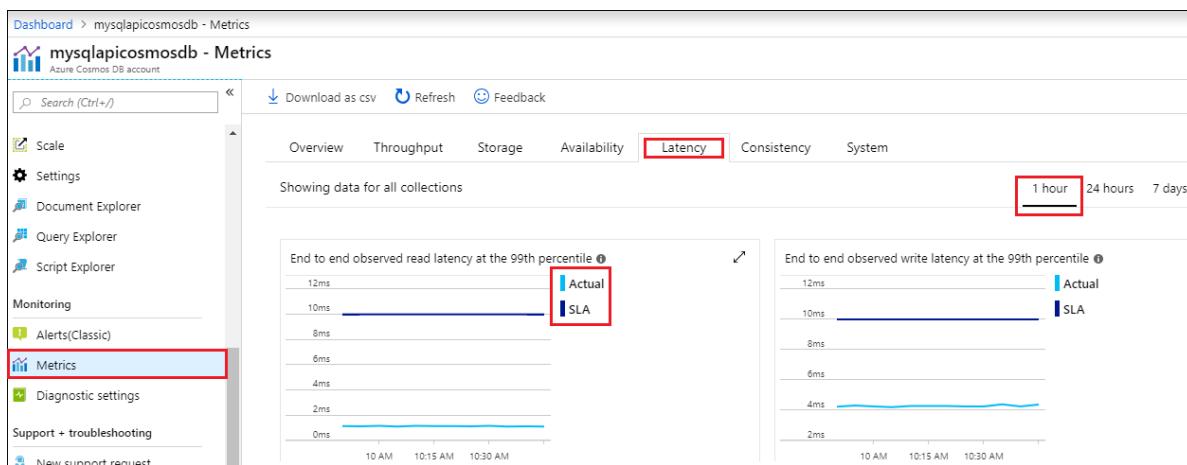
If you plan to store your code in GitHub or other source control options, please be sure to remove your connection strings from the code. They can be set with application settings for the web app instead.

## 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.

The screenshot shows the Azure Resource Groups blade. On the left, there's a list of resource groups: 'myResourceGroupA', 'DefaultResourceGroup', 'DefaultResourceGroupA', and 'myResourceGroup'. The 'myResourceGroup' item is highlighted with a red box. On the right, the details for 'myResourceGroup' are shown, including its subscription information ('Contoso Subscription'), ID ('12345678-abcd-abcd-000000000000'), and tags ('Click here to add tags'). A secondary pane shows a list of items with one item selected ('myResourceGroup').

3. On the resource group **Overview** page, select **Delete resource group**.

The screenshot shows a confirmation dialog titled 'Are you sure you want to delete "myResourceGroup"?'. It includes a warning message about the irreversibility of the action, a field to type the resource group name ('myResourceGroup'), and a table of affected resources. One resource, 'mysqlapicosmosdb', is listed with its type as 'Azure Cosmos DB acc...' and location as 'West US'. The 'Delete' button is highlighted with a red box.

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 MongoDB 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

12/13/2019 • 6 minutes to read • [Edit Online](#)

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.

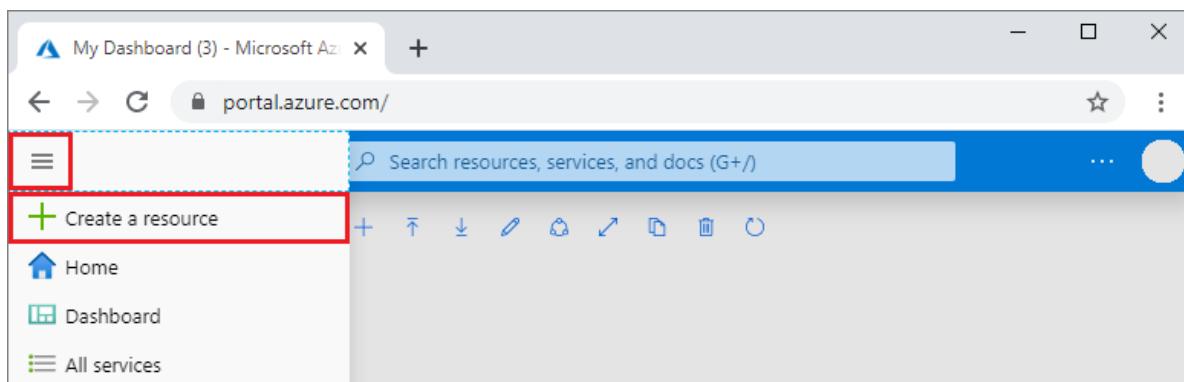
If you prefer to work on a Mac, download [Visual Studio for Mac](#) and run the setup.

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**.

New

Search the Marketplace

Azure Marketplace See all

- Get started
- Recently created
- AI + Machine Learning
- Analytics
- Blockchain
- Compute
- Containers
- Databases**
- Developer Tools
- DevOps
- Identity
- Integration
- Internet of Things
- Media
- Mixed Reality

Featured See all

-  Azure SQL Managed Instance  
[Quickstart tutorial](#)
-  SQL Database  
[Quickstart tutorial](#)
-  Azure Synapse Analytics (formerly SQL DW)  
[Quickstart tutorial](#)
-  Azure Database for MariaDB  
[Learn more](#)
-  Azure Database for MySQL  
[Quickstart tutorial](#)
-  Azure Database for PostgreSQL  
[Quickstart tutorial](#)
-  Azure Cosmos DB  
[Quickstart tutorial](#)

- 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 <b>Create new</b> . 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 <i>mongo.cosmos.azure.com</i> 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	<p>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.</p> <p>Select <b>Azure Cosmos DB for Mongo DB API</b> because in this quickstart you are creating a collection that works with MongoDB.</p> <p><a href="#">Learn more about Azure Cosmos DB for MongoDB API.</a></p>
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.

Select **Review+Create**. You can skip the **Network** and **Tags** section.

Home > New > Create Azure Cosmos DB Account

## Create Azure Cosmos DB Account

[Basics](#) [Network](#) [Tags](#) [Review + create](#)

Azure Cosmos DB is a fully managed globally distributed, multi-model database service, transparently replicating your data across any number of Azure regions. You can elastically scale throughput and storage, and take advantage of fast, single-digit-millisecond data access using the API of your choice backed by 99.999 SLA. [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: Select existing... or Create new

**INSTANCE DETAILS**

\* Account Name: Enter account name

\* API: Azure Cosmos DB for MongoDB API

\* Location: Australia East

Geo-Redundancy: Enable or Disable

Multi-region Writes: Enable or Disable

[Review + create](#) [Previous](#) [Next: Network](#)

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.

Home > myaccount - Quick start

## myaccount - Quick start

Azure Cosmos DB account

Search (Ctrl+ /)

Overview

Activity log

Access control (IAM)

Tags

Diagnose and solve problems

Quick start

Notifications

Data Explorer

Settings

Connection String

Preview Features

Replicate data globally

Default consistency

Firewall and virtual networks

Locks

Automation script

Collections

Browse

Scale

Monitoring

Alerts(Classic)

Metrics

Congratulations! Your Azure Cosmos DB for MongoDB API account is ready.

Now, let's connect your existing MongoDB app to it:

Choose a platform

.NET Node.js MongoDB Shell Java Python Others

**1 Connect your existing MongoDB .NET app**

You can use your existing MongoDB .NET driver to work with Azure Cosmos DB. Make sure to enable SSL. Here is an example:

```
string connectionString =
    @"mongodb://rnagpal-mongo:4IropKPu9DmlqUw0iQBxLTCftOufYwfe1m4ISB7Xu7033QYq3VyRkCq5jNT2RNhCcx
MongoClientSettings settings = MongoClientSettings.FromUrl(
    new MongoUrl(connectionString)
);
settings.SslSettings =
    new SslSettings() { EnabledSslProtocols = SslProtocols.Tls12 };
var mongoClient = new MongoClient(settings);
```

PRIMARY CONNECTION STRING

```
mongodb://rnagpal-mongo:4IropKPu9DmlqUw0iQBxLTCftOufYwfe1m4ISB7Xu7033QYq3VyRkCq5jNT2RNhCcxF...
```

For more details on configuring .NET driver to use SSL, follow [this article](#).

Questions? [Contact us](#)

**2 Learn More**

Code Samples

Migrating to Azure Cosmos DB

Documentation

Using Robomongo

Using MongoChef

Pricing

Forum

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.

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-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 };  
  
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.

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.

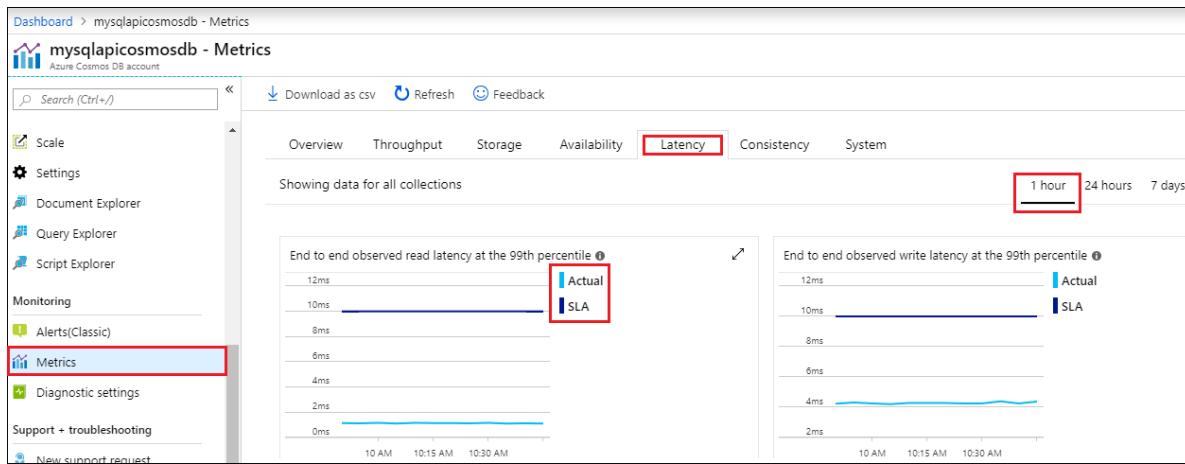
## Review SLAs in the Azure portal

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.

The screenshot shows the Azure portal's 'Resource groups' blade. On the left, there's a list of resource groups: 'myResourceGroupA', 'DefaultResourceGroup', 'DefaultResourceGroupA', and 'myResourceGroup'. The 'myResourceGroup' item is selected and highlighted with a red box. On the right, the 'Overview' page for 'myResourceGroup' is displayed, showing details like 'Subscription (change) : Contoso Subscription', 'Subscription ID : 12345678-abcd-abcd-000000000000', and 'Tags (change) : Click here to add tags'. Below this is a table with one item, 'myResourceGroup'.

3. On the resource group **Overview** page, select **Delete resource group**.

The screenshot shows a confirmation dialog box titled 'Are you sure you want to delete "myResourceGroup"?'. It contains a warning message: 'Warning! Deleting the "myResourceGroup" resource group is irreversible. The action you're about to take can't be undone. Going further will delete this resource group and all the resources it contains.' Below this is a text input field labeled 'TYPE THE RESOURCE GROUP NAME:' with the value 'myResourceGroup'. At the bottom are 'Delete' and 'Cancel' buttons, with 'Delete' highlighted with a red box.

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: Build a console app using Azure Cosmos DB's API for MongoDB and Golang SDK

12/13/2019 • 7 minutes to read • [Edit Online](#)

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 Cosmos DB.

This quickstart demonstrates how to take an existing MongoDB app written in [Golang](#) and connect it to your Cosmos database using the Azure Cosmos DB's API for MongoDB.

In other words, your Golang application only knows that it's connecting using a MongoDB client. It is transparent to the application that the data is stored in a Cosmos database.

## Prerequisites

- An Azure subscription. 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

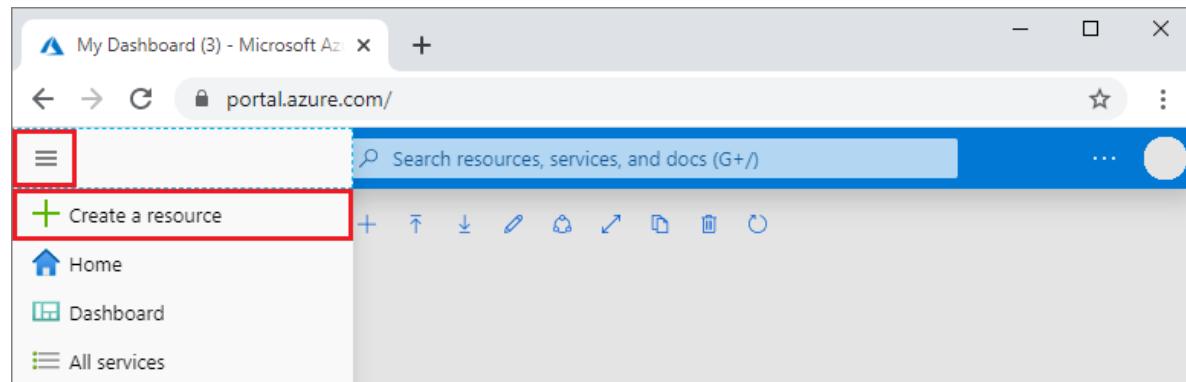
```
mongodb://localhost:C2y6yDjf5/R+ob0N8A7Cgv30VRDJIWEHLM+4QDU5DE2nQ9nDuVTqobD4b8mGgyPMbIZnqyMsEcaGQy67XIwJW==@localhost:10255/admin?ssl=true
```

- [Go](#) and a basic knowledge of the [Go](#) language.
- An IDE — [GoLand](#) by Jetbrains, [Visual Studio Code](#) by Microsoft, or [Atom](#). In this tutorial, I'm using GoLand.

## 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**.

New

Search the Marketplace

Azure Marketplace [See all](#)

Get started  
Recently created  
AI + Machine Learning  
Analytics  
Blockchain  
Compute  
Containers  
**Databases**  
Developer Tools  
DevOps  
Identity  
Integration  
Internet of Things  
Media  
Mixed Reality

Featured [See all](#)

 Azure SQL Managed Instance  
[Quickstart tutorial](#)

 SQL Database  
[Quickstart tutorial](#)

 Azure Synapse Analytics (formerly SQL DW)  
[Quickstart tutorial](#)

 Azure Database for MariaDB  
[Learn more](#)

 Azure Database for MySQL  
[Quickstart tutorial](#)

 Azure Database for PostgreSQL  
[Quickstart tutorial](#)

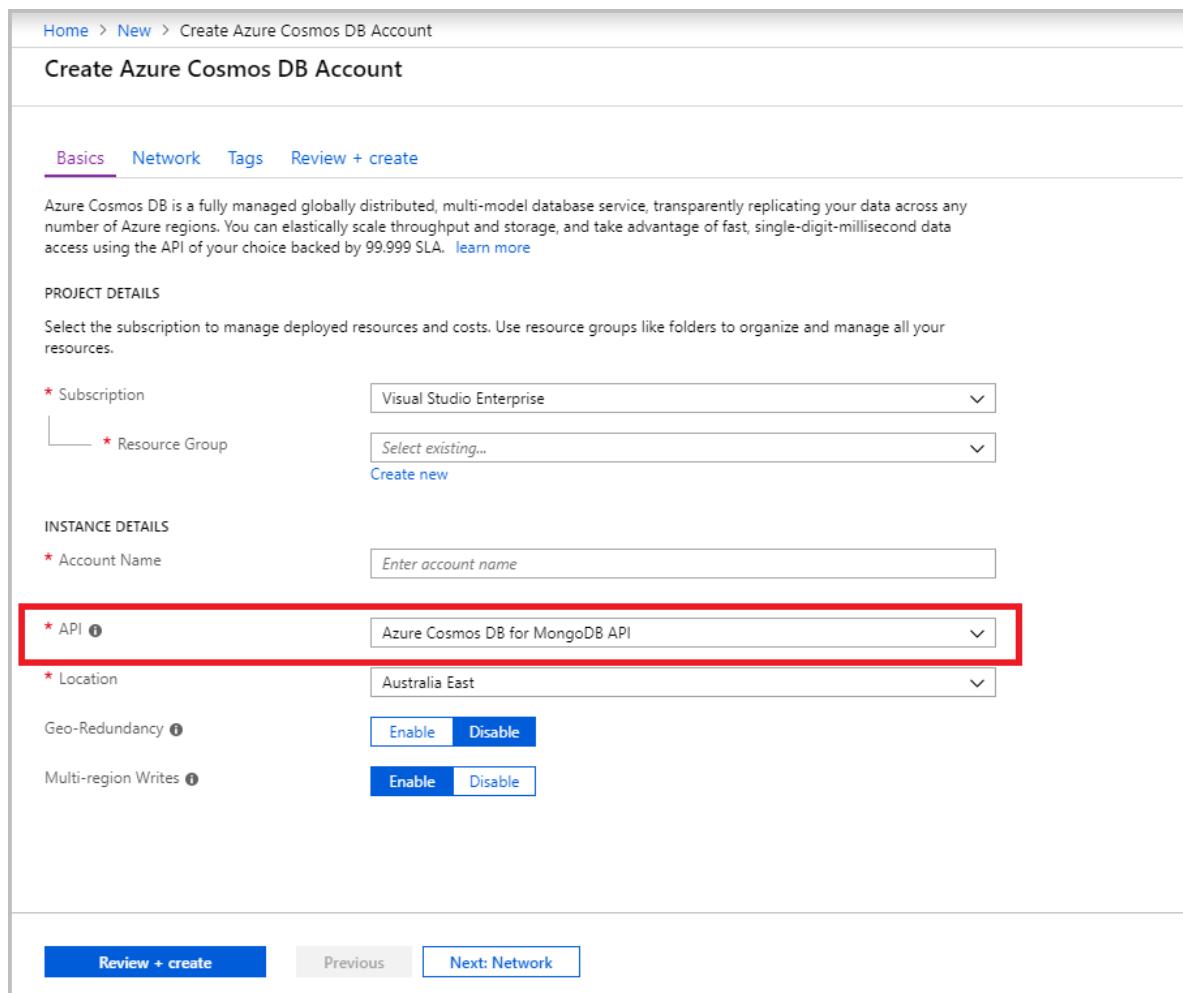
 Azure Cosmos DB  
[Quickstart tutorial](#)

- 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 <b>Create new</b> . 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	<p>Enter a unique name to identify your Azure Cosmos DB account. Your account URI will be <i>mongo.cosmos.azure.com</i> appended to your unique account name.</p> <p>The account name can use only lowercase letters, numbers, and hyphens (-), and must be between 3 and 31 characters long.</p>

SETTING	VALUE	DESCRIPTION
API	Azure Cosmos DB for Mongo DB API	<p>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.</p> <p>Select <b>Azure Cosmos DB for Mongo DB API</b> because in this quickstart you are creating a collection that works with MongoDB.</p> <p><a href="#">Learn more about Azure Cosmos DB for MongoDB API.</a></p>
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.

Select **Review+Create**. You can skip the **Network** and **Tags** section.



Home > New > Create Azure Cosmos DB Account

## Create Azure Cosmos DB Account

[Basics](#) [Network](#) [Tags](#) [Review + create](#)

Azure Cosmos DB is a fully managed globally distributed, multi-model database service, transparently replicating your data across any number of Azure regions. You can elastically scale throughput and storage, and take advantage of fast, single-digit-millisecond data access using the API of your choice backed by 99.99% SLA. [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: Select existing... or Create new

**INSTANCE DETAILS**

\* Account Name: Enter account name

\* API: Azure Cosmos DB for MongoDB API (highlighted with a red box)

\* Location: Australia East

Geo-Redundancy: Enable or Disable

Multi-region Writes: Enable or Disable

[Review + create](#) [Previous](#) [Next: Network](#)

- 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.

Congratulations! Your Azure Cosmos DB for MongoDB API account is ready.

Now, let's connect your existing MongoDB app to it:

**Choose a platform**

.NET    Node.js    MongoDB Shell    Java    Python    Others

**1 Connect your existing MongoDB .NET app**

You can use your existing MongoDB .NET driver to work with Azure Cosmos DB. Make sure to enable SSL. Here is an example:

```
string connectionString =
    @"mongodb://rnatpal-mongo:4IropKPu9DmlqUw0iQBxLTCftOufYwfe1m41SB7Xu7033QYq3VyRkCq5jNT2RNhCcx
MongoClientSettings settings = MongoClientSettings.FromUrl(
    new MongoUrl(connectionString)
);
settings.SslSettings =
    new SslSettings() { EnabledSslProtocols = SslProtocols.Tls12 };
var mongoClient = new MongoClient(settings);
```

PRIMARY CONNECTION STRING  
mongodb://rnatpal-mongo:4IropKPu9DmlqUw0iQBxLTCftOufYwfe1m41SB7Xu7033QYq3VyRkCq5jNT2RNhCcxF...

For more details on configuring .NET driver to use SSL, follow [this article](#).

Questions? [Contact us](#)

**2 Learn More**

[Code Samples](#)  
[Migrating to Azure Cosmos DB](#)  
[Documentation](#)  
[Using Robomongo](#)  
[Using MongoChef](#)  
[Pricing](#)  
[Forum](#)

## Clone the sample application

Clone the sample application and install the required packages.

1. Create a folder named `CosmosDBSample` inside the `GOROOT\src` folder, which is `C:\Go\` by default.
2. Run the following command using a git terminal window such as git bash to clone the sample repository into the `CosmosDBSample` folder.

```
git clone https://github.com/Azure-Samples/azure-cosmos-db-mongodb-golang-getting-started.git
```

3. Run the following command to get the `mgo` package.

```
go get gopkg.in/mgo.v2
```

The `mgo` driver is a [MongoDB](#) driver for the [Go language](#) that implements a rich and well tested selection of features under a very simple API following standard Go idioms.

## Update your connection string

Now go back to the Azure portal to get your connection string information and copy it into the app.

1. Click **Quick start** in the left navigation menu, and then click **Other** to view the connection string information required by the Go application.
2. In Golang, open the `main.go` file in the `GOROOT\CosmosDBSample` directory and update the following lines of code using the connection string information from the Azure portal as shown in the following

screenshot.

The Database name is the prefix of the **Host** value in the Azure portal connection string pane. For the account shown in the image below, the Database name is golang-coach.

```
Database: "The prefix of the Host value in the Azure portal",
Username: "The Username in the Azure portal",
Password: "The Password in the Azure portal",
```

The screenshot shows the Azure portal interface for a Cosmos DB account named 'cdbmongo2'. On the left, there's a sidebar with various navigation options like Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Quick start (which is highlighted with a red box), Notifications, Data Explorer, Settings, Connection String, Preview Features, and Replicate data globally. The main content area has a title 'cdbmongo2 - Quick start' and a sub-section 'Choose a platform' with tabs for .NET, Node.js, MongoDB Shell, Java, Python, and Others (which is also highlighted with a red box). Below this, there's a step-by-step guide: '1 Connect your existing MongoDB app'. It instructs users to use the host and password below to connect to their new account, emphasizing TLS 1.2. It shows fields for HOST (cdbmongo2.documents.azure.com), PORT (10255), USERNAME (cdbmongo2), and PRIMARY PASSWORD (empty). Both the HOST and USERNAME fields are highlighted with red boxes. At the bottom, there's a note about SSL configuration and a 'Contact us' link.

3. Save the main.go 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 [Run the app](#).

The following snippets are all taken from the main.go file.

### Connecting the Go app to Cosmos DB

Azure Cosmos DB's API for MongoDB supports the SSL-enabled connection. To connect, you need to define the **DialServer** function in [mgo.DialInfo](#), and make use of the [tls.Dial](#) function to perform the connection.

The following Golang code snippet connects the Go app with Azure Cosmos DB's API for MongoDB. The *DialInfo* class holds options for establishing a session.

```

// DialInfo holds options for establishing a session.
dialInfo := &mgo.DialInfo{
    Addrs:   []string{"golang-couch.documents.azure.com:10255"}, // Get HOST + PORT
    Timeout: 60 * time.Second,
    Database: "database", // It can be anything
    Username: "username", // Username
    Password: "Azure database connect password from Azure Portal", // PASSWORD
    DialServer: func(addr *mgo.ServerAddr) (net.Conn, error) {
        return tls.Dial("tcp", addr.String(), &tls.Config{})
    },
}

// Create a session which maintains a pool of socket connections
// to Cosmos database (using Azure Cosmos DB's API for MongoDB).
session, err := mgo.DialWithInfo(dialInfo)

if err != nil {
    fmt.Printf("Can't connect, go error %v\n", err)
    os.Exit(1)
}

defer session.Close()

// SetSafe changes the session safety mode.
// If the safe parameter is nil, the session is put in unsafe mode,
// and writes become fire-and-forget,
// without error checking. The unsafe mode is faster since operations won't hold on waiting for a
// confirmation.
//
session.SetSafe(&mgo.Safe{})

```

The **mgo.Dial()** method is used when there is no SSL connection. For an SSL connection, the **mgo.DialWithInfo()** method is required.

An instance of the **DialWithInfo{}** object is used to create the session object. Once the session is established, you can access the collection by using the following code snippet:

```
collection := session.DB("database").C("package")
```

## Create a document

```

// Model
type Package struct {
    Id bson.ObjectId `bson:"_id,omitempty"`
    FullName      string
    Description   string
    StarsCount    int
    ForksCount    int
    LastUpdatedBy string
}

// insert Document in collection
err = collection.Insert(&Package{
    FullName:"react",
    Description:"A framework for building native apps with React.",
    ForksCount: 11392,
    StarsCount:48794,
    LastUpdatedBy:"shergin",
})

if err != nil {
    log.Fatal("Problem inserting data: ", err)
    return
}

```

## Query or read a document

Cosmos DB supports rich queries against data stored in each collection. The following sample code shows a query that you can run against the documents in your collection.

```

// Get a Document from the collection
result := Package{}
err = collection.Find(bson.M{"fullname": "react"}).One(&result)
if err != nil {
    log.Fatal("Error finding record: ", err)
    return
}

fmt.Println("Description:", result.Description)

```

## Update a document

```

// Update a document
updateQuery := bson.M{"_id": result.Id}
change := bson.M{"$set": bson.M{"fullname": "react-native"}}
err = collection.Update(updateQuery, change)
if err != nil {
    log.Fatal("Error updating record: ", err)
    return
}

```

## Delete a document

Cosmos DB supports deletion of documents.

```
// Delete a document
query := bson.M{"_id": result.Id}
err = collection.Remove(query)
if err != nil {
    log.Fatal("Error deleting record: ", err)
    return
}
```

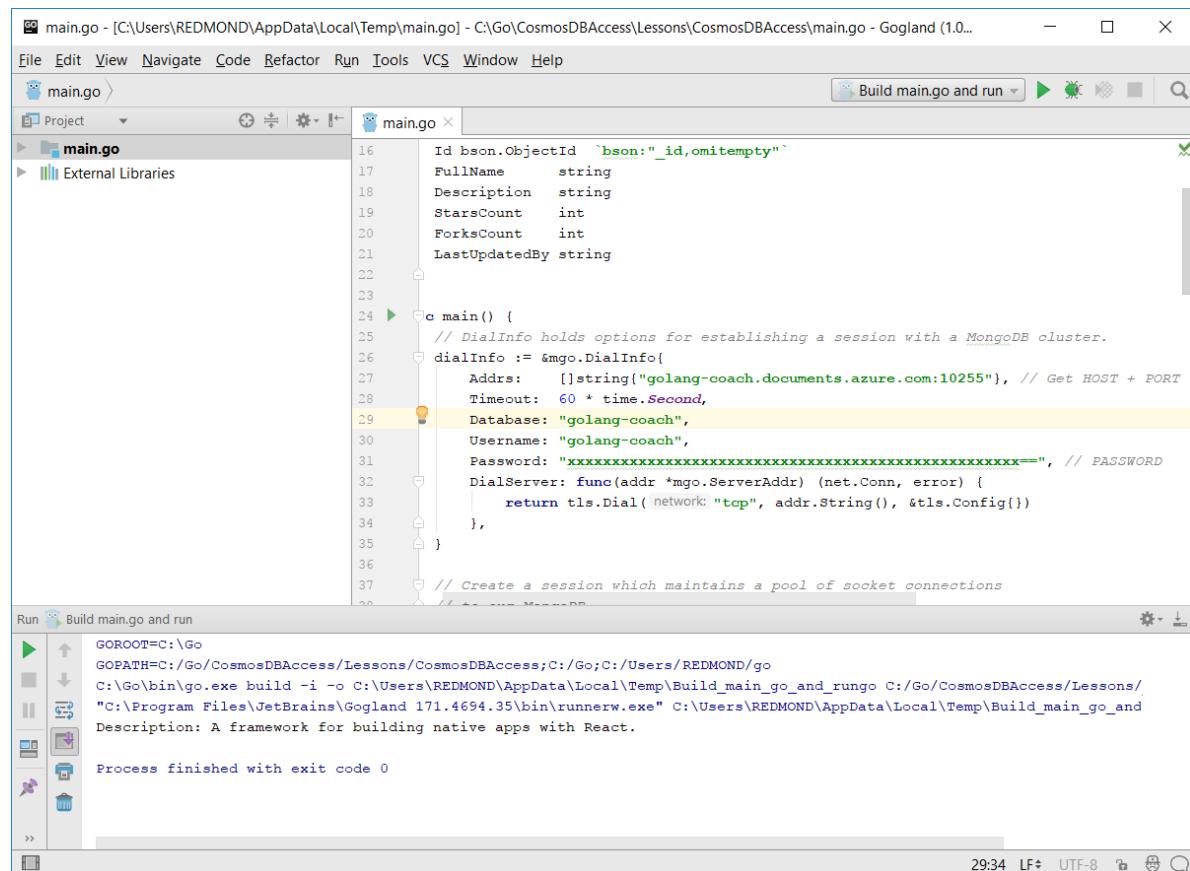
## Run the app

1. In Golang, ensure that your GOPATH (available under **File**, **Settings**, **Go**, **GOPATH**) include the location in which the gopkg was installed, which is **USERPROFILE\go** by default.
  2. Comment out the lines that delete the document, lines 103-107, so that you can see the document after running the app.
  3. In Golang, click **Run**, and then click **Run 'Build main.go and run'**.

The app finishes and displays the description of the document created in [Create a document](#).

Description: A framework for building native apps with React.

Process finished with exit code 0



## Review your document in Data Explorer

Go back to the Azure portal to see your document in Data Explorer.

1. Click **Data Explorer (Preview)** in the left navigation menu, expand **golang-coach, package**, and then click **Documents**. In the **Documents** tab, click the `id` to display the document in the right pane.

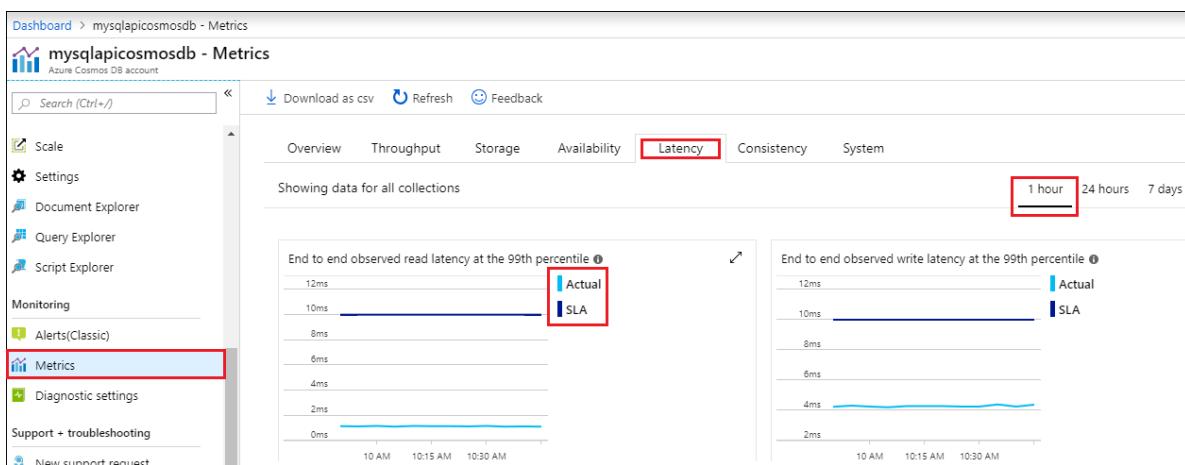
2. You can then work with the document inline and click **Update** to save it. You can also delete the document, or create new documents or queries.

## 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.

The screenshot shows the Azure portal's Resource groups blade. On the left, there's a list of resource groups: 'myResourceGroupA', 'DefaultResourceGroup', 'DefaultResourceGroupA', and 'myResourceGroup'. The 'myResourceGroup' item is selected and highlighted with a red box. On the right, the 'Overview' tab is selected in the navigation menu. The main pane displays subscription information: 'Subscription (change) : Contoso Subscription', 'Subscription ID : 12345678-abcd-abcd-000000000000', and 'Tags (change) : Click here to add tags'. Below this is a search bar and a filter dropdown. At the bottom, it says '1 items' and has a checkbox for 'Show hidden types'.

3. On the resource group **Overview** page, select **Delete resource group**.

The screenshot shows a confirmation dialog box titled 'Are you sure you want to delete "myResourceGroup"?'. It includes a warning message: 'Warning! Deleting the "myResourceGroup" resource group is irreversible. The action you're about to take can't be undone. Going further will delete this resource group and all the resources it contains.' Below the message is a text input field containing 'myResourceGroup' with a red border. At the bottom of the dialog are two buttons: 'Delete' (highlighted with a red box) and 'Cancel'.

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 and run a Golang app. You can now import additional data to your Cosmos database.

[Import MongoDB data into Azure Cosmos DB](#)

# Build an app using Node.js and Azure Cosmos DB's API for MongoDB

1/31/2019 • 2 minutes to read • [Edit Online](#)

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

12/13/2019 • 2 minutes to read • [Edit Online](#)

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 globally distributed multi-model database service. It enables you to quickly create and query document, key/value, and graph databases that benefit from the global distribution and horizontal scale capabilities at the core of Cosmos DB.

This multi-part tutorial covers the following tasks:

- [Create a Nodejs 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 Nodejs Express app.

[Create a Nodejs Express app with the Angular CLI](#)

# Create an Angular app with Azure Cosmos DB's API for MongoDB - Create a Node.js Express app

12/13/2019 • 3 minutes to read • [Edit Online](#)

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
```

- Once the command completes, change directories into the `src/client` folder.

```
cd angular-cosmosdb
```

- Then open the folder in Visual Studio Code.

```
code .
```

## Build the app using the Express framework

- In Visual Studio Code, in the **Explorer** pane, right-click the `src` folder, click **New Folder**, and name the new folder `server`.
- In the **Explorer** pane, right-click the `server` folder, click **New File**, and name the new file `index.js`.
- 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
```

- 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
- Crank's 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.

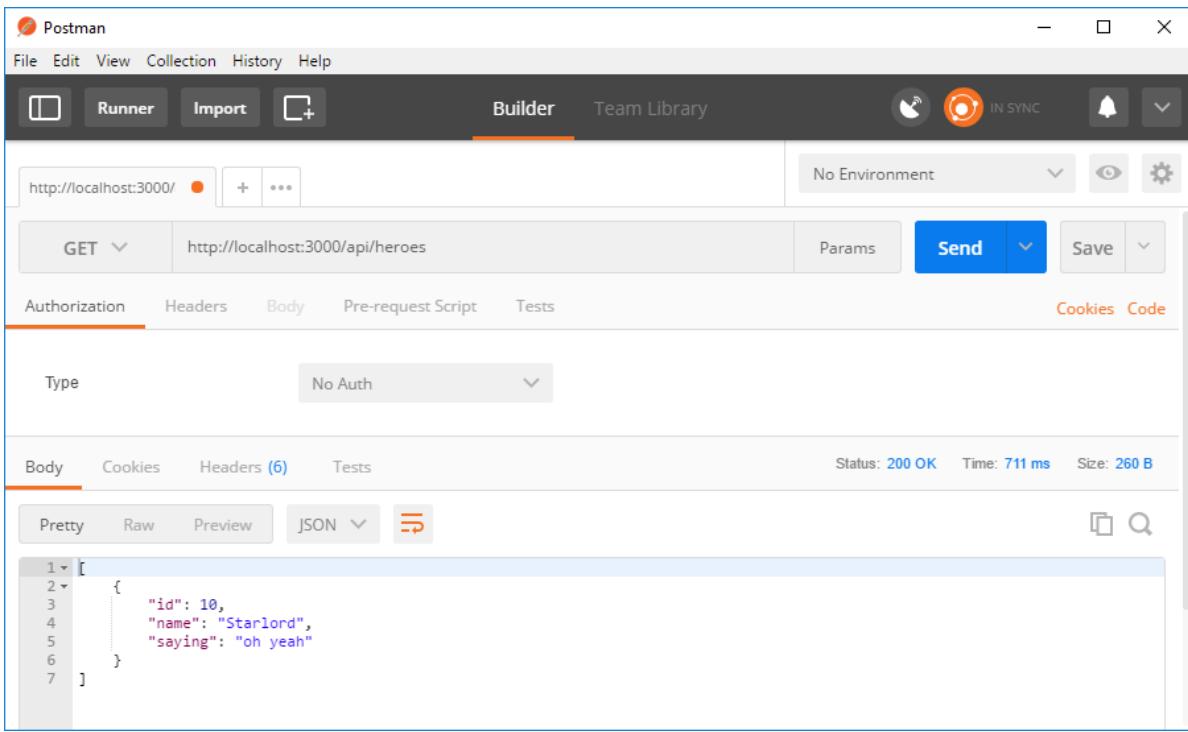
The app should run without errors.

## 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

12/13/2019 • 9 minutes to read • [Edit Online](#)

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)
```

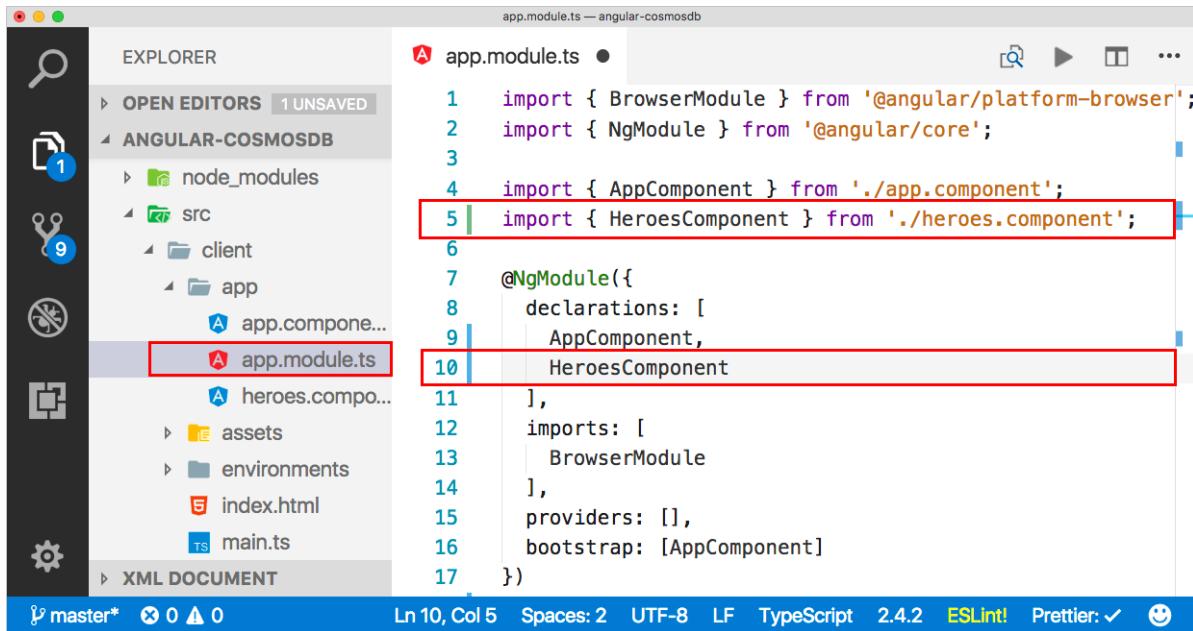
Let's take a look at the files that were created and updated.

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.



```
app.module.ts — angular-cosmosdb
1   import { BrowserModule } from '@angular/platform-browser';
2   import { NgModule } from '@angular/core';
3
4   import { AppComponent } from './app.component';
5   import { HeroesComponent } from './heroes.component';
6
7   @NgModule({
8     declarations: [
9       AppComponent,
10      HeroesComponent
11     ],
12     imports: [
13       BrowserModule
14     ],
15     providers: [],
16     bootstrap: [AppComponent]
17   })

```

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>

```

- 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;
      });
    }
  }
}

```

- 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: #cf8dc;
  }
}

.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: #cf8dc !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
```

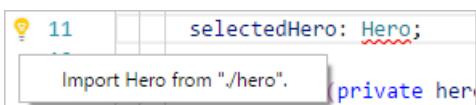
3. In the Explorer pane, open **src\app\hero.ts**. In **hero.ts**, replace the content of the file with the following code, which adds a Hero class with an ID, a name, and a saying.

```

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

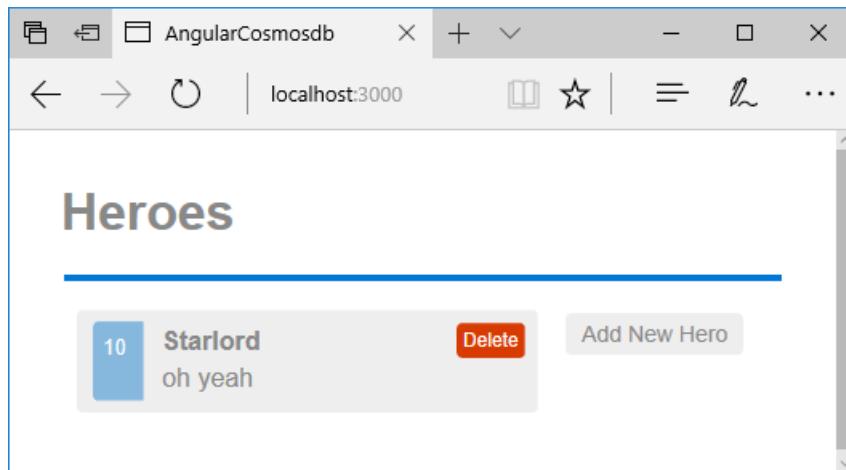
1. At the command prompt, enter the following command to build the Angular application.

```
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

12/13/2019 • 3 minutes to read • [Edit Online](#)

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

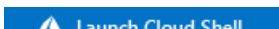
Before starting this part of the tutorial, ensure you've completed the steps in [Part 3](#) of the tutorial.

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 <b>Try It</b> in the upper-right corner of a code block. Selecting <b>Try It</b> doesn't automatically copy the code to Cloud Shell.	
Go to <a href="https://shell.azure.com">https://shell.azure.com</a> , or select the <b>Launch Cloud Shell</b> button to open Cloud Shell in your browser.	
Select the <b>Cloud Shell</b> button on the menu bar at the upper right in the <a href="#">Azure portal</a> .	

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 like 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.

The screenshot shows the Microsoft Azure portal interface. The top navigation bar includes 'File', 'Edit', 'View', 'History', 'Bookmarks', 'People', 'Window', and 'Help'. A user profile 'John' is visible on the top right. The main content area is titled 'Microsoft Azure Azure Cosmos DB > my-cosmos-heroes'. On the left, there's a sidebar with a search bar and a list of options: 'Overview' (highlighted with a red box), 'Activity log', 'Access control (IAM)', 'Tags', 'Diagnose and solve problems', 'Quick start', 'Data Explorer (Preview)', 'SETTINGS' (with 'Connection String', 'Replicate data globally', 'Default consistency', and 'Firewall'), and 'COLLECTIONS' (with 'Add Collection'). The main panel has tabs for 'Add Collection', 'Refresh', 'Move', 'Delete Account', and 'Data Explorer'. Below these tabs, it says 'Collections' and 'Regions'. The 'Regions' section features a world map with various regions marked by blue dots. A button labeled 'Data Explorer' is also present.

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.

The screenshot shows the 'Replicate data globally' page for the 'my-cosmos-heroes' Azure Cosmos DB account. The top navigation bar and user profile are the same as the previous screenshot. The main title is 'Microsoft Azure Azure Cosmos DB > my-cosmos-heroes - Replicate data globally'. The left sidebar includes 'Search (Ctrl+)', 'Diagnose and solve problems', 'Quick start', 'Data Explorer (Preview)', 'SETTINGS' (with 'Connection String', 'Replicate data globally' highlighted with a red box), 'Default consistency', 'Firewall', 'Add Azure Search', 'Locks', and 'Automation script'. The main panel has tabs for 'Save', 'Discard', 'Manual Failover', and 'Automatic Failover'. It displays a world map with regions marked by blue dots. A callout text says 'Click on a location to add or remove regions from your Azure Cosmos DB account.' and 'Each region is billable based on the throughput and storage for the account.' A 'Learn more' link is also present. The bottom of the page shows a 'WRITE REGION' button.

## 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

12/13/2019 • 6 minutes to read • [Edit Online](#)

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}/${env.databaseName}?ssl=true`;

function connect() {
  mongoose.set('debug', true);
  return mongoose.connect(mongoUri, { useNewUrlParser: 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:

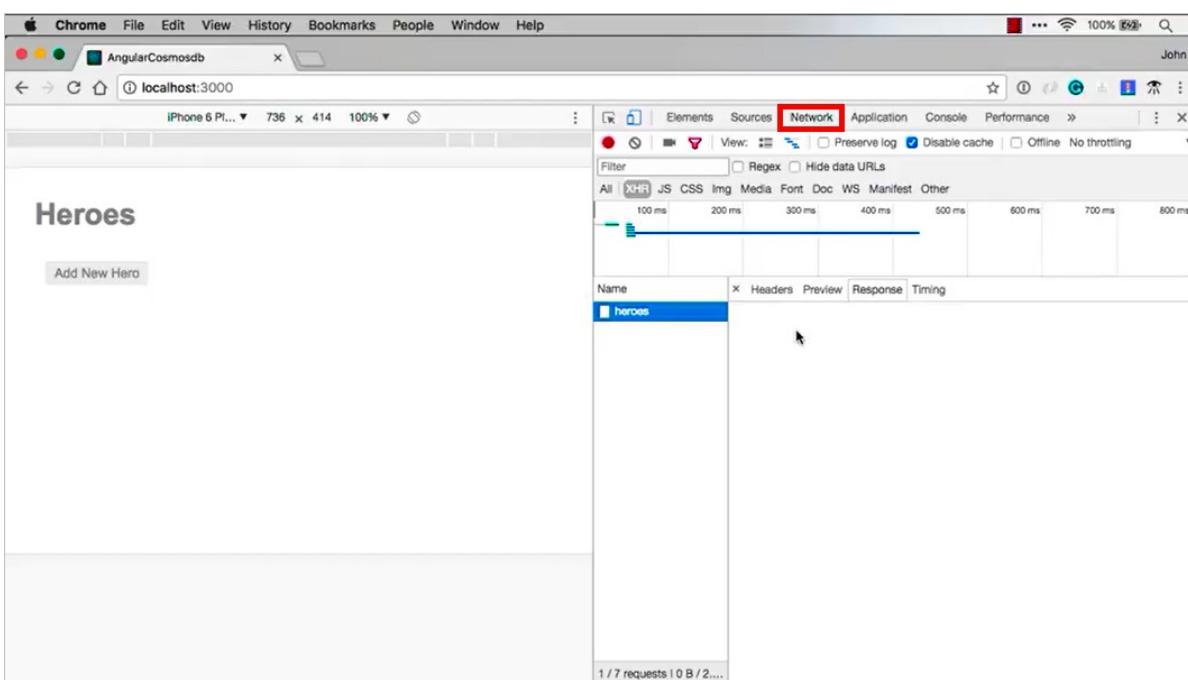
```
function getHeroes(req, res) {
```

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

12/13/2019 • 4 minutes to read • [Edit Online](#)

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.

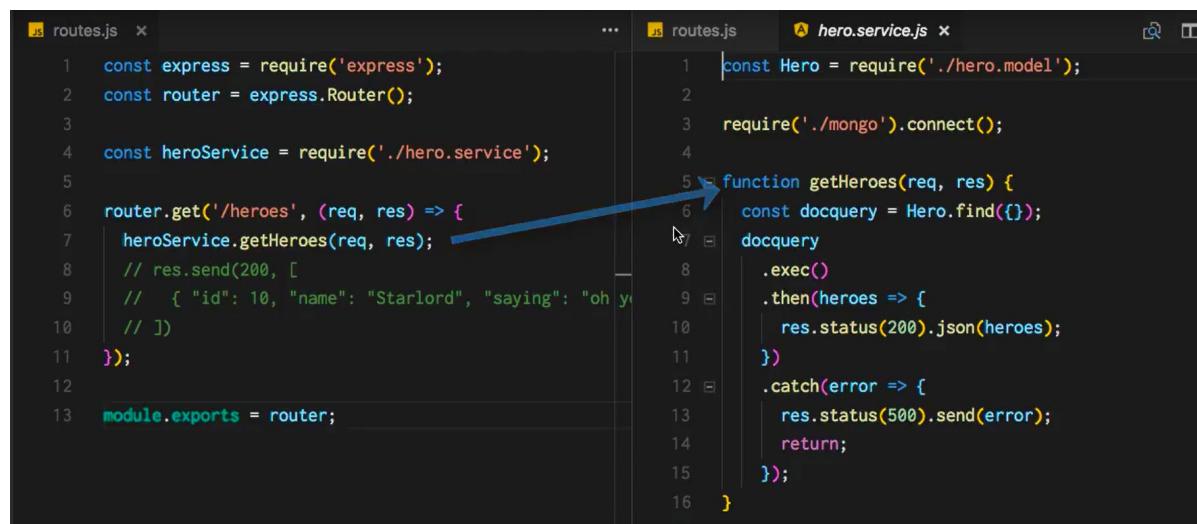
### TIP

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 ya" }
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     });
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;
  }
}
```

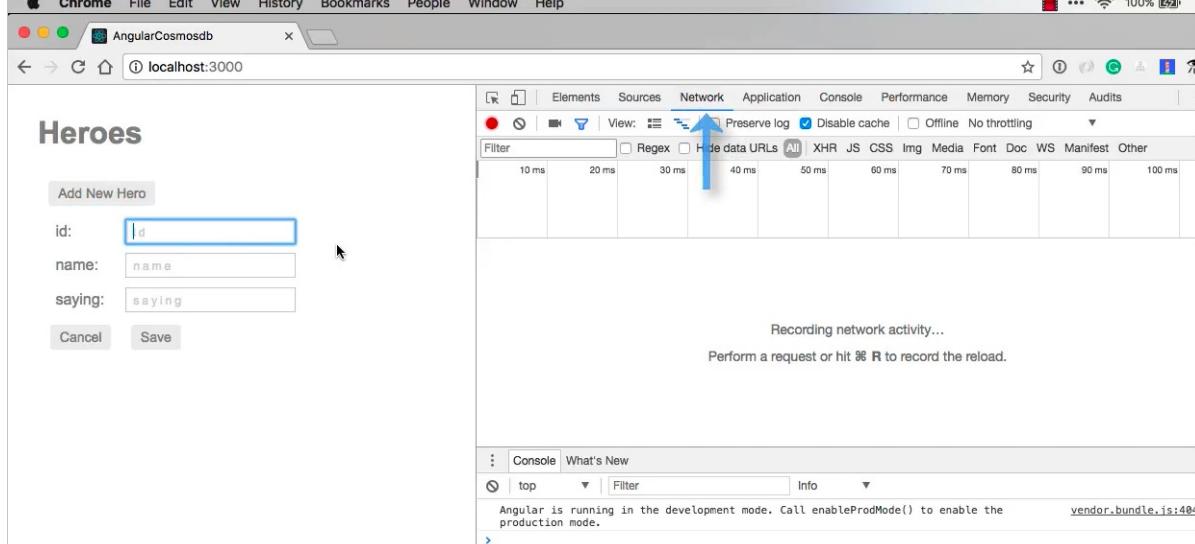
- In **hero.service.js**, update the `module.exports` to include the new `postHero` function.

```
module.exports = {
  getHeroes,
  postHero
};
```

- 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.

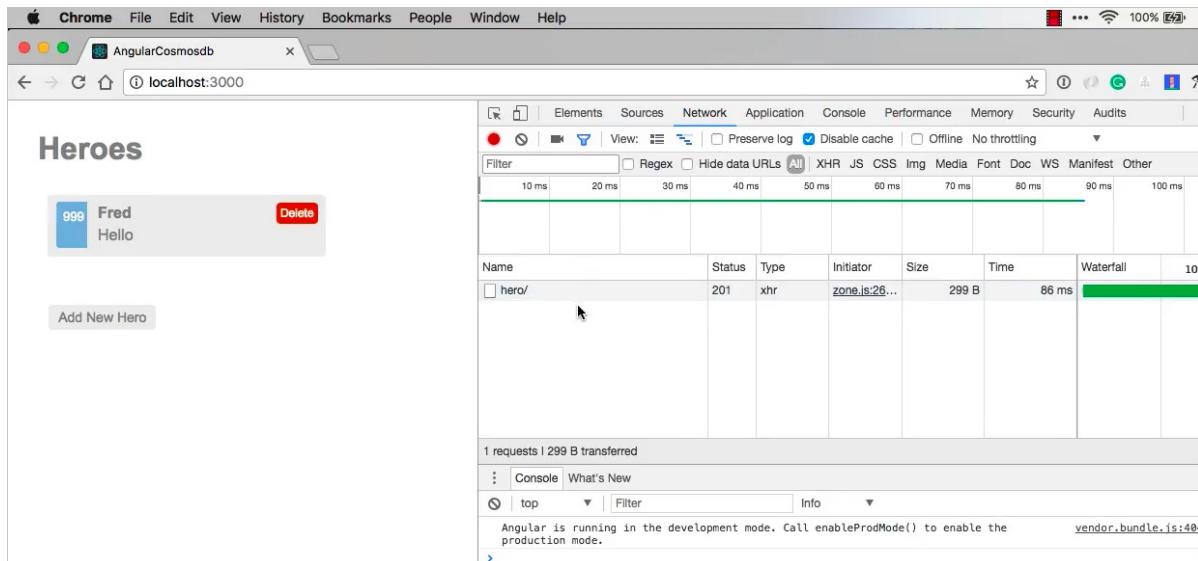
```
router.post('/hero', (req, res) => {
  heroService.postHero(req, res);
});
```

- 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 .
- 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.



- 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.



The screenshot shows the Chrome Developer Tools Network tab. The URL is localhost:3000. A table lists a single request:

Name	Status	Type	Initiator	Size	Time	Waterfall
hero/	201	xhr	zone.js:26...	299 B	86 ms	

Below the table, the status bar indicates "1 requests | 299 B transferred". The bottom of the Network tab shows the message "Angular is running in the development mode. Call enableProdMode() to enable the production mode." and the file "vendor.bundle.js:404".

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".

Name	Status	Type	Initiator	Size	Time	Waterfall	Time
hero/	201	xhr	zone.js:26...	298 B	77 ms		
9	200	xhr	zone.js:26...	294 B	111 ms		
999	200	xhr	zone.js:26...	294 B	54 ms		

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

8/19/2019 • 2 minutes to read • [Edit Online](#)

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.

Import MongoDB data into Azure Cosmos DB

# Tutorial: Migrate MongoDB to Azure Cosmos DB's API for MongoDB offline using DMS

1/8/2020 • 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:

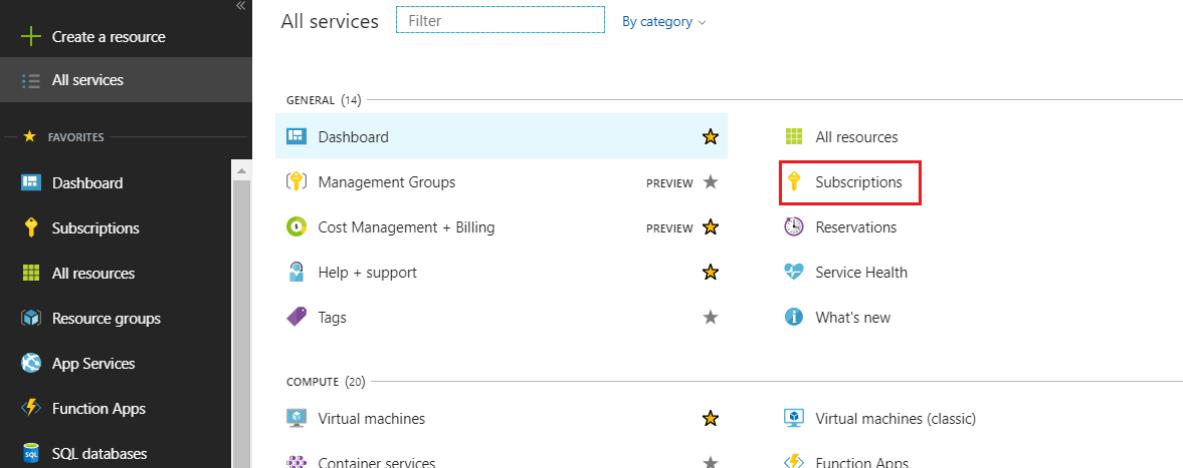
- Target database endpoint (for example, SQL endpoint, Cosmos DB endpoint, and so on)
- Storage endpoint
- Service bus endpoint

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.

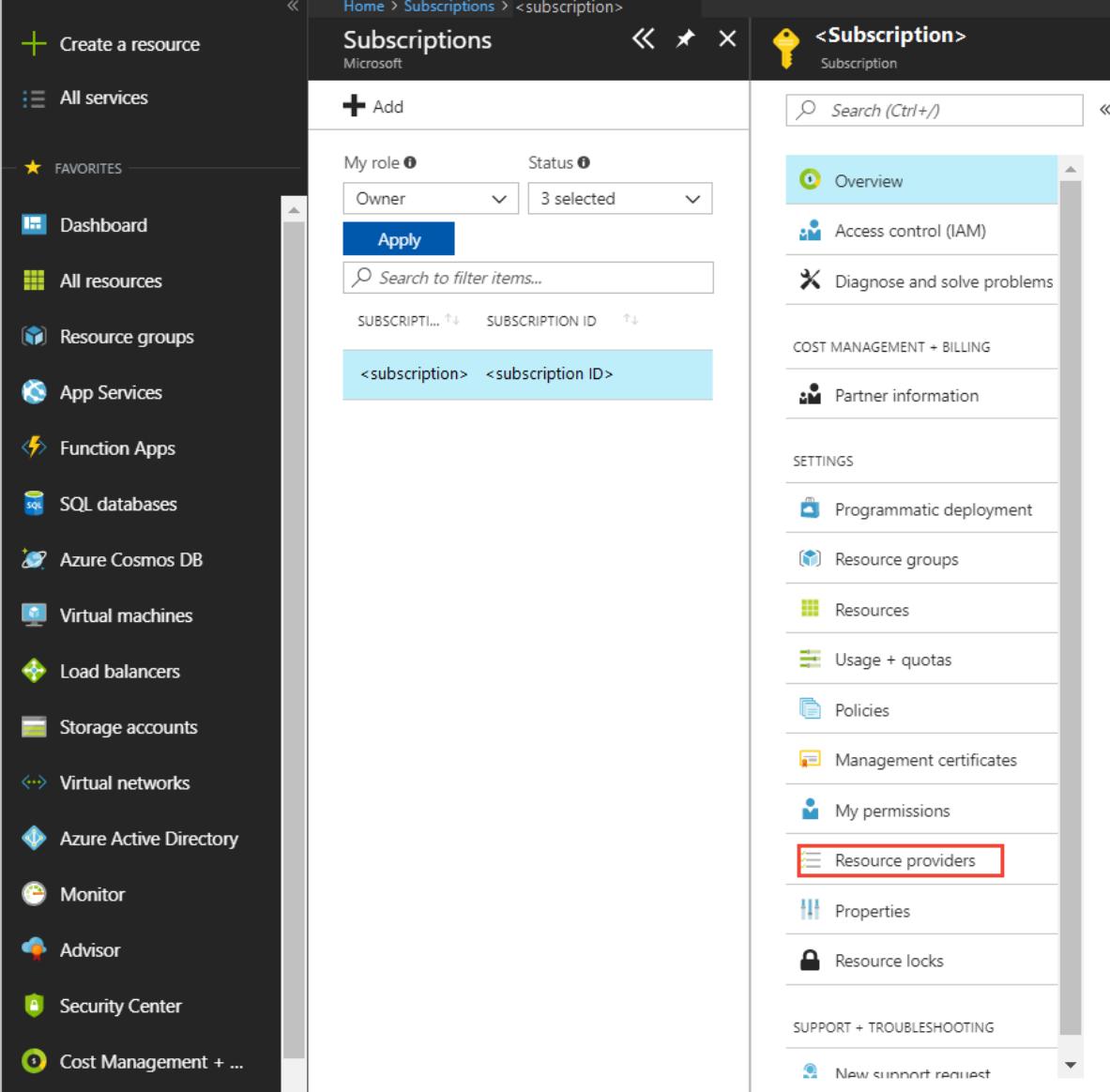
# Register the Microsoft.DataMigration resource provider

1. Sign in to the Azure portal, select **All services**, and then select **Subscriptions**.



The screenshot shows the Azure portal's left sidebar with 'All services' selected. The main area displays a grid of service tiles. In the top right corner of the grid, the 'Subscriptions' tile is highlighted with a red box.

2. Select the subscription in which you want to create the instance of the Azure Database Migration Service, and then select **Resource providers**.



The screenshot shows the 'Subscriptions' blade. On the right, a navigation menu is open under '<Subscription>'. The 'Resource providers' link is highlighted with a red box.

3. Search for migration, and then to the right of **Microsoft.DataMigration**, select **Register**.

The screenshot shows the Azure portal interface. On the left, there's a sidebar with various service icons like App Services, Function Apps, and Storage accounts. The main area has a breadcrumb path: Home > Subscriptions > Subscription. It displays a list of subscriptions under 'Resource providers'. One entry, 'Microsoft.DataMigration', is shown with a status of 'NotRegistered'. A red box highlights the 'Register' button next to it.

## 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.

This screenshot shows the 'New' screen in the Azure portal. The search bar at the top contains the text 'Azure Database Migration Service', which is also highlighted with a red box. Below the search bar, there's a list of service categories: Networking, Storage, Web + Mobile, Containers, Databases, Data + Analytics, AI + Cognitive Services, Internet of Things, Enterprise Integration, Security + Identity, Developer tools, Monitoring + Management, Add-ons, and Blockchain. Under the 'Databases' category, 'Azure Database Migration Service' is listed with a red box around its name. Other items in this category include 'Azure Database for MySQL' and 'Azure Database for PostgreSQL'. To the right of the search bar, there are links for 'Ubuntu Server 16.04 LTS VM', 'Quickstart tutorial', 'DevOps Project', 'Learn more', 'Web App', 'Quickstart tutorial', 'SQL Database', 'Quickstart tutorial', 'Cosmos DB', 'Quickstart tutorial', 'Storage Account', 'Quickstart tutorial', and 'Serverless Function App', 'Quickstart tutorial'. At the bottom of the list, there's a section titled 'Most recently created' with a single item: 'Azure Database Migration Service (preview)'.

2. On the **Azure Database Migration Service** screen, select **Create**.

The screenshot shows the Azure Database Migration Service page. On the left, there's a sidebar with various service icons and names: Create a resource, All services, Favorites, Dashboard, Subscriptions, All resources, Resource groups, App Services, Function Apps, SQL databases, Azure Cosmos DB, Virtual machines, Load balancers, Storage accounts, Virtual networks, Azure Active Directory, Monitor, Advisor, and Security Center. The main content area has a title 'Azure Database Migration Service' with a Microsoft logo. It contains a brief introduction about DMS, steps to prepare, useful links, and a prominent 'Create' button.

The 'Create' button is highlighted with a red box.

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](#).

Create Migration Service

Service Name ●  
DMSTutorial ✓

\* Subscription  
<subscription>

\* Select a resource group ●  
<resource group>  
[Create new](#)

\* Location ●  
South Central US

---

\* Virtual network  
<virtual network>

---

\* Pricing tier  
Standard: 1 vCores

---

**i** Azure Database Migration Service quick start template Experience our database migration service with pre-created source and target

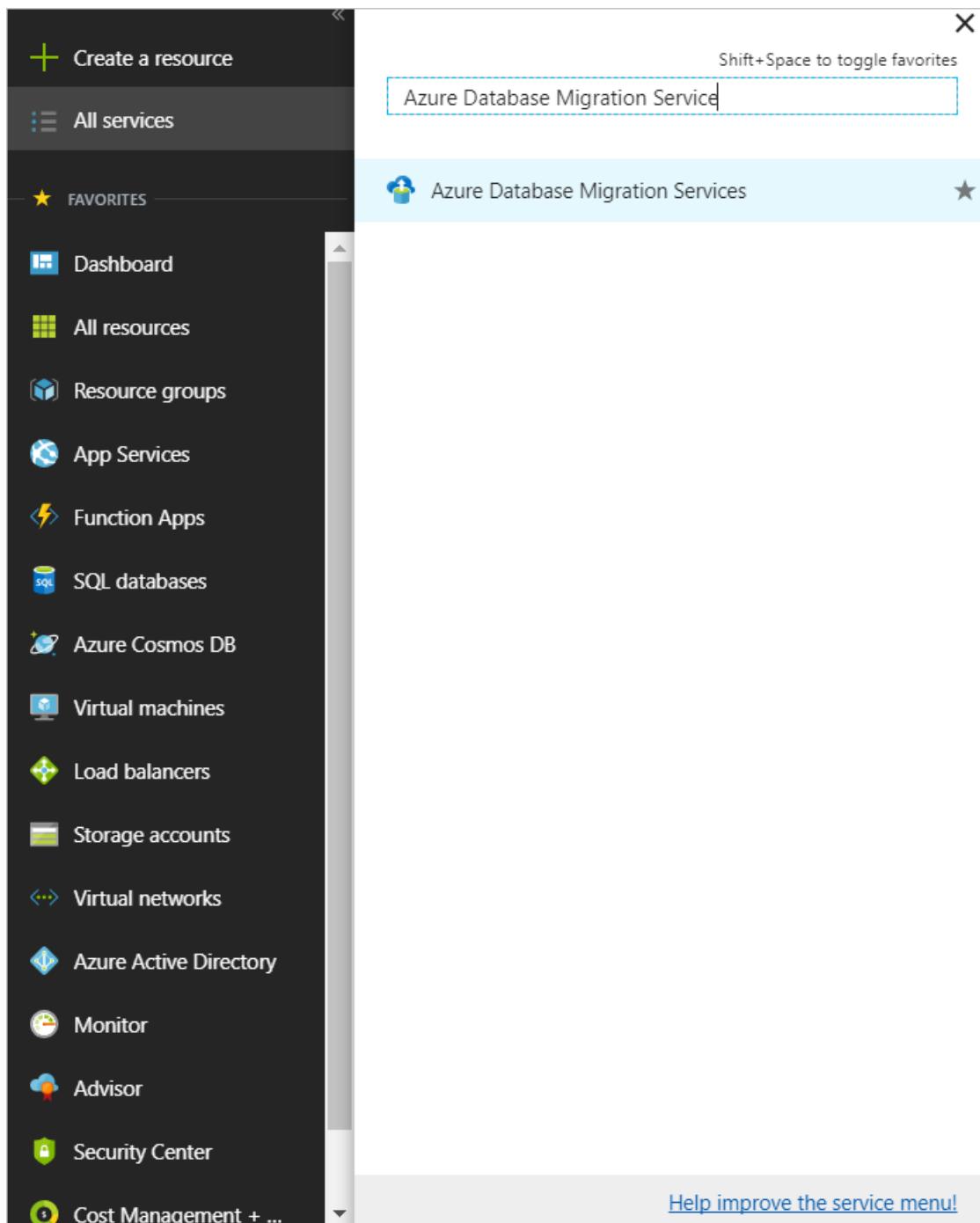
**Create** Automation options

7. Select **Create** to create the service.

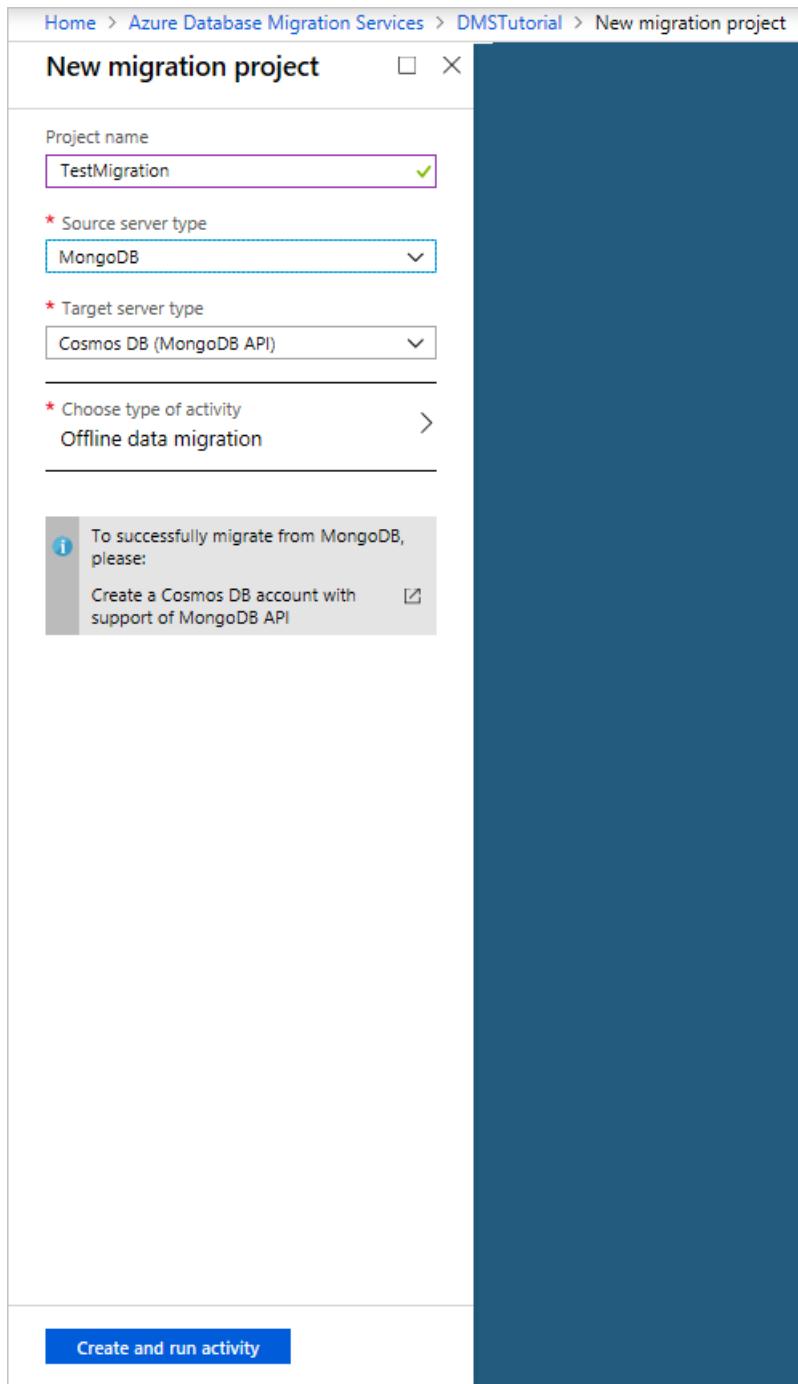
## Create a migration project

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.

Home > Azure Database Migration Services > DMSTutorial > Migration Wizard > Source details

Migration Wizard		Source details
TestMigration		X X
1	Select source	✓
2	Select target	>
3	Database setting	>
4	Collection setting	>
5	Migration summary	>
<hr/> <hr/> <hr/> <hr/> <hr/> <hr/>		
<p>Mode Standard mode</p> <p>* Source server name ⓘ Enter source server name Please enter the name of your source server</p> <p>Server port 27017</p> <p>User Name ⓘ Enter user name</p> <p>Password Enter password</p> <p>Connection properties <input type="checkbox"/> Require SSL</p>		
<p><b>Save</b></p>		

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.

The screenshot shows the Azure Database Migration Services Migration Wizard interface. The left sidebar lists five steps: 1. Select source (done), 2. Select target (selected), 3. Database setting, 4. Collection setting, and 5. Migration summary. Step 2 is currently active. The main panel title is "Migration target details". It includes a warning message: "Using the connection string mode is not recommended as it might reveal sensitive information". Below this are fields for "Mode" (set to "Select Cosmos DB target"), "Subscription" (set to "<subscription>"), "Select Cosmos DB name" (set to "wingtips-tutorial"), and a "Connection string" field containing a MongoDB URI. A "Save" button is located at the bottom of the panel.

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](#).

The screenshot shows the Azure Database Migration Services Migration Wizard interface. The left sidebar lists five steps: 1. Select source (checked), 2. Select target (checked), 3. Database setting, 4. Collection setting, and 5. Migration summary. The main area is titled 'Map to target databases' and contains a table for mapping collections. A row for 'wingtips' is selected, showing 'Create: wingtips' and 'Blank for default or RU between 10000 to 100'. A checkbox for 'Clean up collections' is checked. A 'Save' button is at the bottom.

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](#).

Home > Azure Database Migration Services > DMSTutorial > TestMigration > Migration Wizard > Collection setting

Migration Wizard		TestMigration	X																																			
		Collection setting	X																																			
<b>1</b>	Select source	✓	Mongo collection settings for each database ^ wingtips 6 of 6																																			
<b>2</b>	Select target	✓	<table border="1"> <thead> <tr> <th>NAME</th> <th>TARGET COLLECTION</th> <th>THROUGHPUT (RU/S)</th> <th>SHARD KEY</th> <th>UNIQUE</th> </tr> </thead> <tbody> <tr> <td>Configurations</td> <td>Create: Configurations</td> <td>RU: 1000 ✓</td> <td></td> <td></td> </tr> <tr> <td>Customers</td> <td>Create: Customers</td> <td>RU: 1000 ✓</td> <td>LastName ✓</td> <td></td> </tr> <tr> <td>CustomerTickets</td> <td>Create: CustomerTickets</td> <td>RU: 1000 ✓</td> <td>Customerid ✓</td> <td></td> </tr> <tr> <td>EventSections</td> <td>Create: EventSections</td> <td>RU: 1000 ✓</td> <td>Venuelid ✓</td> <td></td> </tr> <tr> <td>Sections</td> <td>Create: Sections</td> <td>RU: 1000 ✓</td> <td>SectionName ✓</td> <td></td> </tr> <tr> <td>Venues</td> <td>Create: Venues</td> <td>RU: 1000 ✓</td> <td>VenueName ✓</td> <td></td> </tr> </tbody> </table>	NAME	TARGET COLLECTION	THROUGHPUT (RU/S)	SHARD KEY	UNIQUE	Configurations	Create: Configurations	RU: 1000 ✓			Customers	Create: Customers	RU: 1000 ✓	LastName ✓		CustomerTickets	Create: CustomerTickets	RU: 1000 ✓	Customerid ✓		EventSections	Create: EventSections	RU: 1000 ✓	Venuelid ✓		Sections	Create: Sections	RU: 1000 ✓	SectionName ✓		Venues	Create: Venues	RU: 1000 ✓	VenueName ✓	
NAME	TARGET COLLECTION	THROUGHPUT (RU/S)	SHARD KEY	UNIQUE																																		
Configurations	Create: Configurations	RU: 1000 ✓																																				
Customers	Create: Customers	RU: 1000 ✓	LastName ✓																																			
CustomerTickets	Create: CustomerTickets	RU: 1000 ✓	Customerid ✓																																			
EventSections	Create: EventSections	RU: 1000 ✓	Venuelid ✓																																			
Sections	Create: Sections	RU: 1000 ✓	SectionName ✓																																			
Venues	Create: Venues	RU: 1000 ✓	VenueName ✓																																			
<b>3</b>	Database setting	✓																																				
<b>4</b>	Collection setting	>																																				
<b>5</b>	Migration summary	>																																				

**Save**

4. Select **Save**.
5. On the **Migration summary** screen, in the **Activity name** text box, specify a name for the migration activity.

Home > Azure Database Migration Services > DMSTutorial > TestMigration > Migration Wizard > Migration summary

Migration Wizard		TestMigration	X
		Migration summary	X
<b>1</b>	Select source	✓	<p>Activity name <input type="text" value="TestMigration"/></p>
<b>2</b>	Select target	✓	<p>Target server name wingtips-tutorial.documents.azure.com Target server version 3.2.0</p>
<b>3</b>	Database setting	✓	<p>Source server name 40.74.232.123 Source server version 3.2.21</p>
<b>4</b>	Collection setting	✓	<p>Database(s) to migrate 1 of 1 <input checked="" type="checkbox"/> Boost RU to during initial data copy</p>
<b>5</b>	Migration summary	>	

**Run migration**

## Run the migration

- Select **Run migration**.

The migration activity window appears, and the **Status** of the activity is **Not started**.

The screenshot shows the Azure Database Migration Services interface. At the top, there's a breadcrumb navigation: Home > Azure Database Migration Services > DMSTutorial > TestMigration > TestMigration. Below the breadcrumb is a title bar for 'TestMigration' with a 'Delete migration' button, a 'Stop migration' button (disabled), a 'Refresh' button, and a 'Download logs' link. Underneath the title bar are two rows of metrics: 'Status' and 'Duration' (Throughput (bytes/s) and Throughput (documents/s)), and 'Total bytes' and 'Total documents'. A search bar labeled 'Search to filter items...' is present. A table below lists migration tasks:

NAME	STATUS	ERRORS	DOCUM...	BYTES C...	DURATI...
wingtips	Not started	0	0 Byte(s)	---	...

## 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.

Home > Azure Database Migration Services > DMSTutorial > TestMigration > TestMigration

## TestMigration

TestMigration

Delete migration Stop migration Refresh Download logs

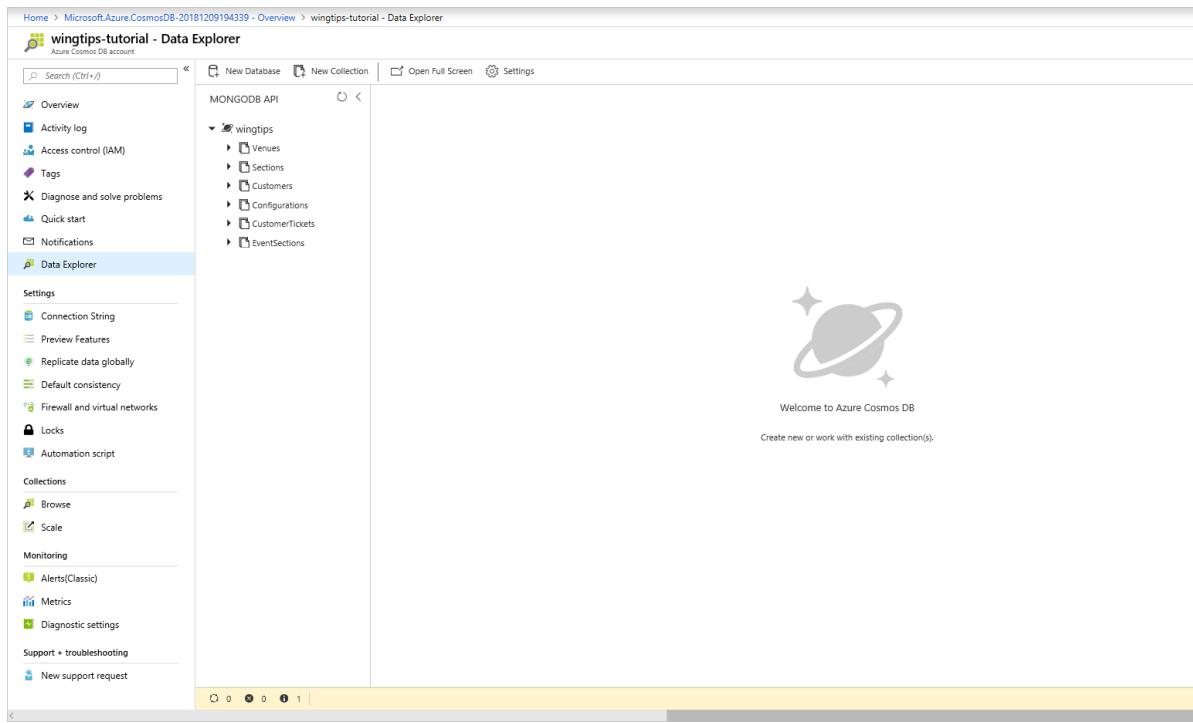
Status Complete	Duration 00:00:16
Throughput (bytes/s) Complete: 39.01 KB of 39.01 KB	Throughput (documents/s) Complete: 123 of 123
Total bytes 39.01 KB	Total documents 123

Search to filter items...

NAME	STATUS	ERRORS	DOCUM...	BYTES C...	DURATI...	...
wingtips	Complete		123/123	39.01 KB/3...	00:00:16	...

## 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](#).

# Query data by using Azure Cosmos DB's API for MongoDB

12/5/2019 • 3 minutes to read • [Edit Online](#)

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",  

```

## 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

12/13/2019 • 4 minutes to read • [Edit Online](#)

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.

andrl - Replicate data globally  
Azure Cosmos DB account

Search (Ctrl+/  
)

Save Discard Manual Failover Failover Priorities

Overview Activity log Access control (IAM) Tags Diagnose and solve problems Quick start Data Explorer

SETTINGS

Keys Replicate data globally Default consistency Firewall Add Azure Search Properties Locks Automation script

MONITORING

Metrics Alert rules

Click on a location to add or remove regions from your Azure Cosmos DB account  
\* Each region is billable based on the throughput and storage for the account. [Learn more](#)



WRITER REGION

Central US

READ REGIONS

Australia East

Australia Southeast

Brazil South

Canada Central

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](#)

# Pre-migration steps for data migrations from MongoDB to Azure Cosmos DB's API for MongoDB

1/14/2020 • 5 minutes to read • [Edit Online](#)

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 [AutoPilot model](#).
- **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
Offline	<a href="#">Data Migration Tool</a>	<ul style="list-style-type: none"><li>• Easy to set up and supports multiple sources</li><li>• Not suitable for large datasets.</li></ul>

MIGRATION TYPE	SOLUTION	CONSIDERATIONS
Offline	Azure Data Factory	<ul style="list-style-type: none"> <li>• Easy to set up and supports multiple sources</li> <li>• Makes use of the Azure Cosmos DB bulk executor library</li> <li>• Suitable for large datasets</li> <li>• Lack of checkpointing means that any issue during the course of migration would require a restart of the whole migration process</li> <li>• Lack of a dead letter queue would mean that a few erroneous files could stop the entire migration process</li> <li>• Needs custom code to increase read throughput for certain data sources</li> </ul>
Offline	Existing Mongo Tools (mongodump, mongorestore, Studio3T)	<ul style="list-style-type: none"> <li>• Easy to set up and integration</li> <li>• Needs custom handling for throttles</li> </ul>
Online	Azure Database Migration Service	<ul style="list-style-type: none"> <li>• Fully managed migration service.</li> <li>• Provides hosting and monitoring solutions for the migration task.</li> <li>• Suitable for large datasets and takes care of replicating live changes</li> <li>• Works only with other MongoDB sources</li> </ul>

## 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 `getLastRequestStatistics` 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

By default, Azure Cosmos DB provides automatic indexing on all data inserted. The indexing capabilities provided by Azure Cosmos DB include adding composite 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 MongoDB](#).

[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](#)

# Post-migration optimization steps when using Azure Cosmos DB's API for MongoDB

1/14/2020 • 2 minutes to read • [Edit Online](#)

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.

The screenshot shows the Microsoft Azure portal interface. The title bar says "Connection String Inform X". The address bar shows "portal.azure.com/?feature.custompoi". The main content area is titled "Microsoft Azure mongodb - Connection String". On the left, there's a sidebar with various icons and a list of settings. The "Connection String" item is highlighted with a red box. The main panel displays connection information for a MongoDB account, including fields for HOST, PORT, USERNAME, PRIMARY PASSWORD, SECONDARY PASSWORD, PRIMARY CONNECTION STRING, SECONDARY CONNECTION STRING, and SSL. There are also sections for "Read-write Keys" and "Read-only Keys". A note at the bottom states: "Azure Cosmos DB has strict security requirements and standards. Azure Cosmos DB accounts require authentication and secure communication via SSL."

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](#)

# Connect a MongoDB application to Azure Cosmos DB

12/5/2019 • 2 minutes to read • [Edit Online](#)

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.

The screenshot shows the Azure Cosmos DB Quick Start blade for the account 'cdbmongotest'. The left sidebar contains navigation links for Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Quick start (which is selected), Notifications, Data Explorer, Settings (with Connection String, Replicate data globally, Default consistency, Firewall and virtual networks, Private Endpoint Connections, Preview Features, Locks, and Export template), Collections (with Browse and Scale), and Monitoring (with Alerts). The main content area displays a success message: 'Congratulations! Your Azure Cosmos DB for MongoDB API account is ready.' It instructs users to connect their existing MongoDB app to the account. A 'Choose a platform' section offers links for .NET (selected), Node.js, MongoDB Shell, Java, Python, and Others. Step 1, 'Connect your existing MongoDB .NET app', provides an example code snippet for connecting to the database using the MongoDB .NET driver:

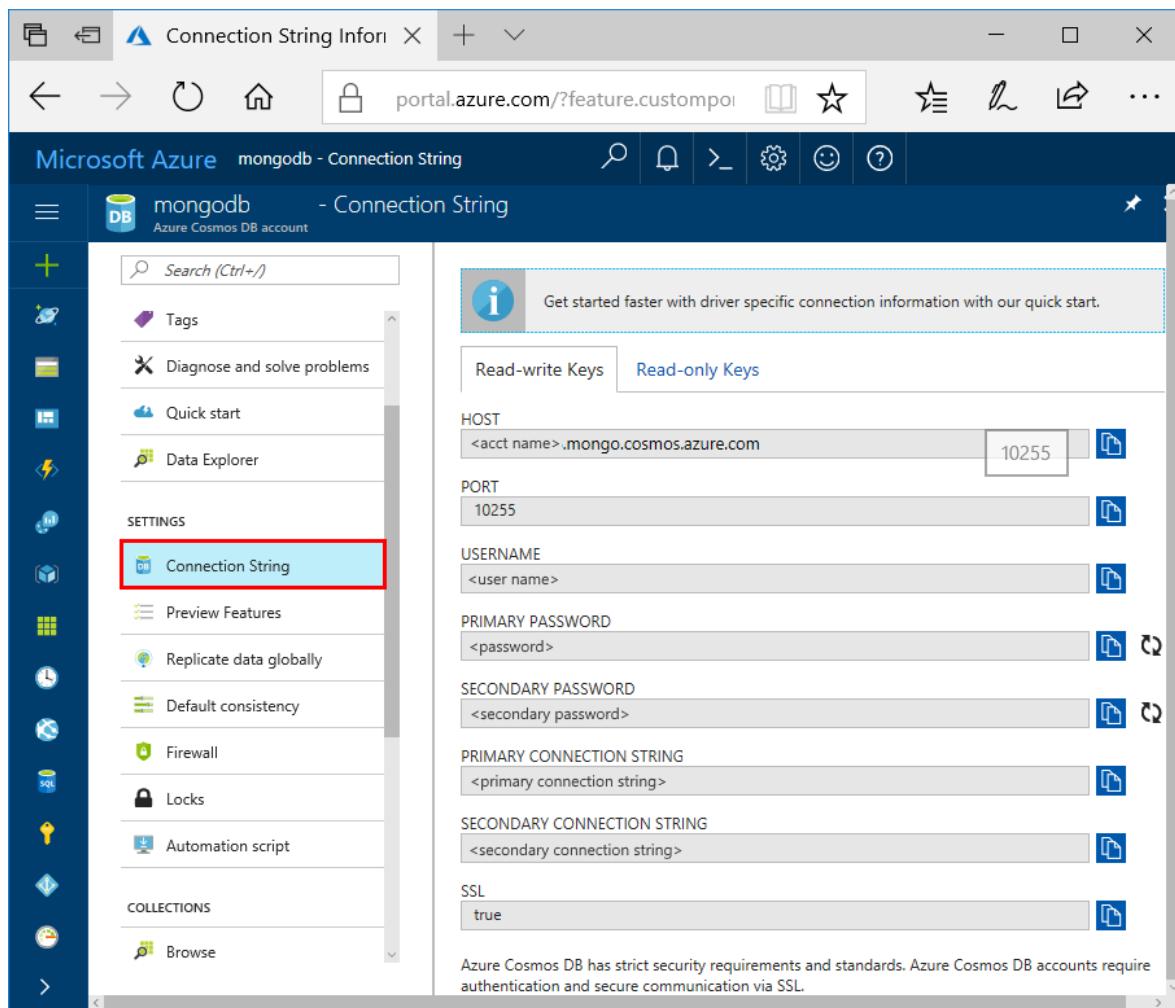
```
string connectionString =
    @"mongodb://cdbmongotest:7ZPicxHRU1LKc9khwM4xZbw12Pv2dHgfs7rXxb7iDtsEy5qCCavX1
MongoClientSettings settings = MongoClientSettings.FromUrl(
    new MongoUrl(connectionString)
);
settings.SslSettings =
    new SslSettings() { EnabledSslProtocols = SslProtocols.Tls12 };
var mongoClient = new MongoClient(settings);
```

Step 1 also includes a 'PRIMARY CONNECTION STRING' field containing the URL: `mongodb://cdbmongotest:7ZPicxHRU1LKc9khwM4xZbw12Pv2dHgfs7rXxb7iDtsEy5qCCavX1...`. A 'Download' button is available next to the string.

Step 2, 'Learn More', lists additional resources: Code Samples, Migrating to Azure Cosmos DB, Documentation, Using Robomongo, Using MongoChef, Pricing, and Forum.

## 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 SSL.

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 SSL. 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

12/13/2019 • 2 minutes to read • [Edit Online](#)

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.

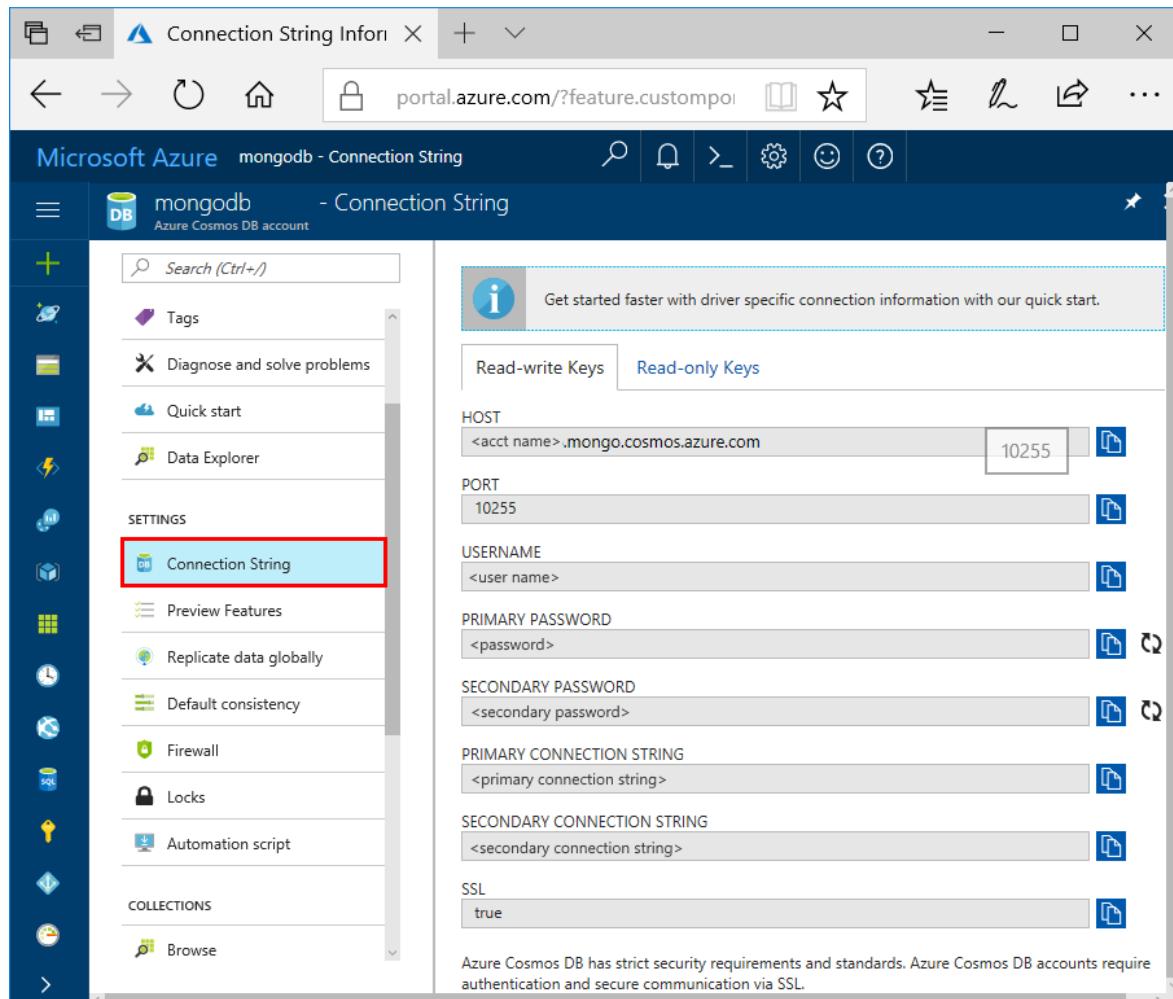
## NOTE

Currently, Robo 3T v1.2 and lower versions are supported with Cosmos DB's API for MongoDB.

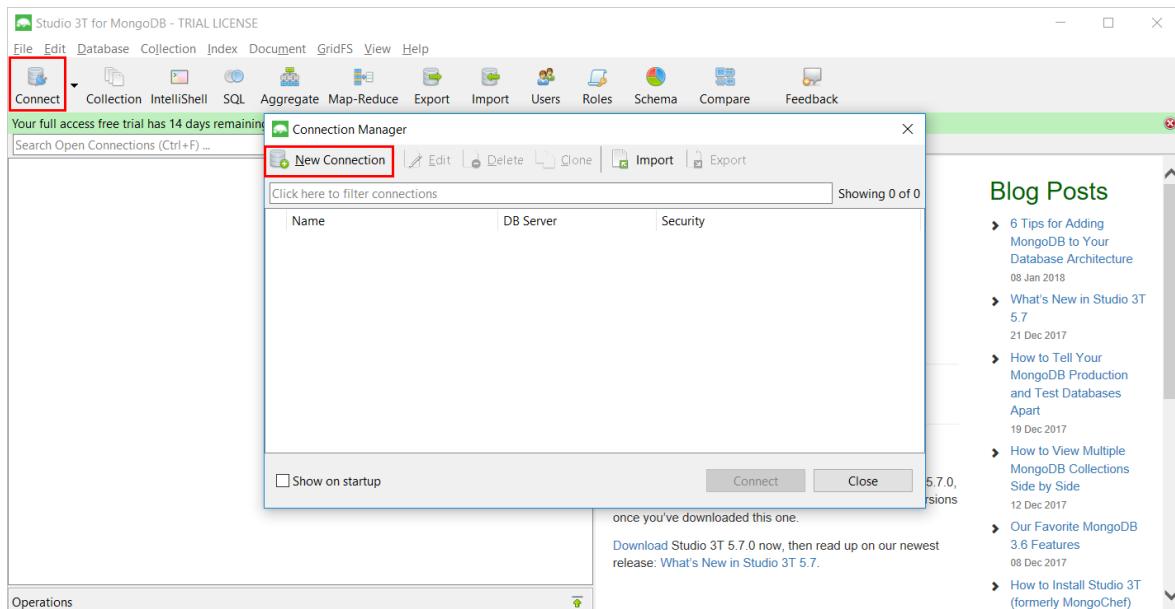
## Create the connection in Studio 3T

To add your Azure Cosmos account to the Studio 3T connection manager, use the following steps:

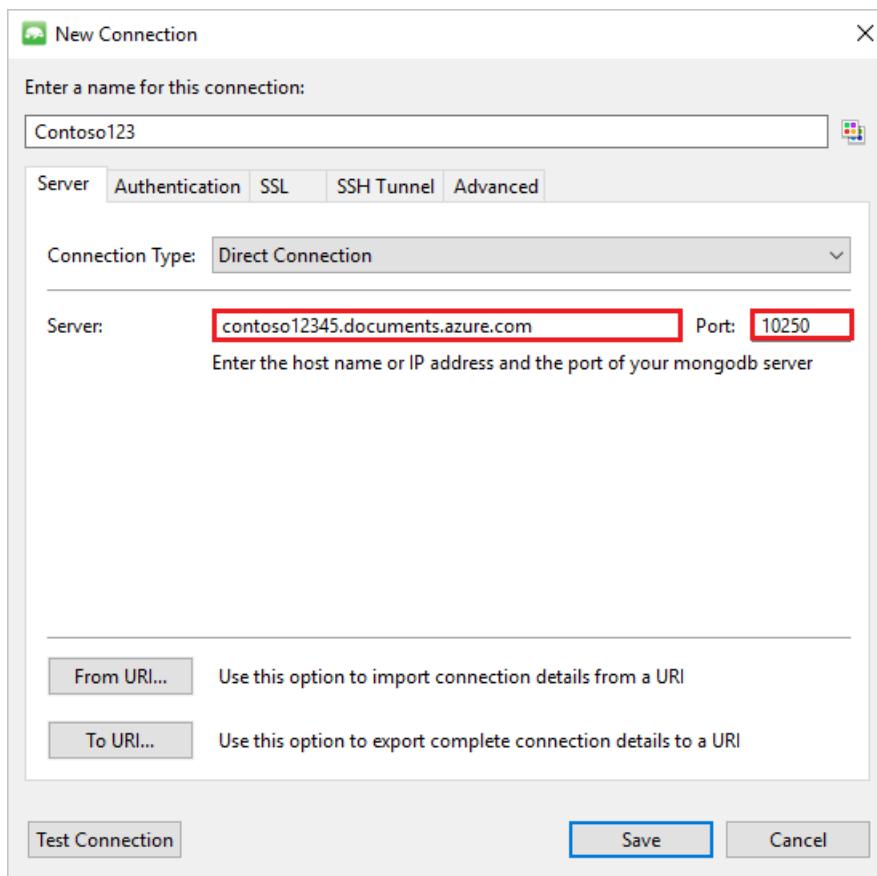
1. Retrieve the connection information for your Azure Cosmos DB's API for MongoDB account using the instructions in the [Connect a MongoDB application to Azure Cosmos DB](#) article.



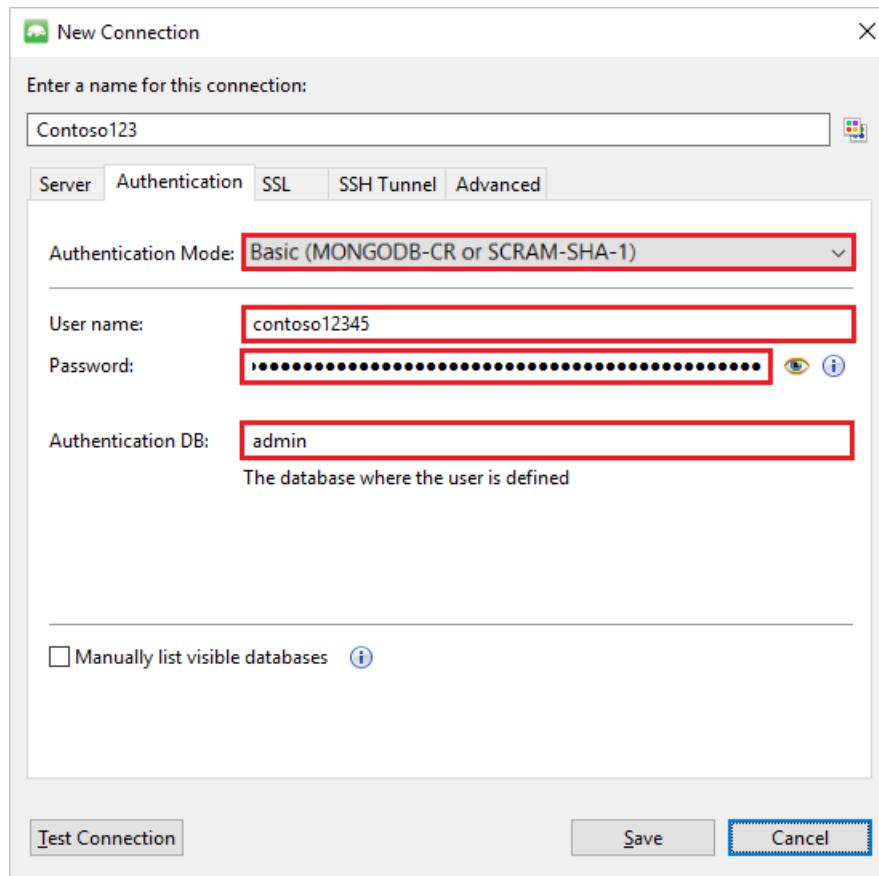
2. Click **Connect** to open the Connection Manager, then click **New Connection**



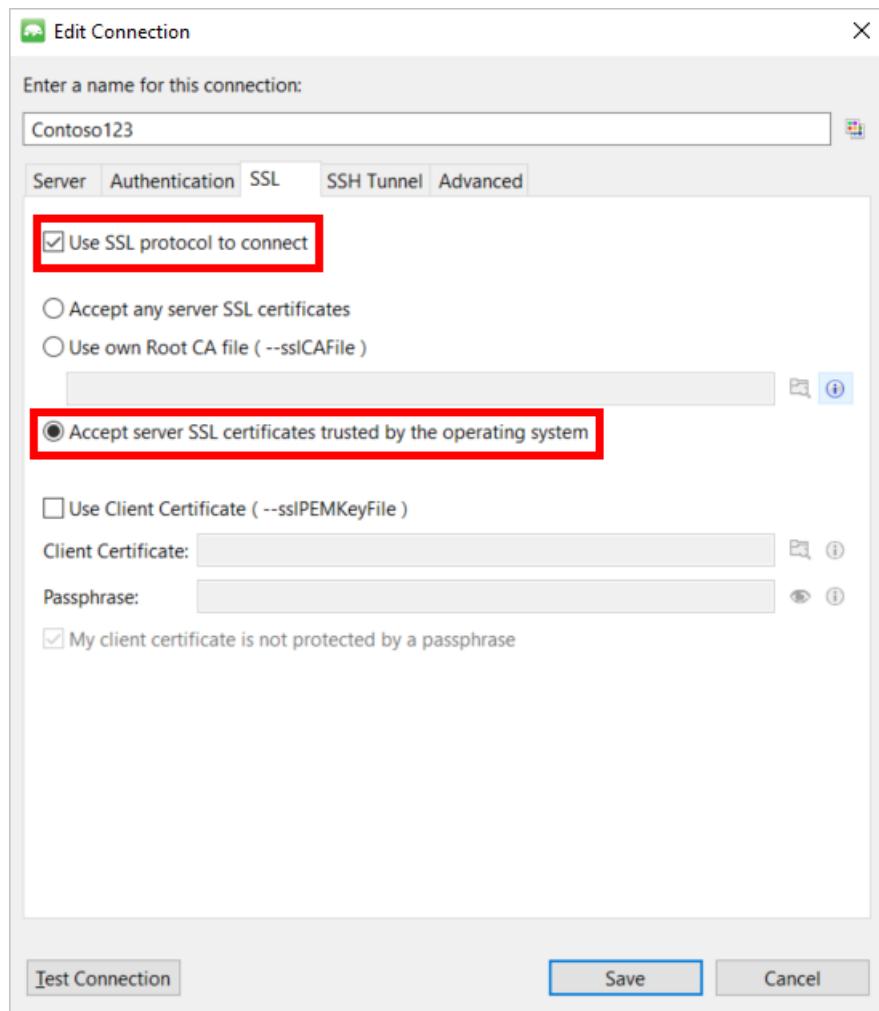
3. In the **New Connection** window, on the **Server** tab, enter the HOST (FQDN) of the Azure Cosmos account and the PORT.



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.

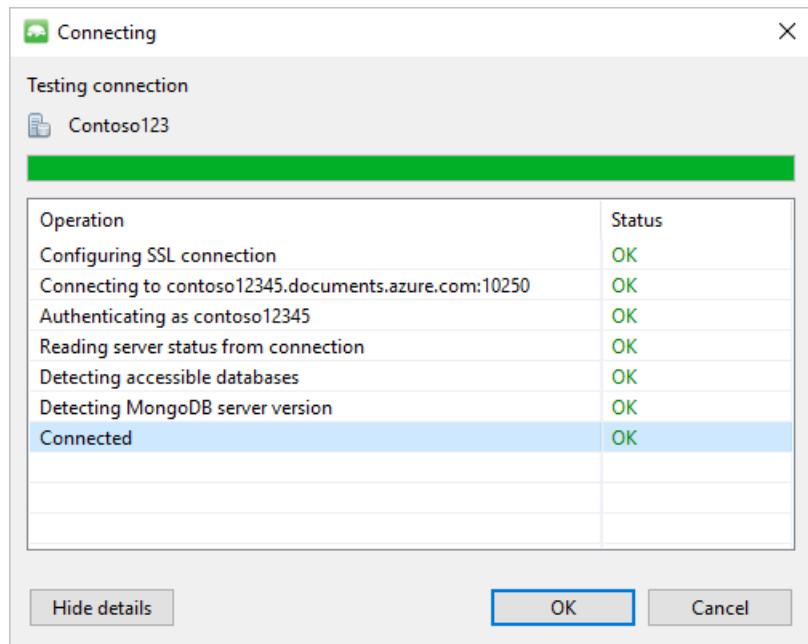


5. In the **New Connection** window, on the **SSL** tab, check the **Use SSL protocol to connect** check box and the **Accept server self-signed SSL certificates** radio button.



6. Click the **Test Connection** button to validate the connection information, click **OK** to return to the New

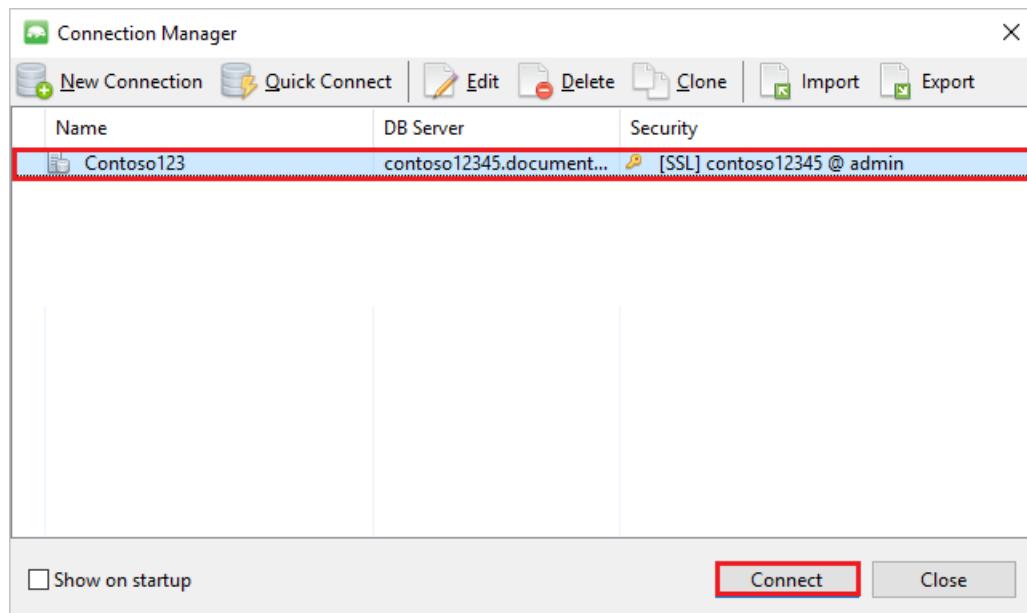
Connection window, and then click **Save**.



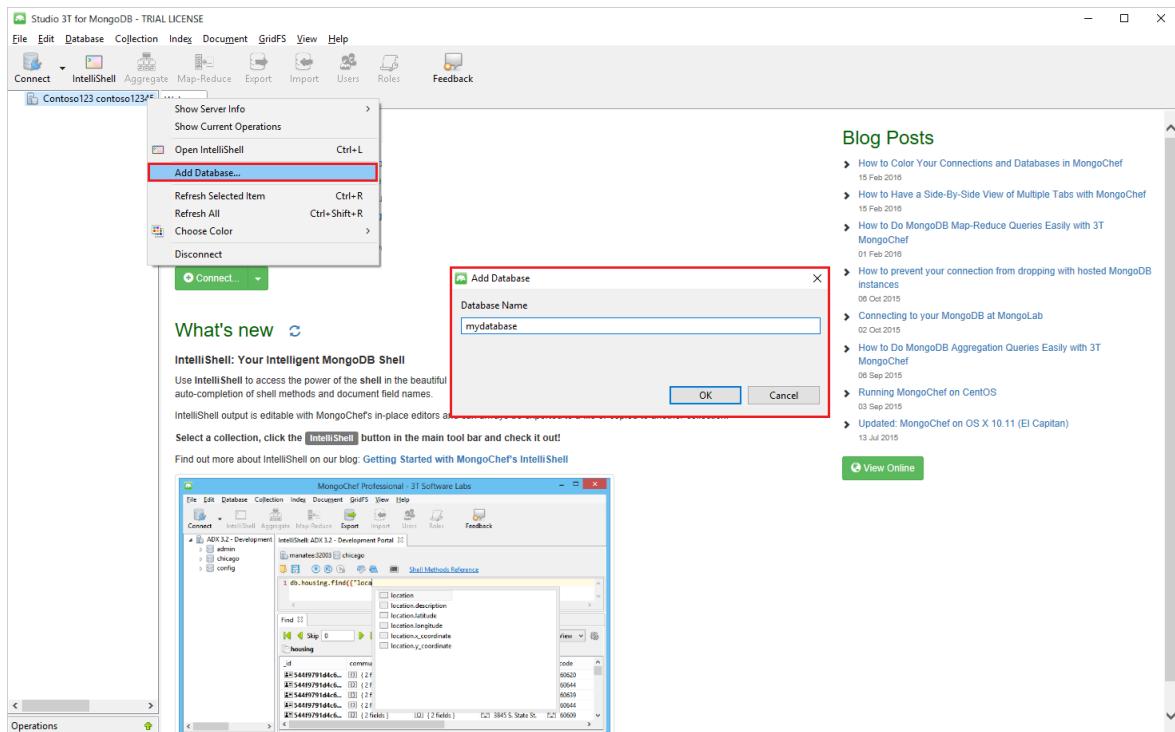
## Use Studio 3T to create a database, collection, and documents

To create a database, collection, and documents using Studio 3T, perform the following steps:

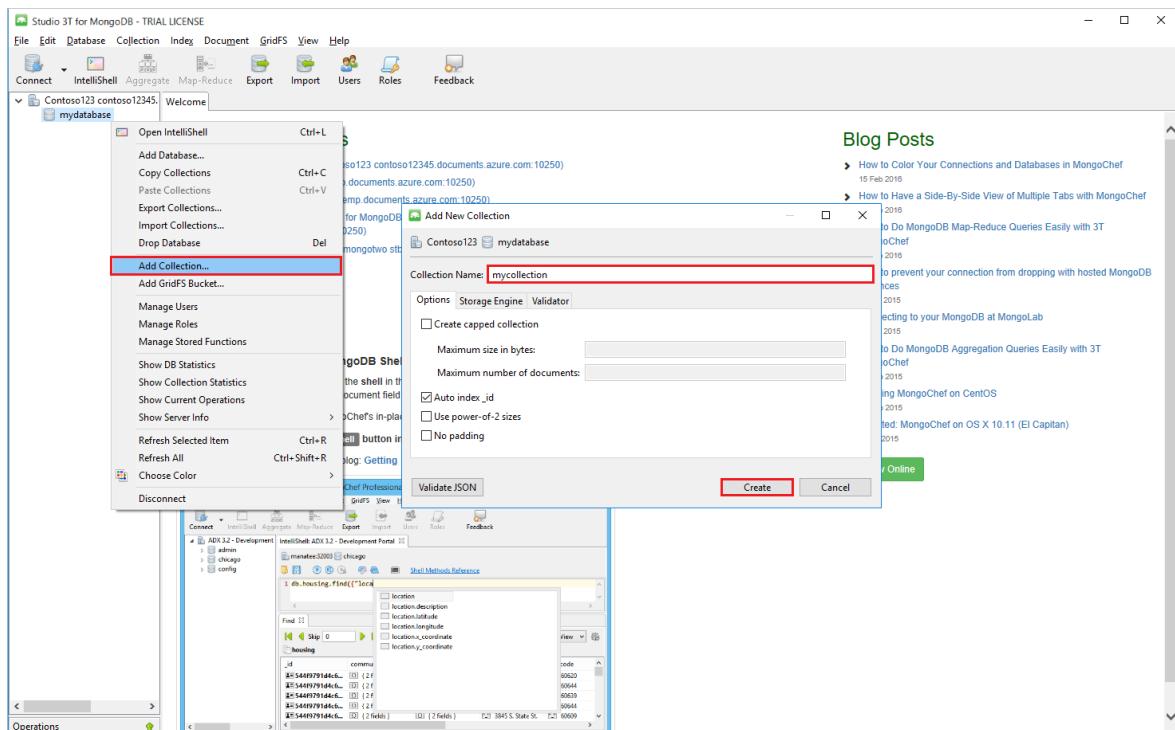
1. In **Connection Manager**, highlight the connection and click **Connect**.



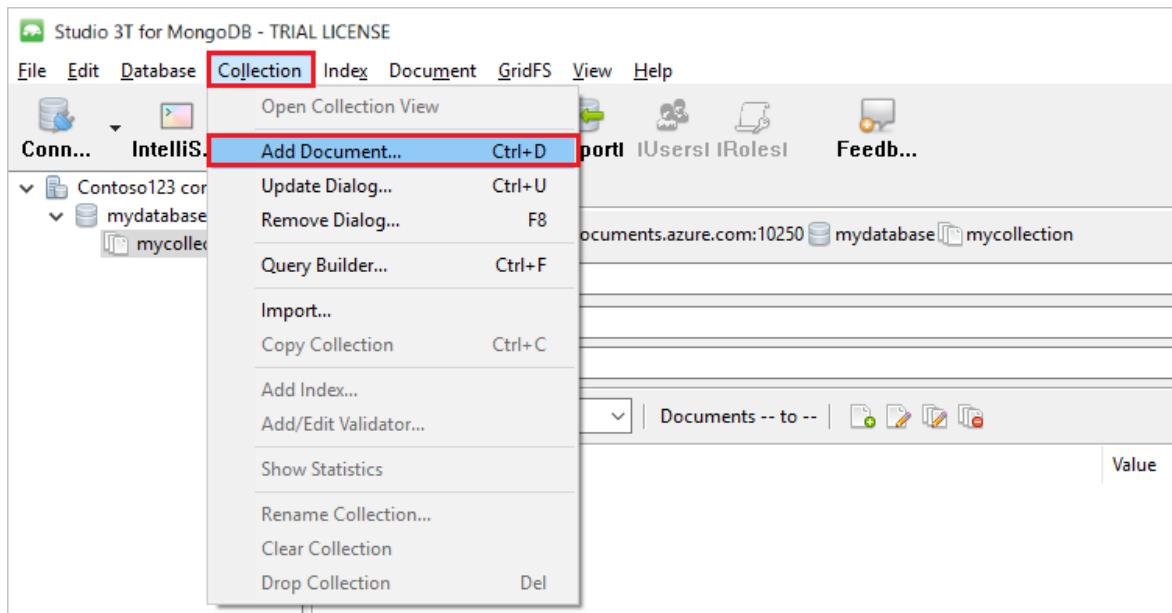
2. Right-click the host and choose **Add Database**. Provide a database name and click **OK**.



### 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.

Key	Value	Type
\$_id	AndersenFamily	Document
parents	[2 elements]	Array
0	{1 fields}	Object
firstName	Thomas	String
1	{1 fields}	Object
firstName	Mary Kay	String
state	WA	String

## 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

1/14/2020 • 2 minutes to read • [Edit Online](#)

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 Robo 3T, 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](#).

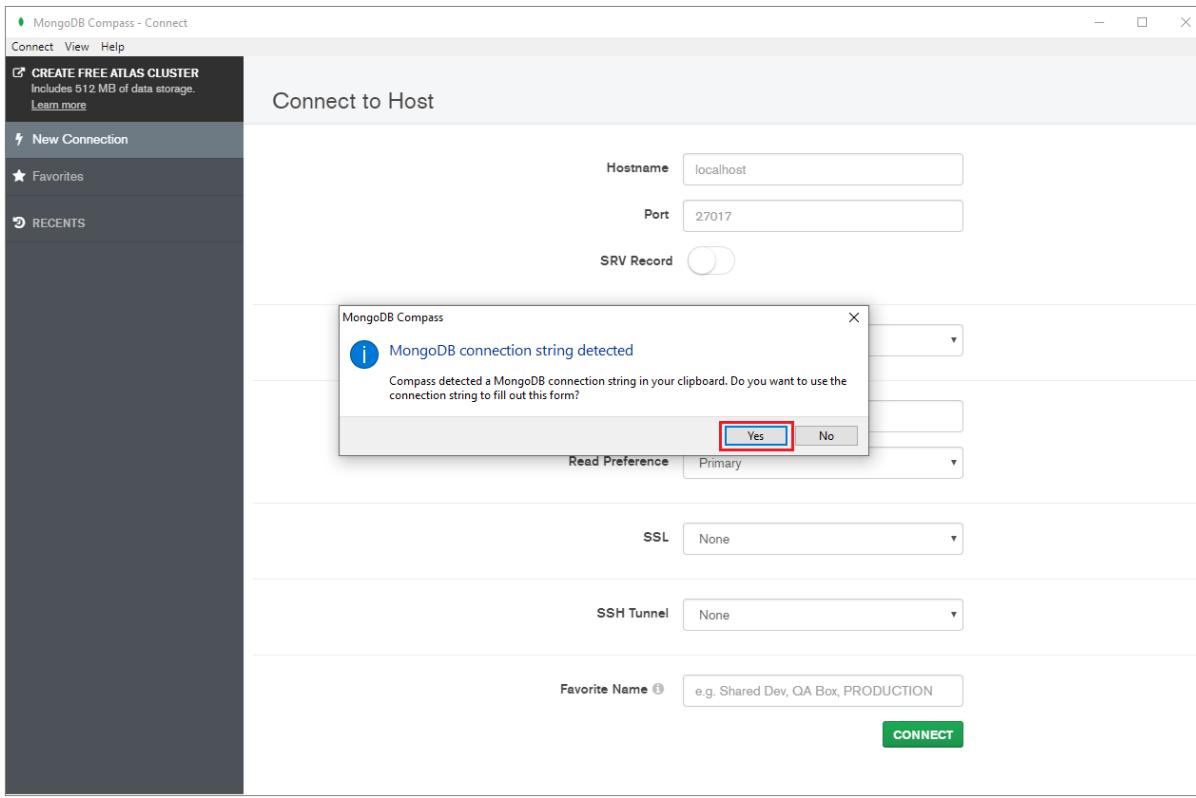
The screenshot shows the Microsoft Azure portal interface. The title bar says 'Connection String Info' and the URL is 'portal.azure.com/?feature.custompopo'. The main content area is titled 'Microsoft Azure mongodb - Connection String'. On the left, there's a sidebar with various icons and a list of settings, one of which, 'Connection String', is highlighted with a red box. The main panel shows connection details: HOST <acct name>.mongo.cosmos.azure.com, PORT 10255, USERNAME <user name>, PRIMARY PASSWORD <password>, SECONDARY PASSWORD <secondary password>. Below these are two large input fields: 'PRIMARY CONNECTION STRING' containing '<primary connection string>' and 'SECONDARY CONNECTION STRING' containing '<secondary connection string>'. Both of these fields are also highlighted with a red box. At the bottom, a note says 'Azure Cosmos DB has strict security requirements and standards. Azure Cosmos DB accounts require authentication and secure communication via SSL.'

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.

This screenshot shows the 'Copy to clipboard' button next to both the Primary and Secondary connection strings, which are highlighted with a red box. The 'Copy to clipboard' button is located in a dark rectangular box. The connection strings are labeled 'PRIMARY CONNECTION STRING' and 'SECONDARY CONNECTION STRING' respectively, each followed by an input field containing the connection string and a 'Copy to clipboard' button.

Azure Cosmos DB has strict security requirements and standards. Azure Cosmos DB accounts require authentication and secure communication via SSL.

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.



- 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.

A screenshot of the 'Replica Set Name' input field in MongoDB Compass. The field is empty and has a blue border, indicating it is selected or active.

- 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

3/5/2019 • 2 minutes to read • [Edit Online](#)

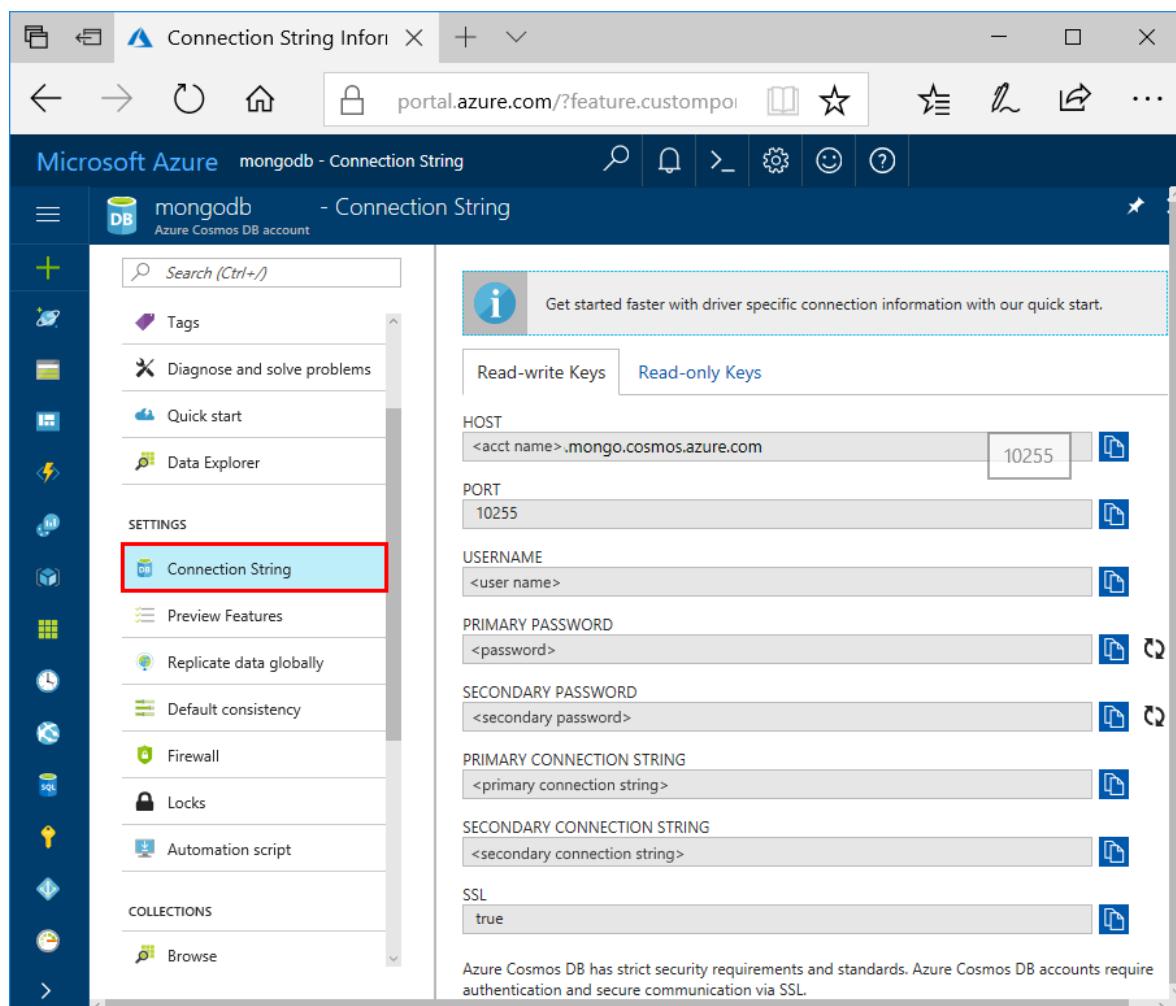
To connect to Cosmos account using Robo 3T, you must:

- Download and install [Robo 3T](#)
- Have your Cosmos DB [connection string](#) information

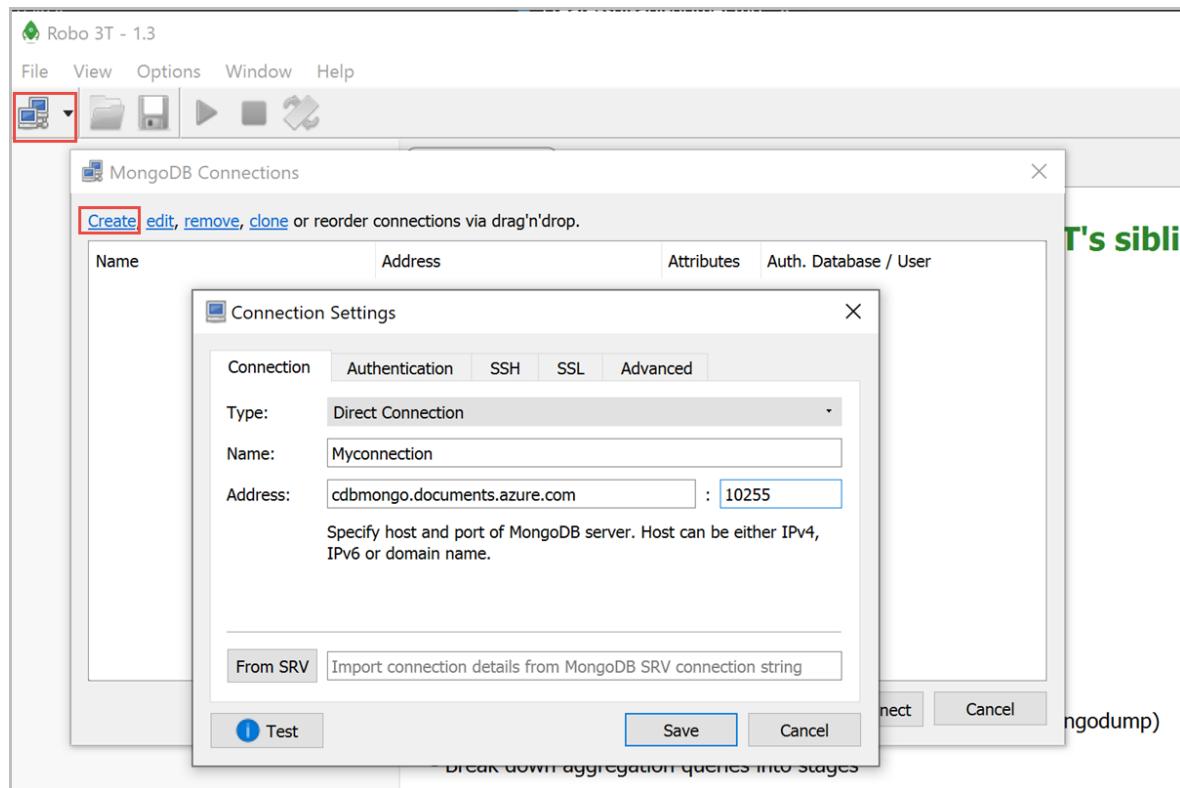
## Connect using Robo 3T

To add your Cosmos account to the Robo 3T connection manager, perform the following steps:

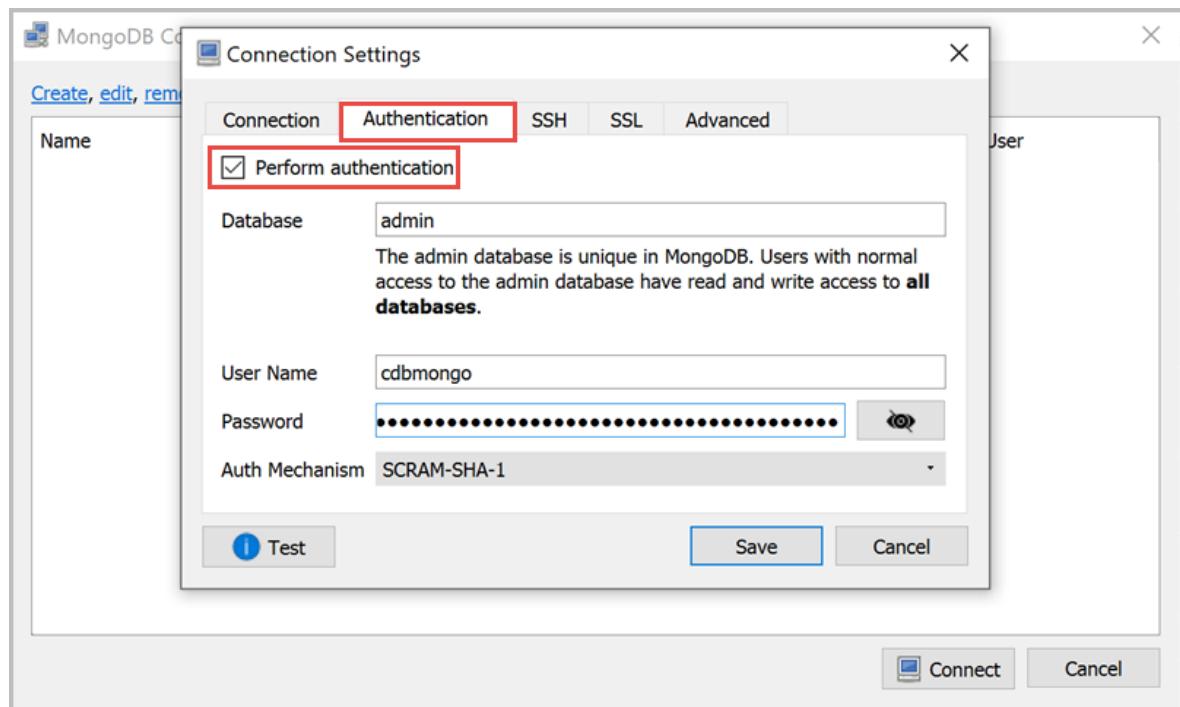
1. Retrieve the connection information for your Cosmos account configured with Azure Cosmos DB's API MongoDB using the instructions [here](#).



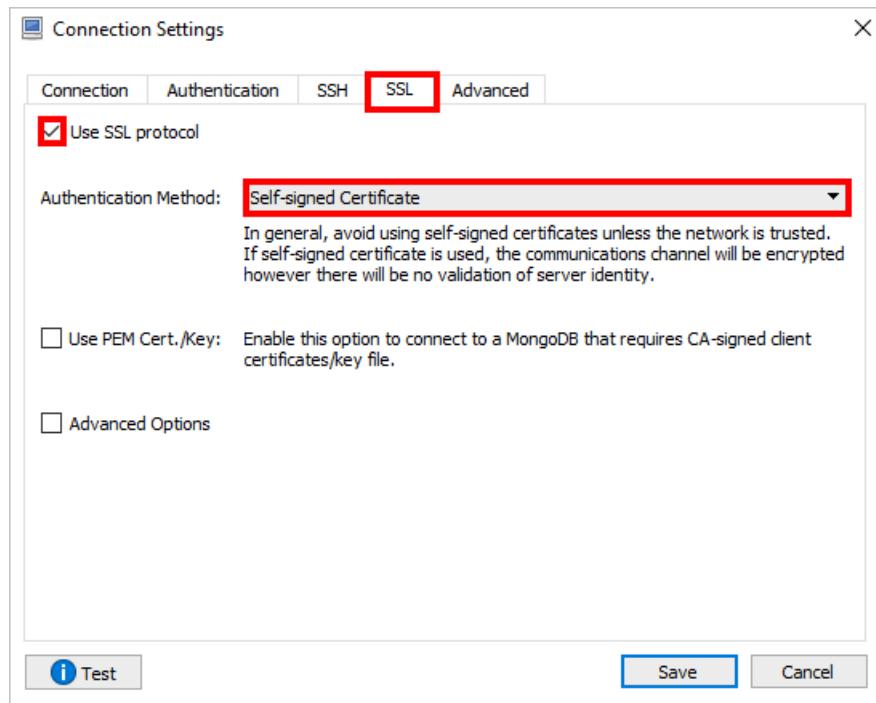
2. Run *Robomongo.exe*
3. Click the connection button under **File** to manage your connections. Then, click **Create** in the **MongoDB Connections** window, which will open up the **Connection Settings** window.
4. In the **Connection Settings** window, choose a name. Then, find the **Host** and **Port** 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

11/19/2019 • 9 minutes to read • [Edit Online](#)

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.

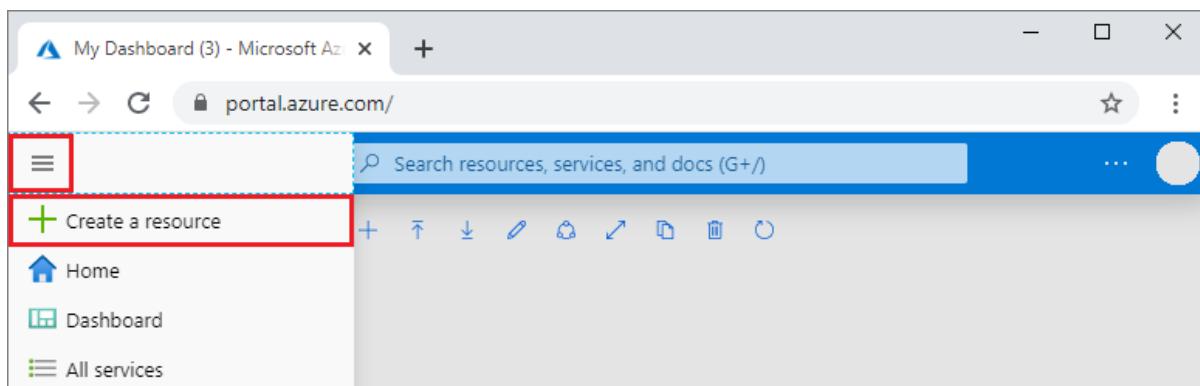
You can [Try Azure Cosmos DB for free](#) without an Azure subscription, free of charge and commitments. Or, you can use the [Azure Cosmos DB Emulator](#) with a URI of `https://localhost:8081`. For the key to use with the emulator, see [Authenticating requests](#).

[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**.

New

Search the Marketplace

Azure Marketplace [See all](#)

Featured [See all](#)

Get started		Azure SQL Managed Instance <a href="#">Quickstart tutorial</a>
Recently created		SQL Database <a href="#">Quickstart tutorial</a>
AI + Machine Learning		Azure Synapse Analytics (formerly SQL DW) <a href="#">Quickstart tutorial</a>
Analytics		Azure Database for MariaDB <a href="#">Learn more</a>
Blockchain		Azure Database for MySQL <a href="#">Quickstart tutorial</a>
Compute		Azure Database for PostgreSQL <a href="#">Quickstart tutorial</a>
Containers		Azure Cosmos DB <a href="#">Quickstart tutorial</a>
Databases		Azure Media Services <a href="#">Learn more</a>
Developer Tools		Internet of Things <a href="#">Quickstart tutorial</a>
DevOps		Media <a href="#">Quickstart tutorial</a>
Identity		Mixed Reality <a href="#">Learn more</a>
Integration		Azure Cosmos DB <a href="#">Quickstart tutorial</a>
Internet of Things		Azure Media Services <a href="#">Learn more</a>
Media		Internet of Things <a href="#">Quickstart tutorial</a>
Mixed Reality		Azure Cosmos DB <a href="#">Quickstart tutorial</a>

- 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 <b>Create new</b> . 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	<p>Enter a unique name to identify your Azure Cosmos DB account. Your account URI will be <i>mongo.cosmos.azure.com</i> appended to your unique account name.</p> <p>The account name can use only lowercase letters, numbers, and hyphens (-), and must be between 3 and 31 characters long.</p>

SETTING	VALUE	DESCRIPTION
API	Azure Cosmos DB for Mongo DB API	<p>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.</p> <p>Select <b>Azure Cosmos DB for Mongo DB API</b> because in this quickstart you are creating a collection that works with MongoDB.</p> <p><a href="#">Learn more about Azure Cosmos DB for MongoDB API.</a></p>
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.

Select **Review + Create**. You can skip the **Network** and **Tags** section.

Home > New > Create Azure Cosmos DB Account

## Create Azure Cosmos DB Account

[Basics](#) [Network](#) [Tags](#) [Review + create](#)

Azure Cosmos DB is a fully managed globally distributed, multi-model database service, transparently replicating your data across any number of Azure regions. You can elastically scale throughput and storage, and take advantage of fast, single-digit-millisecond data access using the API of your choice backed by 99.999 SLA. [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: Select existing... or Create new

**INSTANCE DETAILS**

\* Account Name: Enter account name

\* API: Azure Cosmos DB for MongoDB API (highlighted with a red box)

\* Location: Australia East

Geo-Redundancy: Enable (blue button) or Disable

Multi-region Writes: Enable (blue button) or Disable

[Review + create](#) [Previous](#) [Next: Network](#)

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.

Congratulations! Your Azure Cosmos DB for MongoDB API account is ready.

Now, let's connect your existing MongoDB app to it:

**Choose a platform**

.NET    Node.js    MongoDB Shell    Java    Python    Others

**1 Connect your existing MongoDB .NET app**

You can use your existing MongoDB .NET driver to work with Azure Cosmos DB. Make sure to enable SSL. Here is an example:

```
string connectionString =
    @"mongodb://rnagpal-mongo:4IropKPu9DmlqUw0iQBxLTCFtOufYwfe1m41SB7Xu7033QYq3VyRkCq5jNT2RNhCcxI
    MongoClientSettings settings = MongoClientSettings.FromUrl(
        new MongoUrl(connectionString)
    );
    settings.SslSettings =
        new SslSettings() { EnabledSslProtocols = SslProtocols.Tls12 };
    var mongoClient = new MongoClient(settings);
```

PRIMARY CONNECTION STRING  
mongodb://rnagpal-mongo:4IropKPu9DmlqUw0iQBxLTCFtOufYwfe1m41SB7Xu7033QYq3VyRkCq5jNT2RNhCcxF... [Copy](#)

For more details on configuring .NET driver to use SSL, follow [this article](#).

Questions? [Contact us](#)

**2 Learn More**

[Code Samples](#)  
[Migrating to Azure Cosmos DB](#)  
[Documentation](#)  
[Using Robomongo](#)  
[Using MongoChef](#)  
[Pricing](#)  
[Forum](#)

## 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.
```

```
COSMODDB_USER = "<Azure Cosmos account's user name>"
COSMOSDB_PASSWORD = "<Azure Cosmos account password>"
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.COSMODDB_USER,
    password: process.env.COSMOSDB_PASSWORD
  }
})
.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.

## Caveats to using Mongoose with Cosmos DB

For every model you create, Mongoose creates a new collection. However, given the per-collection billing model of Cosmos DB, it might not be the most cost-efficient way to go, if you've got multiple object models that are sparsely populated.

This walkthrough covers both models. We'll first cover the walkthrough on storing one type of data per collection. This is the defacto behavior for Mongoose.

Mongoose also has a concept called [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.

## One collection per object model

The default Mongoose behavior is to create a MongoDB collection every time you create an Object model. This section explores how to achieve this with Azure Cosmos DB's API for MongoDB. This method is recommended when you have object models with large amounts of data. This is the default operating model for Mongoose, so, you might be familiar with this, if you're familiar with Mongoose.

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: "Blackie" }  
    ],  
    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.

ID	DATABASE	THROUGHPUT (RU/S)
families	testdb	1000
vacationdestinations	testdb	1000

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: "Blackie" }
    ],
    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.

ID	DATABASE	THROUGHPUT (RU/S)
alldata	testdb	1000

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.

The screenshot shows the Azure Resource Groups blade. On the left, a list of resource groups is shown: 'myResourceGroupA', 'DefaultResourceGroup', 'DefaultResourceGroupA', and 'myResourceGroup'. The 'myResourceGroup' item is highlighted with a red box. On the right, the details for 'myResourceGroup' are displayed, including its subscription information ('Contoso Subscription'), subscription ID ('12345678-abcd-abcd-000000000000'), and tags ('Click here to add tags'). A secondary navigation bar at the top right includes 'Add', 'Edit columns', 'Delete resource group', 'Refresh', and 'Move'.

3. On the resource group **Overview** page, select **Delete resource group**.

The screenshot shows the 'myResourceGroup' Overview page. On the left, the navigation menu includes 'Overview', 'Activity log', 'Access control (IAM)', 'Tags', 'Events', 'Settings', 'Quickstart', 'Deployments', and 'Policies'. In the center, it shows the subscription information ('Contoso Subscription'), subscription ID ('12345678-abcd-abcd-000000000000'), and tags ('Click here to add tags'). A warning message at the top right says 'Are you sure you want to delete "myResourceGroup"?'. Below it, a note states 'Warning! Deleting the "myResourceGroup" resource group is irreversible. The action you're about to take can't be undone. Going further will delete this resource group and all the resources it contains.' A text input field says 'Please enter 'myresourcegroup' to confirm delete.' with 'myResourceGroup' typed in. At the bottom, a table titled 'AFFECTED RESOURCES' lists one resource: 'NAME' (mysqlapicosmosdb), 'TYPE' (Azure Cosmos DB account), and 'LOCATION' (West US). Two buttons at the bottom are 'Delete' (highlighted with a red box) and 'Cancel'.

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

12/13/2019 • 4 minutes to read • [Edit Online](#)

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:C2y6Djf5/R+ob0N8A7Cgv30VRDJlWEHLM+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 **SECONDARY PREFERRED** read preference mode. This setting directs read operations to an available READ region. If no READ region is available, then requests are directed to the WRITE region.

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

3/5/2019 • 2 minutes to read • [Edit Online](#)

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.

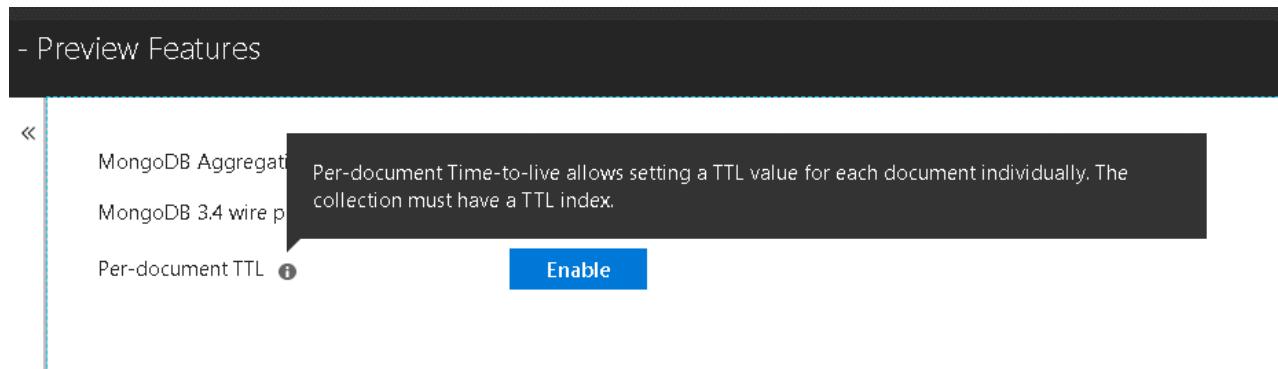
```
globaldb:PRIMARY> db.coll.insert({id:1, location: "Paris", ttl: 20.0})
globaldb:PRIMARY> db.coll.insert({id:1, location: "Paris", ttl: NumberInt(20)})
globaldb:PRIMARY> db.coll.insert({id:1, location: "Paris", ttl: NumberLong(20)})
```

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

2/19/2020 • 2 minutes to read • [Edit Online](#)

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.

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" and set it to "true" and set a TTL on the item, so that you can automatically delete it as well as track it.

```
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());
  }
}
```

The following example shows how to get changes to the items in 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".

```
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. 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.

## Error handling

The following error codes and messages are supported when using change streams:

- **HTTP error code 429** - 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](#)

# Indexing using Azure Cosmos DB's API for MongoDB

2/5/2020 • 4 minutes to read • [Edit Online](#)

Azure Cosmos DB's API for MongoDB leverages automatic index management capabilities of Cosmos DB. As a result, users have access to the default indexing policies of Cosmos DB. So, if no indexes have been defined by the user, or no indexes have been dropped, then all fields will be automatically indexed by default when inserted into a collection. For most scenarios, we recommend using the default indexing policy set on the account.

## Indexing for version 3.6

Accounts serving wire protocol version 3.6 provide a different default indexing policy than the policy provided by earlier versions. By default, only the `_id` field is indexed. To index additional fields, the user must apply the MongoDB index management commands. To apply a sort to a query, currently an index must be created on the fields used in the sort operation.

### Dropping the default indexes (3.6)

For accounts serving wire protocol version 3.6, the only default index is `_id`, which cannot be dropped.

### Creating a compound index (3.6)

True compound indexes are supported for accounts using the 3.6 wire protocol. The following command will create a compound index on the fields '`a`' and '`b`': `db.coll.createIndex({a:1,b:1})`

Compound indexes can be used to sort efficiently on multiple fields at once, such as:

```
db.coll.find().sort({a:1,b:1})
```

### Track the index progress

The 3.6 version of Azure Cosmos DB's API for MongoDB accounts support the `currentOp()` command to track index progress on a database instance. This command returns a document that contains information about the in-progress operations on a database instance. The `currentOp` command is used to track all the in-progress operations in native MongoDB whereas 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 the index progress:

- Get the index progress for a collection:

```
db.currentOp({"command.createIndexes": <collectionName>, "command.$db": <databaseName>})
```

- Get the index progress for all the collections in a database:

```
db.currentOp({"command.$db": <databaseName>})
```

- Get the index progress for all the databases and collections in an Azure Cosmos account:

```
db.currentOp({"command.createIndexes": { $exists : true } })
```

The index progress details contain percentage of progress for the current index operation. The following example

shows the output document format for different stages of index progress:

1. If the index operation on a 'foo' collection and 'bar' database that has 60 % indexing complete will have the following output document. `Inprog[0].progress.total` shows 100 as the target completion.

```
{  
    "inprog" : [  
        {  
            .....  
            "command" : {  
                "createIndexes" : foo  
                "indexes" : [ ],  
                "$db" : bar  
            },  
            "msg" : "Index Build (background) Index Build (background): 60 %",  
            "progress" : {  
                "done" : 60,  
                "total" : 100  
            },  
            .....  
        }  
    ],  
    "ok" : 1  
}
```

2. For an index operation that has just started on a 'foo' collection and 'bar' database, the output document may show 0% progress until it reaches to 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  
}
```

3. When the in-progress index operation completes, the output document shows empty inprog operations.

```
{  
    "inprog" : [],  
    "ok" : 1  
}
```

## Indexing for version 3.2

### Dropping the default indexes (3.2)

The following command can be used to drop the default indexes for a collection `coll`:

```
> db.coll.dropIndexes()
{ "_t" : "DropIndexesResponse", "ok" : 1, "nIndexesWas" : 3 }
```

### Creating a compound index (3.2)

Compound indexes hold references to multiple fields of a document. Logically, they are equivalent to creating multiple individual indexes per field. To take advantage of the optimizations provided by Cosmos DB indexing techniques, we recommend creating multiple individual indexes instead of a single (non-unique) compound index.

## Common Indexing Operations

The following operations are common for both accounts serving wire protocol version 3.6 and accounts serving earlier wire protocol versions.

### Creating unique indexes

[Unique indexes](#) are useful for enforcing that no two or more documents contain the same value for the indexed field(s).

#### IMPORTANT

Currently, 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, as per MongoDB behavior, creating a unique index requires providing the shard (partition) key. 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` (shard key is `university`) with a unique index on 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( { "student_id" : 1, "university" : 1 }, {unique:true})
{
    "_t" : "CreateIndexesResponse",
    "ok" : 1,
    "createdCollectionAutomatically" : false,
    "numIndexesBefore" : 3,
    "numIndexesAfter" : 4
}
```

In the above example, if `"university":1` clause was omitted, an error with the following message would be returned:

```
"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, a ["TTL index" \(time-to-live index\)](#) needs to be created. 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 will cause the deletion of any documents in `db.coll` 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.

## Migrating collections with indexes

Currently, creating unique indexes is possible only when the collection contains no documents. Popular MongoDB migration tools attempt to create the unique indexes after importing the data. To circumvent this issue, it is suggested that users manually create the corresponding collections and unique indexes, instead of allowing the migration tool (for `mongorestore` this behavior is achieved by using the `--noIndexRestore` flag in the command line).

## Next steps

- [Indexing in Azure Cosmos DB](#)
- [Expire data in Azure Cosmos DB automatically with time to live](#)

# Use MongoDB extension commands to manage data stored in Azure Cosmos DB's API for MongoDB

12/13/2019 • 6 minutes to read • [Edit Online](#)

Azure Cosmos DB is Microsoft's globally distributed multi-model database service. 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](#).

By using the Azure Cosmos DB's API for MongoDB, you can enjoy the benefits of 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.

## MongoDB protocol support

By default, the Azure Cosmos DB's API for MongoDB is compatible with MongoDB server version 3.2, for more details, see [supported features and syntax](#). The features or query operators added in MongoDB version 3.4 are currently available as a preview in the Azure Cosmos DB's API for MongoDB. The following extension commands support Azure Cosmos DB specific functionality when performing CRUD operations on the data stored in Azure Cosmos DB's API for MongoDB:

- [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 is used from the databases context against which the command is executed. The format of the CreateDatabase command is as follows:

```
{  
  customAction: "CreateDatabase",  
  offerThroughput: <Throughput that you want to provision on the database>  
}
```

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.

## Output

Returns a default custom command response. See the [default output](#) of custom command for the parameters in the output.

## Examples

### Create a database

To create a database named "test", use the following command:

```
use test
db.runCommand({customAction: "CreateDatabase"});
```

### Create a database with throughput

To create a database named "test" and provisioned throughput of 1000 RUs, use the following command:

```
use test
db.runCommand({customAction: "CreateDatabase", offerThroughput: 1000});
```

## Update database

The update database extension command updates the properties associated with the specified database. Currently, you can only update the "offerThroughput" property.

```
{
  customAction: "UpdateDatabase",
  offerThroughput: <New throughput that you want to provision on the database>
}
```

The following table describes the parameters within the command:

FIELD	TYPE	DESCRIPTION
customAction	string	Name of the custom command. Must be "UpdateDatabase".
offerThroughput	int	New provisioned throughput that you want to set on the database.

## 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 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});
```

## 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
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. This is an optional response parameter.

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"});
```

## Create collection

The create collection extension command creates a new MongoDB collection. The database name is used from the databases context against which the command is executed. The format of the CreateCollection command is as follows:

```
{  
    customAction: "CreateCollection",  
    collection: <Collection Name>,  
    offerThroughput: <Throughput that you want to provision on the collection>,  
    shardKey: <Shard key path>  
}
```

The following table describes the parameters within the command:

FIELD	TYPE	DESCRIPTION
customAction	string	Name of the custom command. Must be "CreateCollection"
collection	string	Name of the collection
offerThroughput	int	Provisioned Throughput to set on the database. It's an Optional parameter
shardKey	string	Shard Key path to create a sharded collection. It's an Optional parameter

## Output

Returns a default custom command response. See the [default output](#) of custom command for the parameters in the output.

## Examples

### 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});
```

### Create a sharded collection

To create a sharded collection with name "testCollection" and provisioned throughput of 1000 RUs, use the following command:

```
use test
db.runCommand({customAction: "CreateCollection", collection: "testCollection", offerThroughput: 1000, shardKey: "a.b" });
```

## 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: <New throughput that you want to provision on 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.

FIELD	TYPE	DESCRIPTION
offerThroughput	int	Provisioned throughput to set on the collection.

## 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.

FIELD	TYPE	DESCRIPTION
<code>provisionedThroughput</code>	<code>int</code>	Provisioned Throughput to set on the collection. This is an optional response parameter.

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"});
```

## Default output of a custom command

If not specified, a custom response contains a document with the following fields:

FIELD	TYPE	DESCRIPTION
<code>ok</code>	<code>int</code>	Status of response. 1 == success. 0 == failure.
<code>code</code>	<code>int</code>	Only returned when the command failed (i.e. <code>ok == 0</code> ). Contains the MongoDB error code. This is an optional response parameter.
<code>errMsg</code>	<code>string</code>	Only returned when the command failed (i.e. <code>ok == 0</code> ). Contains a user-friendly error message. This is an optional response parameter.

## 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

2/24/2020 • 4 minutes to read • [Edit Online](#)

This article describes how to perform different operations to automate management of your Azure Cosmos DB accounts, databases and containers using Azure Resource Manager templates. 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.

## Create Azure Cosmos DB API for MongoDB account, database and collection

Create Azure Cosmos DB resources using an Azure Resource Manager template. This template will create an Azure Cosmos account for MongoDB API with two collections that share 400 RU/s throughput at the database level. Copy the template and deploy as shown below or visit [Azure Quickstart Gallery](#) and deploy from the Azure portal. You can also download the template to your local computer or create a new template and specify the local path with the `--template-file` parameter.

## NOTE

Account names must be lowercase and 44 or fewer characters. To update RU/s, resubmit the template with updated throughput property values.

Currently you can only create 3.2 version (that is, accounts using the endpoint in the format `*.documents.azure.com`) of Azure Cosmos DB's API for MongoDB accounts by using PowerShell, CLI, and Resource Manager templates. To create 3.6 version of accounts, use Azure portal instead.

```
        "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."
    }
},
"multipleWriteLocations": {
    "type": "bool",
    "defaultValue": false,
    "allowedValues": [ true, false ],
    "metadata": {
        "description": "Enable multi-master to make all regions writable."
    }
},
"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": "[toLowerCase(parameters('accountName'))]",
    "region": "[parameters('region')]"
}
```

```

"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": "2019-08-01",
    "location": "[parameters('location')]",
    "kind": "MongoDB",
    "properties": {
        "consistencyPolicy": "[variables('consistencyPolicy')[parameters('defaultConsistencyLevel')]]",
        "locations": "[variables('locations')]",
        "databaseAccountOfferType": "Standard",
        "enableMultipleWriteLocations": "[parameters('multipleWriteLocations')]"
    }
},
{
    "type": "Microsoft.DocumentDB/databaseAccounts/mongodbDatabases",
    "name": "[concat(variables('accountName'), '/', parameters('databaseName'))]",
    "apiVersion": "2019-08-01",
    "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'))]",
    "apiVersion": "2019-08-01",
    "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": "2019-08-01",
    "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>"
            }
        }
    }
}
]
}

```

## Deploy via the Azure CLI

To deploy the Azure Resource Manager template using the Azure CLI, **Copy** the script and select **Try it** to open Azure Cloud Shell. To paste the script, right-click the shell, and then select **Paste**:

```
read -p 'Enter the Resource Group name: ' resourceGroupName
read -p 'Enter the location (i.e. westus2): ' location
read -p 'Enter the account name: ' accountName
read -p 'Enter the primary region (i.e. westus2): ' primaryRegion
read -p 'Enter the secondary region (i.e. eastus2): ' secondaryRegion
read -p 'Enter the database name: ' databaseName
read -p 'Enter the first collection name: ' collection1Name
read -p 'Enter the second collection name: ' collection2Name

az group create --name $resourceGroupName --location $location
az group deployment create --resource-group $resourceGroupName \
    --template-uri https://raw.githubusercontent.com/azure/azure-quickstart-templates/master/101-cosmosdb-
mongodb/azuredeploy.json \
    --parameters accountName=$accountName primaryRegion=$primaryRegion secondaryRegion=$secondaryRegion \
    databaseName=$databaseName collection1Name=$collection1Name collection2Name=$collection2Name

az cosmosdb show --resource-group $resourceGroupName --name accountName --output tsv
```

The `az cosmosdb show` command shows the newly created Azure Cosmos account after it has been provisioned. If you choose to use a locally installed version of the Azure CLI instead of using Cloud Shell, see the [Azure CLI](#) article.

## 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 MongoDB API

2/21/2020 • 2 minutes to read • [Edit Online](#)

The following table includes links to sample Azure PowerShell scripts for Azure Cosmos DB for MongoDB API.

## NOTE

Currently you can only create 3.2 version (that is, accounts using the endpoint in the format `*.documents.azure.com`) of Azure Cosmos DB's API for MongoDB accounts by using PowerShell, CLI, and Resource Manager templates. To create 3.6 version of accounts, use Azure portal instead.

<a href="#">Create an account, database and collection</a>	Creates an Azure Cosmos account, database and collection.
<a href="#">List or get databases or collections</a>	List or get database or collection.
<a href="#">Get RU/s</a>	Get RU/s for a database or collection.
<a href="#">Update RU/s</a>	Update RU/s for a database or collection.
<a href="#">Update an account or add a region</a>	Add a region to a Cosmos account. Can also be used to modify other account properties but these must be separate from changes to regions.
<a href="#">Change failover priority or trigger failover</a>	Change the regional failover priority of an Azure Cosmos account or trigger a manual failover.
<a href="#">Account keys or connection strings</a>	Get primary and secondary keys, connection strings or regenerate an account key of an Azure Cosmos account.
<a href="#">Create a Cosmos Account with IP Firewall</a>	Create an Azure Cosmos account with IP Firewall enabled.

# Azure CLI samples for Azure Cosmos DB MongoDB API

2/21/2020 • 2 minutes to read • [Edit Online](#)

The following table includes links to sample Azure CLI scripts for Azure Cosmos DB MongoDB API. Reference pages for all Azure Cosmos DB CLI commands are available in the [Azure CLI Reference](#). All Azure Cosmos DB CLI script samples can be found in the [Azure Cosmos DB CLI GitHub Repository](#).

## NOTE

Currently you can only create 3.2 version (that is, accounts using the endpoint in the format `*.documents.azure.com`) of Azure Cosmos DB's API for MongoDB accounts by using PowerShell, CLI, and Resource Manager templates. To create 3.6 version of accounts, use Azure portal instead.

<a href="#">Create an Azure Cosmos account, database and collection</a>	Creates an Azure Cosmos DB account, database, and collection for MongoDB API.
<a href="#">Change throughput</a>	Update RU/s on a database and collection.
<a href="#">Add or failover regions</a>	Add a region, change failover priority, trigger a manual failover.
<a href="#">Account keys and connection strings</a>	List account keys, read-only keys, regenerate keys and list connection strings.
<a href="#">Secure with IP firewall</a>	Create a Cosmos account with IP firewall configured.
<a href="#">Secure new account with service endpoints</a>	Create a Cosmos account and secure with service-endpoints.
<a href="#">Secure existing account with service endpoints</a>	Update a Cosmos account to secure with service-endpoints when the subnet is eventually configured.

# Troubleshoot common issues in Azure Cosmos DB's API for MongoDB

1/14/2020 • 2 minutes to read • [Edit Online](#)

Azure Cosmos DB implements the wire protocols of common NoSQL databases, including MongoDB. Due to the wire protocol implementation, you can transparently interact with Azure Cosmos DB by using the existing client SDKs, drivers, and tools that work with NoSQL databases. Azure Cosmos DB does not use any source code of the databases for providing wire-compatible APIs for any of the NoSQL databases. Any MongoDB client driver that understands the wire protocol versions can connect to Azure Cosmos DB.

While Azure Cosmos DB's API for MongoDB is compatible with 3.2 version of the MongoDB's wire protocol (the query operators and features added in version 3.4 are currently available as a preview), there are some custom error codes that correspond to Azure Cosmos DB specific errors. This article explains different errors, error codes, and the steps to resolve those errors.

## Common errors and solutions

ERROR	CODE	DESCRIPTION	SOLUTION
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 <a href="#">Azure portal</a> . Example: <code>db.getCollection('users').aggregate([{\$name: "Andy"}], {\$sort: {age: -1}}]</code>
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 <code>appName=@accountName@</code> at the end of your Cosmos DB's API for MongoDB connection string, where <b>accountName</b> is your Cosmos DB account name.

## Next steps

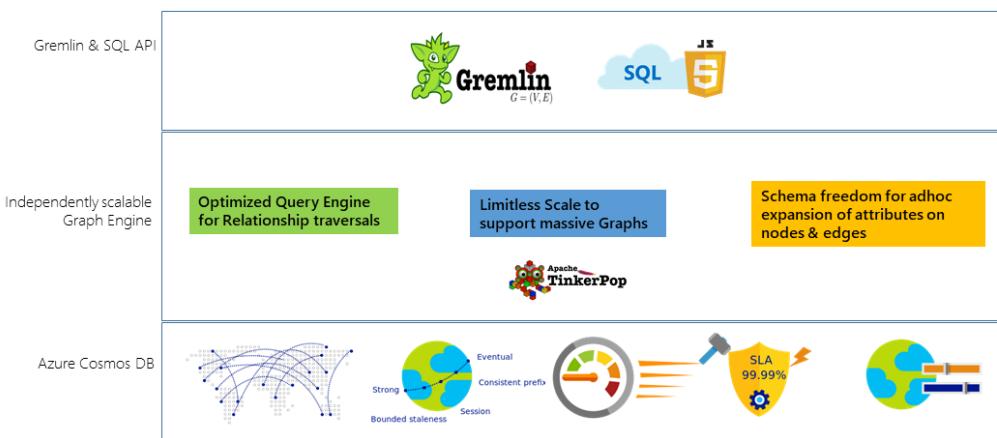
- Learn how to [use Studio 3T](#) with Azure Cosmos DB's API for MongoDB.
- Learn how to [use Robo 3T](#) with Azure Cosmos DB's API for MongoDB.
- Explore MongoDB [samples](#) with Azure Cosmos DB's API for MongoDB.

# Introduction to Azure Cosmos DB: Gremlin API

12/26/2019 • 7 minutes to read • [Edit Online](#)

Azure Cosmos DB is 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. The Azure Cosmos DB Gremlin API is used to store and operate with graph data on a fully managed database service designed for any scale.

## Azure Cosmos DB – Graph API PaaS



This article provides an overview of the Azure Cosmos DB Gremlin API and explains how you can use it 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 based on the [Apache TinkerPop](#) graph database standard, and 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.

## Features of Azure Cosmos DB graph database

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. These guarantees allow 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 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 can 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

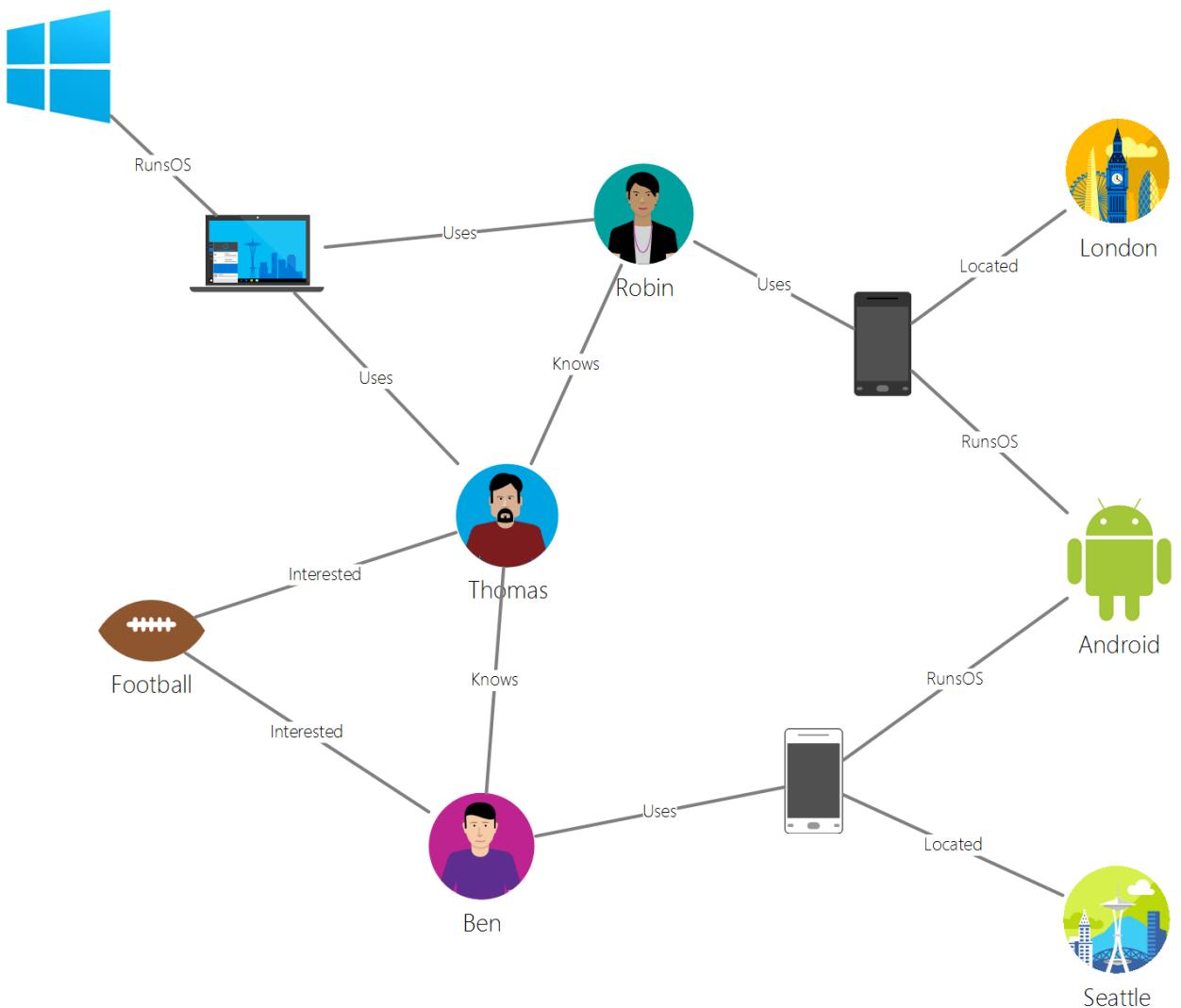
A property [graph](#) is a structure that's composed of [vertices](#) and [edges](#). Both objects can have an arbitrary number of key-value pairs as properties.

- **Vertices** - Vertices denote discrete entities, such as a person, a place, or an event.
- **Edges** - 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.

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.

### Gremlin 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 (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

We represent the relationships between these entities via the following *edge* types/labels:

- **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

Let's run some operations against this graph using the [Gremlin Console](#). You can also perform these operations using Gremlin drivers in the platform of your choice (Java, Nodejs, Python, or .NET). Before we look at what's supported in Azure Cosmos DB, let's look at a few examples to get familiar with the syntax.

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('runssos').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

12/26/2019 • 6 minutes to read • [Edit Online](#)

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.

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	<a href="#">Gremlin.NET on GitHub</a>	<a href="#">Create Graph using .NET</a>	3.4.0-RC2
Java	<a href="#">Gremlin JavaDoc</a>	<a href="#">Create Graph using Java</a>	3.2.0+
Node.js	<a href="#">Gremlin-JavaScript on GitHub</a>	<a href="#">Create Graph using Node.js</a>	3.3.4+
Python	<a href="#">Gremlin-Python on GitHub</a>	<a href="#">Create Graph using Python</a>	3.2.7
PHP	<a href="#">Gremlin-PHP on GitHub</a>	<a href="#">Create Graph using PHP</a>	3.1.0
Gremlin console	<a href="#">TinkerPop docs</a>	<a href="#">Create Graph using Gremlin Console</a>	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

CATEGORY	AZURE COSMOS DB IMPLEMENTATION	NOTES
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: GraphSON

Azure Cosmos DB uses the [GraphSON format](#) when returning results from Gremlin operations. Azure Cosmos DB currently supports "GraphSONv2" version. GraphSON is the Gremlin standard format for representing vertices, edges, and properties (single and multi-valued properties) using JSON.

For example, the following snippet shows a GraphSON 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 GraphSON for vertices are described below:

PROPERTY	DESCRIPTION
<code>id</code>	The ID for the vertex. Must be unique (in combination with the value of <code>_partition</code> if applicable). If no value is provided, it will be automatically supplied with a GUID
<code>label</code>	The label of the vertex. This property is used to describe the entity type.
<code>type</code>	Used to distinguish vertices from non-graph documents
<code>properties</code>	Bag of user-defined properties associated with the vertex. Each property can have multiple values.
<code>_partition</code>	The partition key of the vertex. Used for <a href="#">graph partitioning</a> .
<code>outE</code>	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
<code>id</code>	The ID for the edge. Must be unique (in combination with the value of <code>_partition</code> if applicable)
<code>label</code>	The label of the edge. This property is optional, and used to describe the relationship type.
<code>inV</code>	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.
<code>properties</code>	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
<code>value</code>	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
<code>addE</code>	Adds an edge between two vertices	<a href="#">addE step</a>
<code>addV</code>	Adds a vertex to the graph	<a href="#">addV step</a>
<code>and</code>	Ensures that all the traversals return a value	<a href="#">and step</a>
<code>as</code>	A step modulator to assign a variable to the output of a step	<a href="#">as step</a>
<code>by</code>	A step modulator used with <code>group</code> and <code>order</code>	<a href="#">by step</a>
<code>coalesce</code>	Returns the first traversal that returns a result	<a href="#">coalesce step</a>
<code>constant</code>	Returns a constant value. Used with <code>coalesce</code>	<a href="#">constant step</a>
<code>count</code>	Returns the count from the traversal	<a href="#">count step</a>
<code>dedup</code>	Returns the values with the duplicates removed	<a href="#">dedup step</a>

STEP	DESCRIPTION	TINKERPOP 3.2 DOCUMENTATION
<code>drop</code>	Drops the values (vertex/edge)	<a href="#">drop step</a>
<code>executionProfile</code>	Creates a description of all operations generated by the executed Gremlin step	<a href="#">executionProfile step</a>
<code>fold</code>	Acts as a barrier that computes the aggregate of results	<a href="#">fold step</a>
<code>group</code>	Groups the values based on the labels specified	<a href="#">group step</a>
<code>has</code>	Used to filter properties, vertices, and edges. Supports <code>hasLabel</code> , <code>hasId</code> , <code>hasNot</code> , and <code>has</code> variants.	<a href="#">has step</a>
<code>inject</code>	Inject values into a stream	<a href="#">inject step</a>
<code>is</code>	Used to perform a filter using a boolean expression	<a href="#">is step</a>
<code>limit</code>	Used to limit number of items in the traversal	<a href="#">limit step</a>
<code>local</code>	Local wraps a section of a traversal, similar to a subquery	<a href="#">local step</a>
<code>not</code>	Used to produce the negation of a filter	<a href="#">not step</a>
<code>optional</code>	Returns the result of the specified traversal if it yields a result else it returns the calling element	<a href="#">optional step</a>
<code>or</code>	Ensures at least one of the traversals returns a value	<a href="#">or step</a>
<code>order</code>	Returns results in the specified sort order	<a href="#">order step</a>
<code>path</code>	Returns the full path of the traversal	<a href="#">path step</a>
<code>project</code>	Projects the properties as a Map	<a href="#">project step</a>
<code>properties</code>	Returns the properties for the specified labels	<a href="#">properties step</a>
<code>range</code>	Filters to the specified range of values	<a href="#">range step</a>
<code>repeat</code>	Repeats the step for the specified number of times. Used for looping	<a href="#">repeat step</a>

STEP	DESCRIPTION	TINKERPOP 3.2 DOCUMENTATION
<code>sample</code>	Used to sample results from the traversal	<a href="#">sample step</a>
<code>select</code>	Used to project results from the traversal	<a href="#">select step</a>
<code>store</code>	Used for non-blocking aggregates from the traversal	<a href="#">store step</a>
<code>TextP.startsWith(string)</code>	String filtering function. This function is used as a predicate for the <code>has()</code> step to match a property with the beginning of a given string	<a href="#">TextP predicates</a>
<code>TextP.endsWith(string)</code>	String filtering function. This function is used as a predicate for the <code>has()</code> step to match a property with the ending of a given string	<a href="#">TextP predicates</a>
<code>TextP.contains(string)</code>	String filtering function. This function is used as a predicate for the <code>has()</code> step to match a property with the contents of a given string	<a href="#">TextP predicates</a>
<code>TextP.notStartingWith(string)</code>	String filtering function. This function is used as a predicate for the <code>has()</code> step to match a property that doesn't start with a given string	<a href="#">TextP predicates</a>
<code>TextP.notEndingWith(string)</code>	String filtering function. This function is used as a predicate for the <code>has()</code> step to match a property that doesn't end with a given string	<a href="#">TextP predicates</a>
<code>TextP.notContaining(string)</code>	String filtering function. This function is used as a predicate for the <code>has()</code> step to match a property that doesn't contain a given string	<a href="#">TextP predicates</a>
<code>tree</code>	Aggregate paths from a vertex into a tree	<a href="#">tree step</a>
<code>unfold</code>	Unroll an iterator as a step	<a href="#">unfold step</a>
<code>union</code>	Merge results from multiple traversals	<a href="#">union step</a>
<code>v</code>	Includes the steps necessary for traversals between vertices and edges <code>V</code> , <code>E</code> , <code>out</code> , <code>in</code> , <code>both</code> , <code>outE</code> , <code>inE</code> , <code>bothE</code> , <code>outV</code> , <code>inV</code> , <code>bothV</code> , and <code>otherV</code> for	<a href="#">vertex steps</a>

STEP	DESCRIPTION	TINKERPOP 3.2 DOCUMENTATION
<code>where</code>	Used to filter results from the traversal. Supports <code>eq</code> , <code>neq</code> , <code>lt</code> , <code>lte</code> , <code>gt</code> , <code>gte</code> , and <code>between</code> operators	<a href="#">where step</a>

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](#).

## Next steps

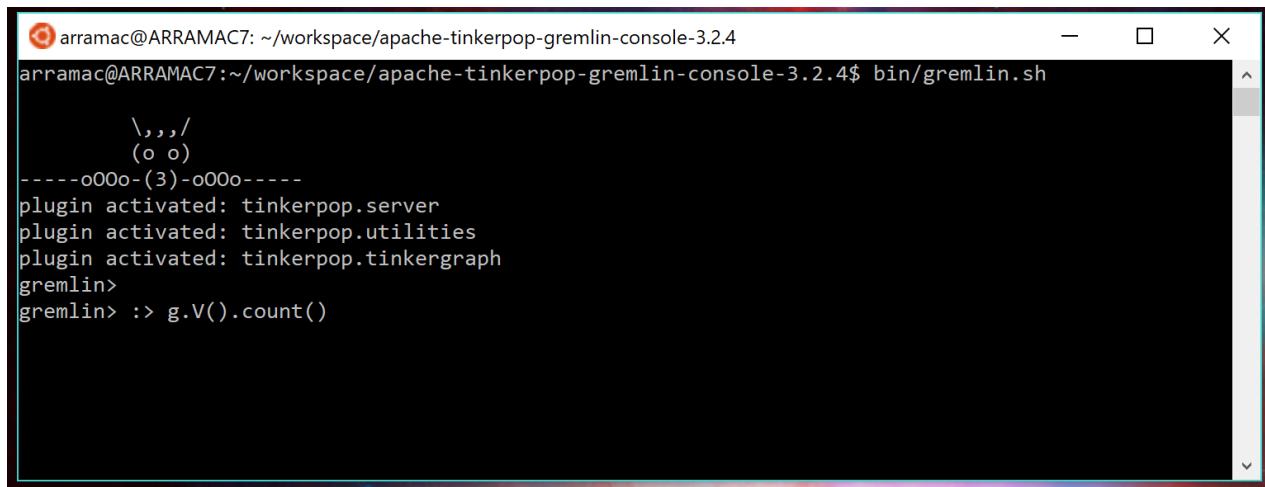
- Get started building a graph application [using our SDKs](#)
- Learn more about [graph support](#) in Azure Cosmos DB

# Quickstart: Create, query, and traverse an Azure Cosmos DB graph database using the Gremlin console

2/10/2020 • 8 minutes to read • [Edit Online](#)

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.



```
arramac@ARRAMAC7: ~/workspace/apache-tinkerpop-gremlin-console-3.2.4
arramac@ARRAMAC7:~/workspace/apache-tinkerpop-gremlin-console-3.2.4$ bin/gremlin.sh
      \,,,/ 
      (o o)
-----o00o-(3)-o00o-----
plugin activated: tinkerpop.server
plugin activated: tinkerpop.utilities
plugin activated: tinkerpop.tinkergraph
gremlin>
gremlin> :> g.V().count()
```

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**.

The screenshot shows the Microsoft Azure portal dashboard. At the top left, there's a navigation bar with icons for Home, Dashboard, and All services. A search bar at the top right says 'Search resources, services, and docs (G+)'. Below the navigation bar, there's a large 'Create a resource' button with a green plus sign icon, which is also highlighted with a red box. To the right of the button are several small blue icons for managing resources.

3. On the **New** page, select **Databases > Azure Cosmos DB**.

The screenshot shows the 'New' page in the Azure Marketplace. On the left, there's a sidebar with categories like Get started, Recently created, AI + Machine Learning, Analytics, Blockchain, Compute, Containers, Databases (which is highlighted with a blue box), Developer Tools, DevOps, Identity, Integration, Internet of Things, Media, and Mixed Reality. On the right, there's a 'Featured' section with cards for Azure SQL Managed Instance, SQL Database, Azure Synapse Analytics (formerly SQL DW), Azure Database for MariaDB, Azure Database for MySQL, Azure Database for PostgreSQL, and Azure Cosmos DB. The 'Azure Cosmos DB' card is highlighted with a red box.

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 <b>Create new</b> . 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	<p>Enter a unique name to identify your Azure Cosmos DB account. Your account URI will be <i>gremlin.azure.com</i> appended to your unique account name.</p> <p>The account name can use only lowercase letters, numbers, and hyphens (-), and must be between 3 and 31 characters long.</p>
API	Gremlin (graph)	<p>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.</p> <p>Select <b>Gremlin (graph)</b>, because in this quickstart you are creating a table that works with the Gremlin API.</p> <p><a href="#">Learn more about the Gremlin API</a>.</p>
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.

Select **Review+Create**. You can skip the **Network** and **Tags** section.

**Create Azure Cosmos DB Account**

**Basics** Networking 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 \*** A Subscription

**Resource Group \*** Select existing... Create new

**Instance Details**

**Account Name \*** Enter account name

**API \*** Gremlin (graph)

**Apache Spark** Notebooks (preview) Notebooks with Apache Spark (preview) None Sign up for Apache Spark preview

**Location \*** (US) West US

**Geo-Redundancy** Enable Disable

**Multi-region Writes** Enable Disable

\*Up to 33% off multi-region writes is available to qualifying new accounts only. Accounts must be created between December 1, 2019 and February 29, 2020. 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

- The account creation takes a few minutes. Wait for the portal to display the **Congratulations! Your Azure Cosmos DB account was created** page.

**cosmos-db-quickstart - Quick start**

Congratulations! Your Azure Cosmos DB account was created.

Now, let's connect to it using a sample app:

**Choose a platform**

- .NET
- Gremlin Console
- Guided Gremlin Tour

**1 Add a graph**

In Azure Cosmos DB, you can create containers to store graph elements (vertices and edges).

**Create 'Persons' container**

Create 'Persons' graph container with 10GB storage capacity and 400 Request Units per second (RU/s) throughput capacity, for up to 400 reads/sec. Estimated hourly bill: \$0.033 USD. Estimated hourly bill: \$0.033 USD

**2 Download and run your .NET app**

Once graph is created, download a sample .NET app connected to it, extract, build and run.

**Download**

## Add a graph

You can now use the Data Explorer tool in the Azure portal to create a graph database.

- Select **Data Explorer > New Graph**.

The **Add Graph** area is displayed on the far right, you may need to scroll right to see it.

The screenshot shows the Azure portal interface for a Cosmos DB account named 'cdbgraph1'. On the left, there's a sidebar with various management links like Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Quick start, Notifications, and Data Explorer, which is currently selected and highlighted with a red box. The main area is titled 'GREMLIN API' and has a 'New Database' and 'New Graph' button. A modal window titled 'Add Graph' is open, prompting for settings: 'Database id' (radio buttons for 'Create new' or 'Use existing', with 'sample-database' selected), 'Provision database throughput' (checkbox checked), 'Throughput (400 - 100,000 RU/s)' (input field set to 1000), 'Estimated spend (USD): \$0.032 hourly / \$0.77 daily (1 region, 400RU/s, \$0.00008/RU)', 'Graph id' (input field set to 'sample-graph'), 'Partition key' (input field set to '/pk'), and a note about 'My partition key is larger than 100 bytes'. At the bottom right of the modal is an 'OK' button.

2. In the **Add graph** page, enter the settings for the new graph.

SETTING	SUGGESTED VALUE	DESCRIPTION
Database ID	sample-database	Enter <i>sample-database</i> 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 <i>sample-graph</i> 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 <a href="#">Graph Data Partitioning article</a> .

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 <b>Gremlin URI</b> 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 <b>not</b> 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 SSL.
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:

The screenshot shows the Microsoft Azure portal interface. The URL in the address bar is portal.azure.com. The main title is "Microsoft Azure" followed by "Azure Cosmos DB > testgraphacct". On the left, there's a sidebar with icons for Add Graph, Refresh, Move, Delete Account, Data Explorer, Activity log, Access control (IAM), Tags, and Diagnose and solve problems. The main content area is titled "testgraphacct - Azure Cosmos DB account". It has a search bar and a navigation menu with "Overview", "Activity log", "Access control (IAM)", "Tags", and "Diagnose and solve problems". Below the menu, there's a "Essentials" section with details: Resource group (change) testgraphacct, Subscription (change) Visual Studio Ultimate with MSDN, Read Locations North Central US, Write Location North Central US, Status --, Subscription ID 2c6090c6-f494-488e-a9c6-4f1b0edfd382, .NET SDK URI https://testgraphacct.documents.azure.co..., and Gremlin URI testgraphacct.graphs.azure.com:443/. The "Gremlin URI" line is specifically highlighted with a red box.

For the password value, copy the **Primary key** from the **Keys** page:

The screenshot shows the Microsoft Azure portal interface. The top navigation bar includes the Azure logo, a back arrow, a forward arrow, a refresh icon, a lock icon, the URL 'portal.azure.com', and various navigation icons. The main title is 'Microsoft Azure testgraphacct - Keys'. On the left, there's a sidebar with a list of options: Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Quick start, Data Explorer (Preview), SETTINGS (Replicate data globally, Default consistency, Firewall), and Keys (which is highlighted with a red box). The main content area is titled 'testgraphacct - Keys' and 'Azure Cosmos DB account'. It shows tabs for 'Read-write Keys' (selected) and 'Read-only Keys'. Under 'Read-write Keys', there are sections for 'URI' (https://testgraphacct.documents.azure.com:443/), 'PRIMARY KEY' (<primary key>), 'SECONDARY KEY' (<secondary key>), 'PRIMARY CONNECTION STRING' (AccountEndpoint=https://testgraphacct.documents.azure.com:443/;AccountKey=Pam...), and 'SECONDARY CONNECTION STRING' (AccountEndpoint=https://testgraphacct.documents.azure.com:443/;AccountKey=zCY...). Each string has a copy icon (a clipboard with a pencil) and a refresh icon.

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 [].

1. In your terminal, run `bin/gremlin.bat` or `bin/gremlin.sh` to start the [Gremlin Console](#).
2. 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.

3. 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-a442-26fcdb042ce,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-a106-0ddff18719d15,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()
```

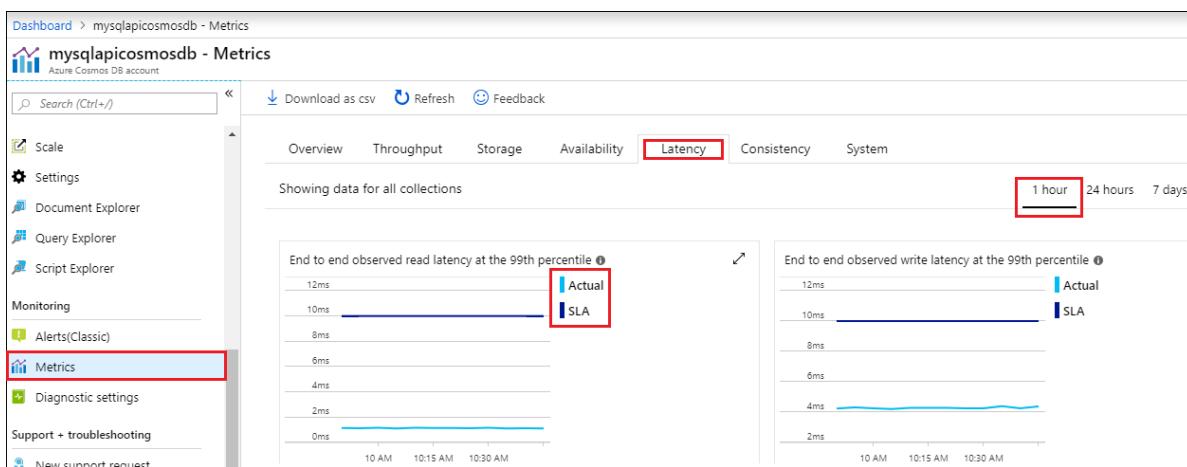
Congratulations! You've completed this Azure Cosmos DB: Gremlin API tutorial!

## 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.

The screenshot shows the 'Resource groups' blade in the Azure portal. It lists several resource groups, with 'myResourceGroup' selected and highlighted with a red box. The right pane shows the 'Overview' details for 'myResourceGroup', including the subscription (Contoso Subscription), subscription ID (12345678-abcd-abcd-000000000000), and tags. The 'myResourceGroup' blade also has a 'Delete resource group' button.

3. On the resource group **Overview** page, select **Delete resource group**.

The screenshot shows a confirmation dialog box asking 'Are you sure you want to delete "myResourceGroup"?'. It contains a warning message about the irreversibility of the action and a text input field asking for confirmation with the name 'myResourceGroup'. Below the dialog, the 'myResourceGroup' blade shows a table of affected resources, listing 'mysqlapicosmosdb' as an Azure Cosmos DB account in West US.

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

2/23/2020 • 8 minutes to read • [Edit Online](#)

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](#).

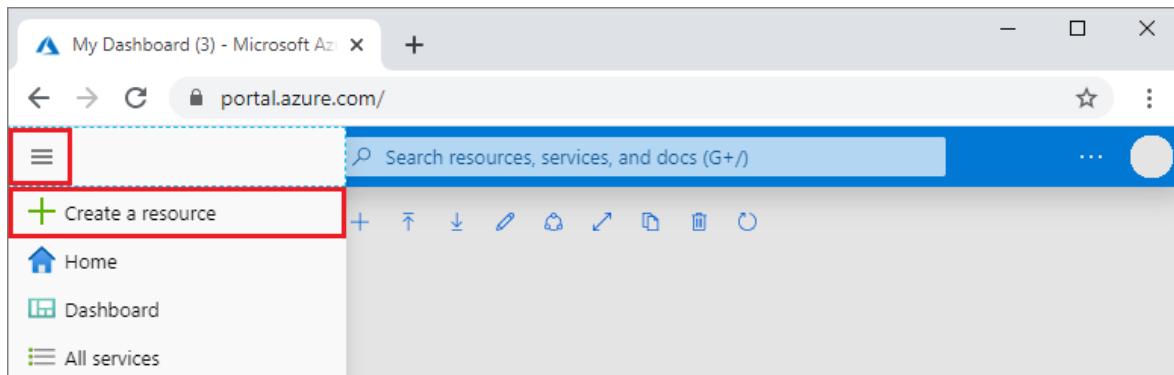
## Prerequisites

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**.

New

Search the Marketplace

Azure Marketplace [See all](#)

Get started  
Recently created  
AI + Machine Learning  
Analytics  
Blockchain  
Compute  
Containers  
**Databases**  
Developer Tools  
DevOps  
Identity  
Integration  
Internet of Things  
Media  
Mixed Reality

Featured [See all](#)

 Azure SQL Managed Instance  
[Quickstart tutorial](#)

 SQL Database  
[Quickstart tutorial](#)

 Azure Synapse Analytics (formerly SQL DW)  
[Quickstart tutorial](#)

 Azure Database for MariaDB  
[Learn more](#)

 Azure Database for MySQL  
[Quickstart tutorial](#)

 Azure Database for PostgreSQL  
[Quickstart tutorial](#)

 Azure Cosmos DB  
[Quickstart tutorial](#)

- 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 <b>Create new</b> . 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	<p>Enter a unique name to identify your Azure Cosmos DB account. Your account URI will be <i>gremlin.azure.com</i> appended to your unique account name.</p> <p>The account name can use only lowercase letters, numbers, and hyphens (-), and must be between 3 and 31 characters long.</p>

SETTING	VALUE	DESCRIPTION
API	Gremlin (graph)	<p>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.</p> <p>Select <b>Gremlin (graph)</b>, because in this quickstart you are creating a table that works with the Gremlin API.</p> <p><a href="#">Learn more about the Gremlin API.</a></p>
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.

Select **Review+Create**. You can skip the **Network** and **Tags** section.

**Create Azure Cosmos DB Account**

[X](#)

[Basics](#) [Networking](#) [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 \* [A Subscription](#)

Resource Group \* [Select existing...](#) [Create new](#)

**Instance Details**

Account Name \*

API \* [Gremlin \(graph\)](#)

Apache Spark [Notebooks \(preview\)](#) [Notebooks with Apache Spark \(preview\)](#) [None](#) [Sign up for Apache Spark preview](#)

Location \* [\(US\) West US](#)

Geo-Redundancy [Enable](#) [Disable](#)

Multi-region Writes [Enable](#) [Disable](#)

\*Up to 33% off multi-region writes is available to qualifying new accounts only. Accounts must be created between December 1, 2019 and February 29, 2020. 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](#)

- The account creation takes a few minutes. Wait for the portal to display the **Congratulations! Your Azure Cosmos DB account was created** page.

The screenshot shows the 'cosmos-db-quickstart - Quick start' page for an Azure Cosmos DB account. On the left, a sidebar lists various management options like Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Quick start (which is selected), Notifications, Data Explorer, Settings, and Connection String. The main content area starts with a message: 'Congratulations! Your Azure Cosmos DB account was created.' It then instructs the user to 'Choose a platform' and provides links for '.NET', 'Gremlin Console', and 'Guided Gremlin Tour'. Below this, two numbered steps are shown: Step 1 'Add a graph' and Step 2 'Download and run your .NET app'. Step 1 includes a note about creating a 'Persons' container and a 'Create 'Persons' container' button.

## 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.

The screenshot shows the 'cdbgraph1 - Data Explorer' page in the Azure portal. The left sidebar has 'Data Explorer' selected and highlighted with a red box. The main area shows the 'GREMLIN API' and has a 'New Database' and 'New Graph' button. A modal window titled 'Add Graph' is open on the right. It contains fields for 'Database id' (set to 'sample-database'), 'Provision database throughput' (set to '1000'), 'Graph id' (set to 'sample-graph'), and 'Partition key' (set to '/pk'). There are also checkboxes for 'Create new' and 'Use existing' and a note about estimated spend. At the bottom of the modal is an 'OK' button.

### 2. In the **Add graph** page, enter the settings for the new graph.

SETTING	SUGGESTED VALUE	DESCRIPTION
Database ID	sample-database	Enter <i>sample-database</i> 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.

SETTING	SUGGESTED VALUE	DESCRIPTION
Graph ID	sample-graph	Enter <i>sample-graph</i> 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 <a href="#">Graph Data Partitioning article</a> .

- 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.

- Open a command prompt, create a new folder named git-samples, then close the command prompt.

```
md "C:\git-samples"
```

- 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"
```

- 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
```

- Then open Visual Studio and open the solution file.
- Restore the NuGet packages in the project. This should include the Gremlin.Net driver, as well as the Newtonsoft.Json package.
- 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 string EndpointUrl = Environment.GetEnvironmentVariable("EndpointUrl");
private string PrimaryKey = Environment.GetEnvironmentVariable("PrimaryKey");
private static int port = 443;
private static string database = "your-database-name";
private static string container = "your-container-or-graph-name";

```

- 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', desc)" },
    { "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:

```

var gremlinServer = new GremlinServer(EndpointUrl, port, enableSsl: true,
                                         username: "/dbs/" + database + "/colls/" + container,
                                         password: PrimaryKey);

using (var gremlinClient = new GremlinClient(gremlinServer, new GraphSON2Reader(), new
                                         GraphSON2Writer(), GremlinClient.GraphSON2MimeType))
{

```

- 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.
    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.

- 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.

ID	DATABASE	THROUGHPUT (RU/S)
graphcoll	graphdb	400

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`

- Next, navigate to the **Keys** tab and copy the **PRIMARY KEY** value from the Azure portal.
- 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 EndpointUrl "https://<your cosmos db 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.

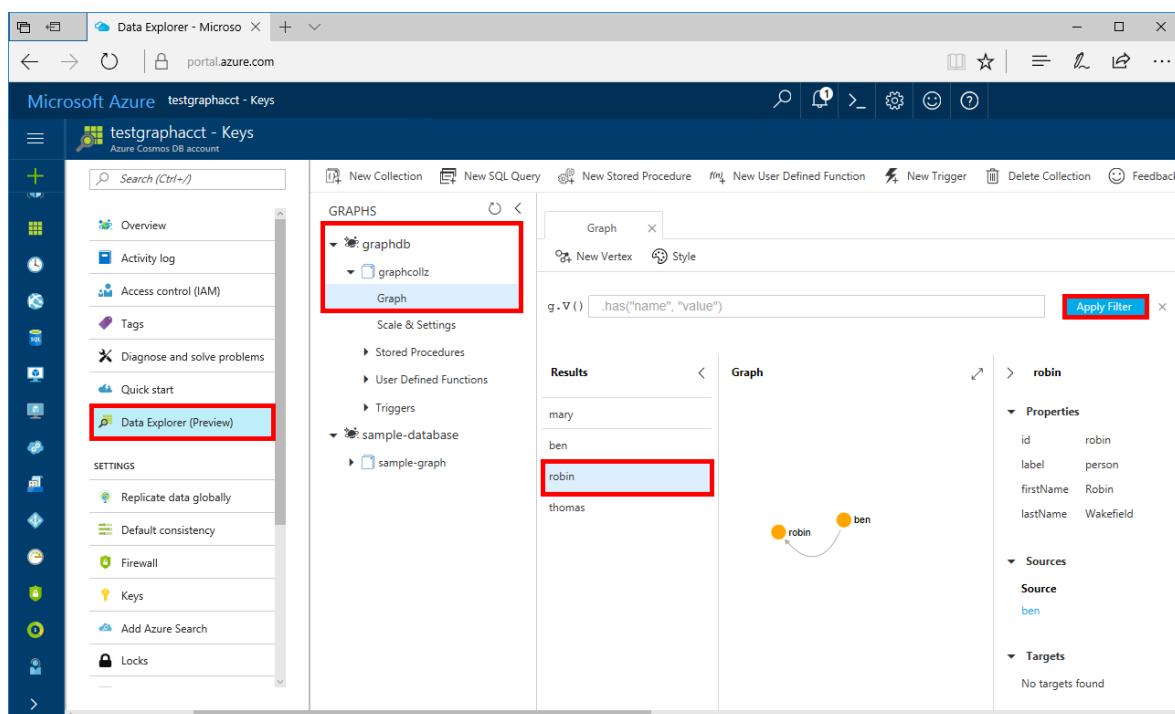
The console window displays the vertexes and edges being added to the graph. When the script completes, press ENTER to close the console window.

## 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.

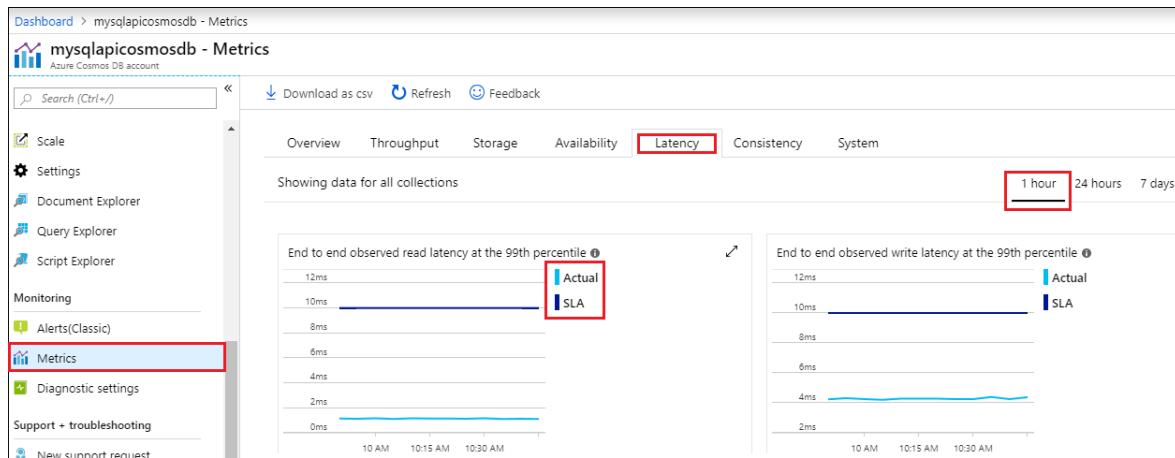


# Review SLAs in the Azure portal

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.

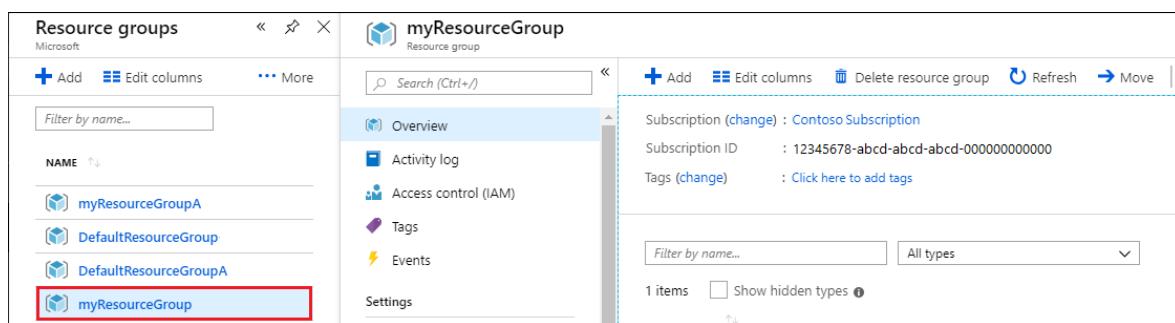


3. Review the metrics on the other tabs.

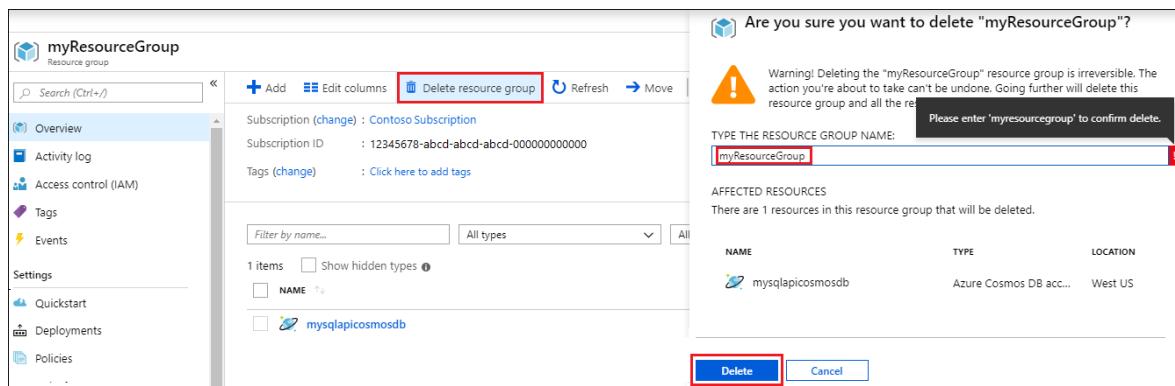
## Clean up resources

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

2/6/2020 • 9 minutes to read • [Edit Online](#)

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.

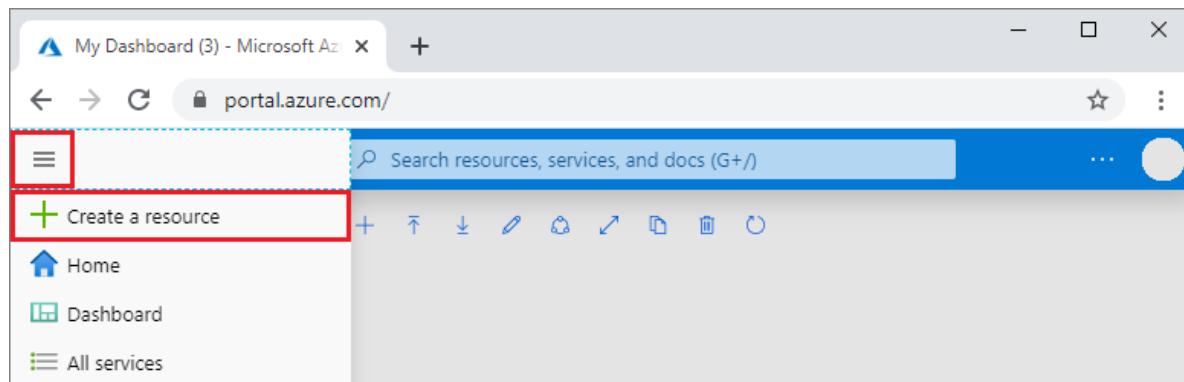
## Prerequisites

- 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**.

New

Search the Marketplace

Azure Marketplace [See all](#)

Get started  
Recently created  
AI + Machine Learning  
Analytics  
Blockchain  
Compute  
Containers  
**Databases**  
Developer Tools  
DevOps  
Identity  
Integration  
Internet of Things  
Media  
Mixed Reality

Featured [See all](#)

 Azure SQL Managed Instance  
[Quickstart tutorial](#)

 SQL Database  
[Quickstart tutorial](#)

 Azure Synapse Analytics (formerly SQL DW)  
[Quickstart tutorial](#)

 Azure Database for MariaDB  
[Learn more](#)

 Azure Database for MySQL  
[Quickstart tutorial](#)

 Azure Database for PostgreSQL  
[Quickstart tutorial](#)

 Azure Cosmos DB  
[Quickstart tutorial](#)

- 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 <b>Create new</b> . 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	<p>Enter a unique name to identify your Azure Cosmos DB account. Your account URI will be <i>gremlin.azure.com</i> appended to your unique account name.</p> <p>The account name can use only lowercase letters, numbers, and hyphens (-), and must be between 3 and 31 characters long.</p>

SETTING	VALUE	DESCRIPTION
API	Gremlin (graph)	<p>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.</p> <p>Select <b>Gremlin (graph)</b>, because in this quickstart you are creating a table that works with the Gremlin API.</p> <p><a href="#">Learn more about the Gremlin API.</a></p>
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.

Select **Review + Create**. You can skip the **Network** and **Tags** section.

**Create Azure Cosmos DB Account**

[Basics](#) [Networking](#) [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 *	A Subscription
Resource Group *	<input type="button" value="Select existing..."/> <input type="button" value="Create new"/>

**Instance Details**

Account Name *	Enter account name
API * ⓘ	Gremlin (graph)
Apache Spark ⓘ	<input type="radio" value="Notebooks (preview)"/> Notebooks (preview) <input type="radio" value="Notebooks with Apache Spark (preview)"/> Notebooks with Apache Spark (preview) <input type="radio" value="None"/> None <a href="#">Sign up for Apache Spark preview</a>
Location *	(US) West US
Geo-Redundancy ⓘ	<input type="button" value="Enable"/> <input type="button" value="Disable"/>
Multi-region Writes ⓘ	<input type="button" value="Enable"/> <input type="button" value="Disable"/>

\*Up to 33% off multi-region writes is available to qualifying new accounts only. Accounts must be created between December 1, 2019 and February 29, 2020. 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 account was created** page.

The screenshot shows the 'cosmos-db-quickstart - Quick start' page in the Azure portal. On the left, there's a sidebar with links like Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Quick start (which is selected), Notifications, Data Explorer, Settings, and Connection String. The main content area says 'Congratulations! Your Azure Cosmos DB account was created.' and 'Now, let's connect to it using a sample app:'. It has a 'Choose a platform' section with '.NET' selected, and options for 'Gremlin Console' and 'Guided Gremlin Tour'. Below that, there are two numbered steps: '1 Add a graph' and '2 Download and run your .NET app'.

## 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.

The screenshot shows the 'cdbgraph1 - Data Explorer' page in the Azure portal. The left sidebar has 'Data Explorer' selected. In the center, there's a 'New Database' button and a 'New Graph' button, with 'New Graph' highlighted by a red box. A modal window titled 'Add Graph' is open on the right. It contains fields for 'Database id' (radio buttons for 'Create new' and 'Use existing', with 'sample-database' selected), 'Provision database throughput' (checkbox checked), 'Throughput' (text input set to 1000), 'Graph id' (text input set to 'sample-graph'), 'Partition key' (text input set to '/pk'), and a note about partition key size. At the bottom is an 'OK' button.

### 2. In the **Add graph** page, enter the settings for the new graph.

SETTING	SUGGESTED VALUE	DESCRIPTION
Database ID	sample-database	Enter <i>sample-database</i> 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.

SETTING	SUGGESTED VALUE	DESCRIPTION
Graph ID	sample-graph	Enter <i>sample-graph</i> 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 <a href="#">Graph Data Partitioning article</a> .

- 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.

- Open a command prompt, create a new folder named git-samples, then close the command prompt.

```
md "C:\git-samples"
```

- 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"
```

- 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.

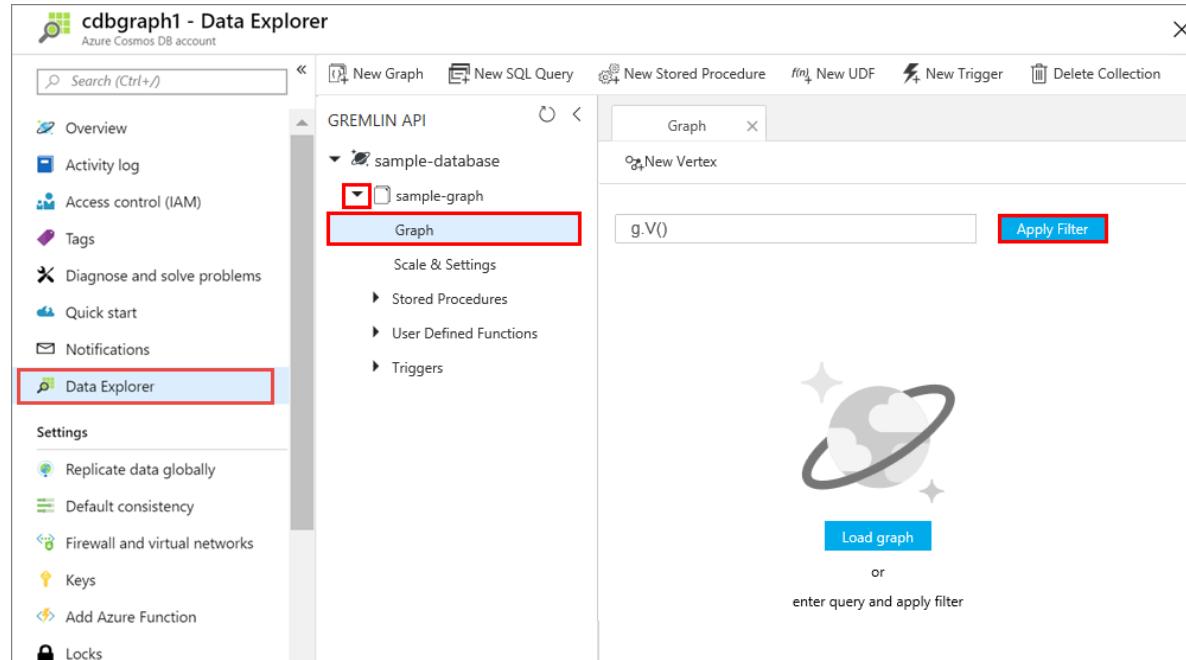
If you experience timeout errors, check that you updated the connection information correctly in [Update your connection information](#), and also try running the last command again.

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.

The screenshot shows the Neo4j Web Console interface. On the left, under the 'Results' tab, there is a list of node names: 'mary', 'ben' (which is highlighted with a red box), and 'robin'. In the center, a graph visualization shows two orange circular nodes labeled 'ben' and 'robin' connected by a single edge. On the right, the 'Graph' tab is active, displaying detailed information about the node 'ben': its properties (id: ben, label: person, firstName: Ben, lastName: Miller), sources (none), and targets (an edge to 'robin' labeled 'knows').

3. Let's add a few new users. Select **New Vertex** to add data to your graph.

The screenshot shows the Neo4j Data Explorer interface. On the left, the 'Data Explorer' sidebar is visible with options like 'New Graph', 'New SQL Query', etc. The main area shows a 'Graph API' section with a 'sample-graph' database selected. A 'New Vertex' dialog box is open, containing fields for 'label' (set to 'person'), 'id' (set to 'ashley'), 'gender' (set to 'female'), and 'tech' (set to 'java'). A red box highlights the 'label' field and the property input fields. At the bottom of the dialog, the 'OK' button is also highlighted with a red box.

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.

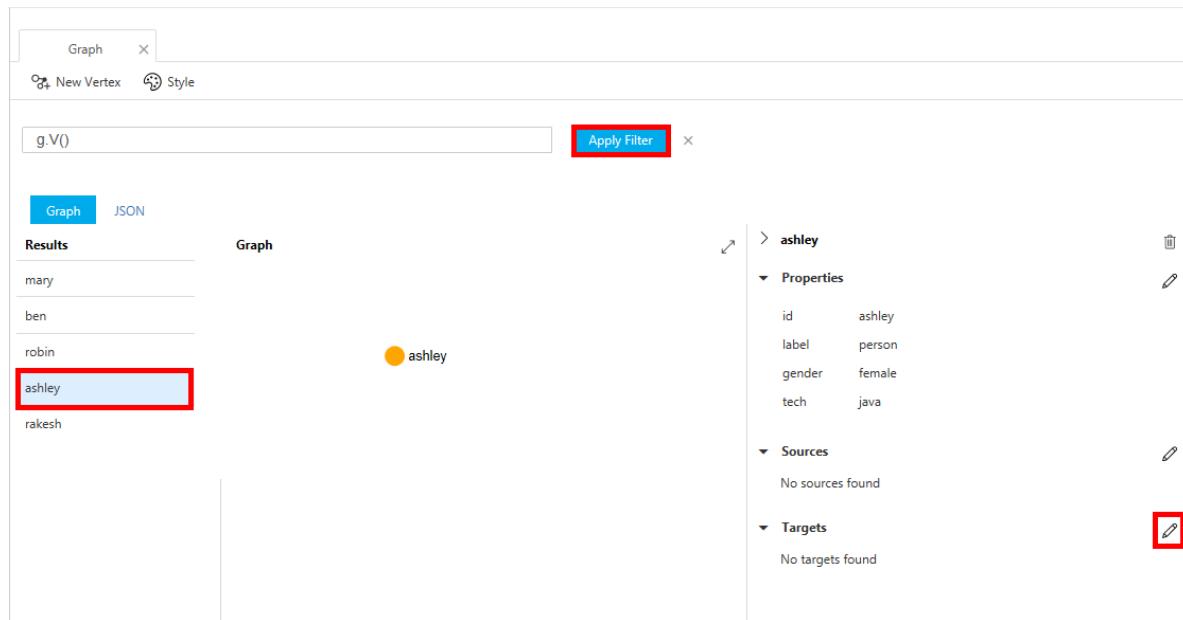
6. Select **OK**. You may need to expand your screen to see **OK** on the bottom of the screen.

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	
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.

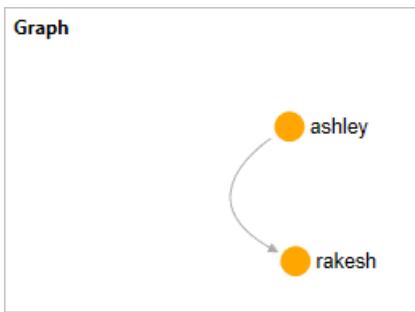


13. In the **Target** box enter *rakesh*, and in the **Edge label** box enter *knows*, and then select the check box.



Target	Edge label
rakesh	knows

14. Now select **rakesh** from the results list and see that ashley and rakesh are connected.



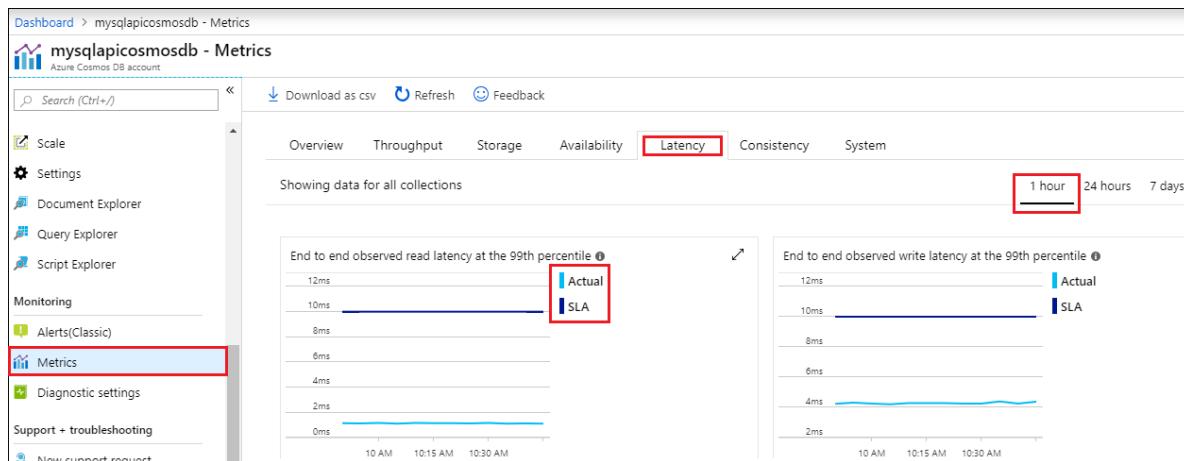
That completes the resource creation part of this tutorial. You can continue to add vertexes to your graph, modify the existing vertexes, or change the queries. Now let's review the metrics Azure Cosmos DB provides, and then clean up the resources.

## 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.

The screenshot shows the Azure portal's Resource groups blade. On the left, a list of resource groups is shown, with 'myResourceGroup' selected and highlighted by a red box. The right pane displays the 'Overview' page for 'myResourceGroup', which includes the subscription details ('Contoso Subscription'), subscription ID ('12345678-abcd-abcd-000000000000'), and tags ('Click here to add tags'). Below this, there is a list of resources with one item ('1 items') and a checkbox for 'Show hidden types'. The 'myResourceGroup' entry is also highlighted with a red box.

3. On the resource group **Overview** page, select **Delete resource group**.

The screenshot shows a confirmation dialog titled 'Are you sure you want to delete "myResourceGroup"?'. It includes a warning message about the irreversibility of the action and a text input field where 'myResourceGroup' is typed. Below the input field, it says 'AFFECTED RESOURCES' and lists one resource: 'mysqlapicosmosdb' (Azure Cosmos DB account, West US). At the bottom, there are 'Delete' and 'Cancel' buttons, with 'Delete' highlighted by a red box.

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

2/6/2020 • 6 minutes to read • [Edit Online](#)

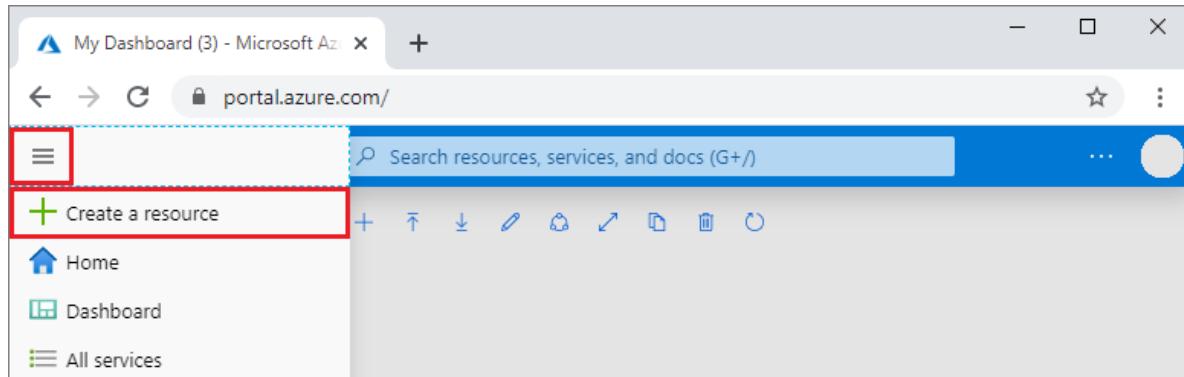
In this quickstart, you create and manage an Azure Cosmos DB Gremlin (graph) API account from the Azure portal, and add data by using a Node.js app cloned from GitHub. Azure Cosmos DB is a multi-model database service that lets you quickly create and query document, table, key-value, and graph databases 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**.

New

Search the Marketplace

Azure Marketplace [See all](#)

Get started  
Recently created  
AI + Machine Learning  
Analytics  
Blockchain  
Compute  
Containers  
**Databases**  
Developer Tools  
DevOps  
Identity  
Integration  
Internet of Things  
Media  
Mixed Reality

Featured [See all](#)

 Azure SQL Managed Instance  
[Quickstart tutorial](#)

 SQL Database  
[Quickstart tutorial](#)

 Azure Synapse Analytics (formerly SQL DW)  
[Quickstart tutorial](#)

 Azure Database for MariaDB  
[Learn more](#)

 Azure Database for MySQL  
[Quickstart tutorial](#)

 Azure Database for PostgreSQL  
[Quickstart tutorial](#)

 Azure Cosmos DB  
[Quickstart tutorial](#)

- 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 <b>Create new</b> . 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	<p>Enter a unique name to identify your Azure Cosmos DB account. Your account URI will be <i>gremlin.azure.com</i> appended to your unique account name.</p> <p>The account name can use only lowercase letters, numbers, and hyphens (-), and must be between 3 and 31 characters long.</p>

SETTING	VALUE	DESCRIPTION
API	Gremlin (graph)	<p>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.</p> <p>Select <b>Gremlin (graph)</b>, because in this quickstart you are creating a table that works with the Gremlin API.</p> <p><a href="#">Learn more about the Gremlin API.</a></p>
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.

Select **Review + Create**. You can skip the **Network** and **Tags** section.

**Create Azure Cosmos DB Account**

[Basics](#) [Networking](#) [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 *	A Subscription
Resource Group *	<input type="button" value="Select existing..."/> <input type="button" value="Create new"/>

**Instance Details**

Account Name *	Enter account name
API * ⓘ	Gremlin (graph)
Apache Spark ⓘ	<input type="radio" value="Notebooks (preview)"/> Notebooks (preview) <input type="radio" value="Notebooks with Apache Spark (preview)"/> Notebooks with Apache Spark (preview) <input type="radio" value="None"/> None <a href="#">Sign up for Apache Spark preview</a>
Location *	(US) West US
Geo-Redundancy ⓘ	<input type="button" value="Enable"/> <input type="button" value="Disable"/>
Multi-region Writes ⓘ	<input type="button" value="Enable"/> <input type="button" value="Disable"/>

\*Up to 33% off multi-region writes is available to qualifying new accounts only. Accounts must be created between December 1, 2019 and February 29, 2020. 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 account was created** page.

The screenshot shows the 'cosmos-db-quickstart - Quick start' page for an Azure Cosmos DB account. On the left, a sidebar lists various management options like Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Quick start (which is selected), Notifications, Data Explorer, Settings, and Connection String. The main content area starts with a message: 'Congratulations! Your Azure Cosmos DB account was created.' It then prompts the user to 'Choose a platform' with options for .NET, Gremlin Console, and Guided Gremlin Tour. The first step, '1 Add a graph', is highlighted. It explains that you can create containers to store graph elements (vertices and edges) and provides a link to 'Create 'Persons' container'. Below this, the second step, '2 Download and run your .NET app', is shown with a link to 'Download'. The overall interface is clean and modern, typical of the Azure portal.

## 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.

The screenshot shows the 'cdbgraph1 - Data Explorer' page in the Azure portal. The left sidebar has 'Data Explorer' selected. In the center, there's a 'GREMLIN API' section with a 'New Database' button and a 'New Graph' button, which is highlighted with a red box. To the right, a modal window titled 'Add Graph' is open. It contains fields for 'Database id' (set to 'sample-database'), 'Throughput' (set to '1000'), 'Graph id' (set to 'sample-graph'), and 'Partition key' (set to '/pk'). There are also checkboxes for 'Provision database throughput' and 'My partition key is larger than 100 bytes'. At the bottom of the modal is an 'OK' button. The URL at the bottom of the page is /ms.portal.azure.com/#.

### 2. In the **Add graph** page, enter the settings for the new graph.

SETTING	SUGGESTED VALUE	DESCRIPTION
Database ID	sample-database	Enter <i>sample-database</i> 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.

SETTING	SUGGESTED VALUE	DESCRIPTION
Graph ID	sample-graph	Enter <i>sample-graph</i> 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 <a href="#">Graph Data Partitioning article</a> .

- 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.

- Open a command prompt, create a new folder named git-samples, then close the command prompt.

```
md "C:\git-samples"
```

- 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"
```

- 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
```

- 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 Nodejs](#) 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/";
```

The screenshot shows the Azure portal interface for a Cosmos DB account. On the left, there's a sidebar with links like 'Search (Ctrl+)', 'Activity log', 'Access control (IAM)', 'Tags', 'Diagnose and solve problems', and 'Quick start'. The main area is titled 'Overview' and contains the following information:

- Status: Online
- Resource group: [change](#) cosmos-db
- Subscription: [change](#) A Subscription
- Subscription ID: 12345678-1234-1234-1234-123456781234
- Read Locations: West US
- Write Locations: West US
- .NET SDK URI: <https://cosmos-db-graphacct.documents.azure.com:443/>
- Gremlin Endpoint: <wss://cosmos-db-graphacct.gremlin.cosmos.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";
```

The screenshot shows the Azure portal interface for the 'Keys' page of the same Cosmos DB account. The sidebar includes 'Access control (IAM)', 'Tags', 'Diagnose and solve problems', 'Quick start', 'Notifications', 'Data Explorer', 'Settings' (with 'Replicate data globally', 'Default consistency', 'Firewall and virtual networks', 'Private Endpoint Connections'), and 'Keys'. The main area displays the following keys:

Type	Value
.NET SDK URI	<a href="https://cosmos-db-graphacct.documents.azure.com:443/">https://cosmos-db-graphacct.documents.azure.com:443/</a>
GREMLIN ENDPOINT	<a href="wss://cosmos-db-graphacct.gremlin.cosmos.azure.com:443/">wss://cosmos-db-graphacct.gremlin.cosmos.azure.com:443/</a>
PRIMARY KEY	cukO8qRTLh4zik76ov9PRVS1UELZ2iQh2eDOUnWsRjcw3xrQzVOs93jKLojOp9Wh3IjpRmcR6gSYXyHLh4zik76o...
SECONDARY KEY	RA8Q6Bc2HcuJsqNf4vSrPkuZiq7SrDeGUbn1XbL6leBYF43cvzgvaFl4VCjvtEPK4QzWeIwpZMQfnrLh4zik76o...
PRIMARY CONNECTION STRING	AccountEndpoint=https://cosmos-db-graphacct.documents.azure.com:443/;AccountKey=cukO8qRTLh4zik76o...
SECONDARY CONNECTION STRING	AccountEndpoint=https://cosmos-db-graphacct.documents.azure.com:443/;AccountKey=RA8Q6Bc2HcuJsqNf...

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 not have HTTPS or 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**.

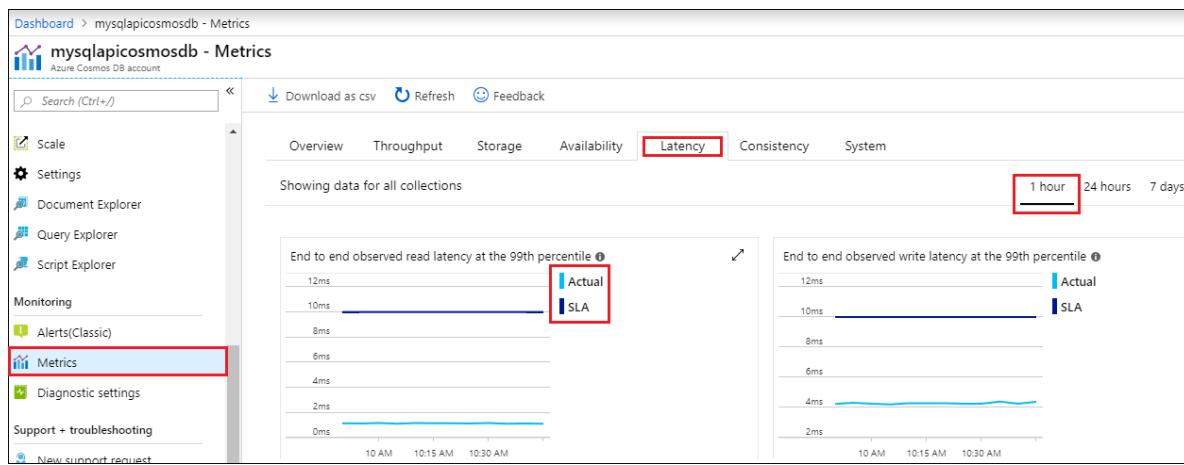
Try completing `g.V()` with `.has('firstName', 'Thomas')` to test the filter. Note that the value is case sensitive.

## 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.



- 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:

- In the Azure portal Search bar, search for and select **Resource groups**.
- From the list, select the resource group you created for this quickstart.

- On the resource group **Overview** page, select **Delete resource group**.

- 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

2/6/2020 • 9 minutes to read • [Edit Online](#)

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](#).

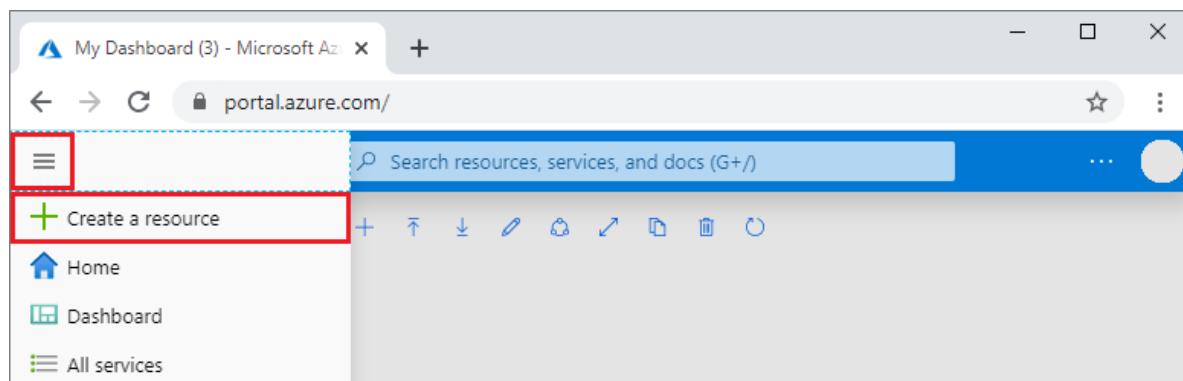
### NOTE

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**.

New

Search the Marketplace

Azure Marketplace See all

Featured See all

Get started	 Azure SQL Managed Instance <a href="#">Quickstart tutorial</a>
Recently created	 SQL Database <a href="#">Quickstart tutorial</a>
AI + Machine Learning	 Azure Synapse Analytics (formerly SQL DW) <a href="#">Quickstart tutorial</a>
Analytics	 Blockchain
Blockchain	 Compute
Compute	 Containers
Databases	 Azure Database for MariaDB <a href="#">Learn more</a>
Developer Tools	 DevOps
DevOps	 Identity
Identity	 Integration
Integration	 Internet of Things
Internet of Things	 Media
Media	 Mixed Reality
Mixed Reality	 Azure Cosmos DB <a href="#">Quickstart tutorial</a>

- 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 <b>Create new</b> . 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 <i>gremlin.azure.com</i> 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	Gremlin (graph)	<p>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.</p> <p>Select <b>Gremlin (graph)</b>, because in this quickstart you are creating a table that works with the Gremlin API.</p> <p><a href="#">Learn more about the Gremlin API.</a></p>
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.

Select **Review+Create**. You can skip the **Network** and **Tags** section.

**Create Azure Cosmos DB Account**

[Basics](#) [Networking](#) [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 \*

Resource Group \*  [Create new](#)

**Instance Details**

Account Name \*

API \*

Apache Spark [\(preview\)](#)  [Sign up for Apache Spark preview](#)

Location \*

Geo-Redundancy [\(preview\)](#) [Enable](#) [Disable](#)

Multi-region Writes [\(preview\)](#) [Enable](#) [Disable](#)

\*Up to 33% off multi-region writes is available to qualifying new accounts only. Accounts must be created between December 1, 2019 and February 29, 2020. 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 account was created** page.

The screenshot shows the 'cosmos-db-quickstart - Quick start' page for an Azure Cosmos DB account. On the left, a sidebar lists options like Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Quick start (which is selected), Notifications, Data Explorer, Settings, and Connection String. The main content area starts with a message: 'Congratulations! Your Azure Cosmos DB account was created.' It then says 'Now, let's connect to it using a sample app:' followed by 'Choose a platform' with options for .NET, Gremlin Console, and Guided Gremlin Tour. Below this, a large numbered step '1 Add a graph' is shown, with a sub-step 'Create 'Persons' container'. It explains that you can create containers to store graph elements (vertices and edges) and provides a link to 'Create 'Persons' container'. Step '2 Download and run your .NET app' follows, with a note that once the graph is created, you can download a sample .NET app connected to it, extract, build, and run. A 'Download' button is provided.

## 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.

The screenshot shows the 'cdbgraph1 - Data Explorer' page in the Azure portal. The left sidebar is identical to the one in the previous screenshot, with 'Data Explorer' selected. In the center, there is a 'GREMLIN API' section. On the right, a modal dialog box titled 'Add Graph' is open. This dialog contains fields for creating a new graph: 'Database id' (radio buttons for 'Create new' or 'Use existing', with 'sample-database' selected), 'Provision database throughput' (checkbox checked), 'Throughput (400 - 100,000 RU/s)' (input field set to 1000), 'Graph id' (input field set to 'sample-graph'), 'Partition key' (input field set to '/pk'), and a checkbox 'My partition key is larger than 100 bytes'. At the bottom of the dialog are 'OK' and 'Cancel' buttons.

### 2. In the **Add graph** page, enter the settings for the new graph.

SETTING	SUGGESTED VALUE	DESCRIPTION
Database ID	sample-database	Enter <i>sample-database</i> 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.

SETTING	SUGGESTED VALUE	DESCRIPTION
Graph ID	sample-graph	Enter <i>sample-graph</i> 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 <a href="#">Graph Data Partitioning article</a> .

- 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.

- Open a command prompt, create a new folder named git-samples, then close the command prompt.

```
md "C:\git-samples"
```

- 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"
```

- 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:

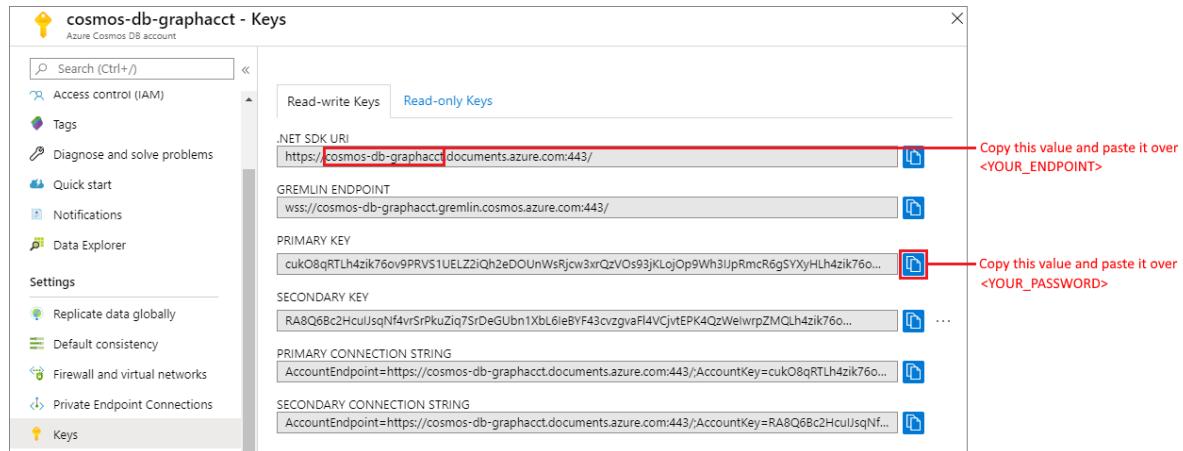
```
client.submitAsync(_gremlin_cleanup_graph)
```

## 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.

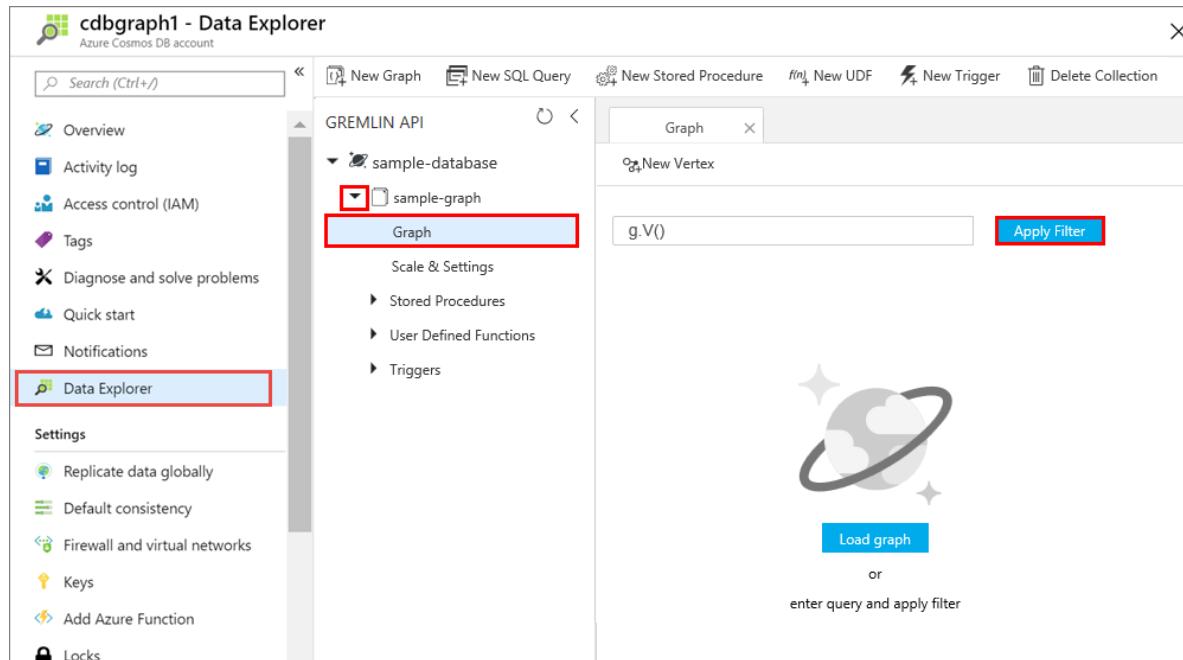
If you experience timeout errors, check that you updated the connection information correctly in [Update your connection information](#), and also try running the last command again.

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.

The screenshot shows the Data Explorer interface with the following details:

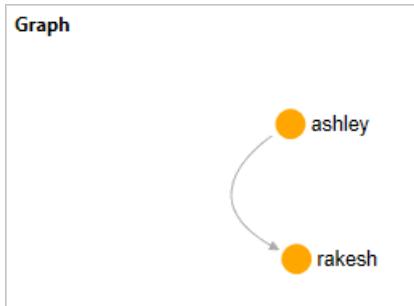
- Graph Tab:** The active tab, showing a search bar with `g.V()` and an **Apply Filter** button (highlighted in red).
- Results List:** A list of vertices: mary, ben, robin, ashley (selected and highlighted with a red box), and rakesh.
- Graph View:** A visualization showing a single orange node labeled "ashley".
- Properties Panel:** Details for the selected vertex "ashley".
  - Properties:** id: ashley, label: person, gender: female, tech: java.
  - Sources:** No sources found.
  - Targets:** No targets found.

13. In the **Target** box type *rakesh*, and in the **Edge label** box type *knows*, and then select the check.

Target	Edge label
rakesh	knows

+ Add Target

14. Now select **rakesh** from the results list and see that ashley and rakesh are connected.



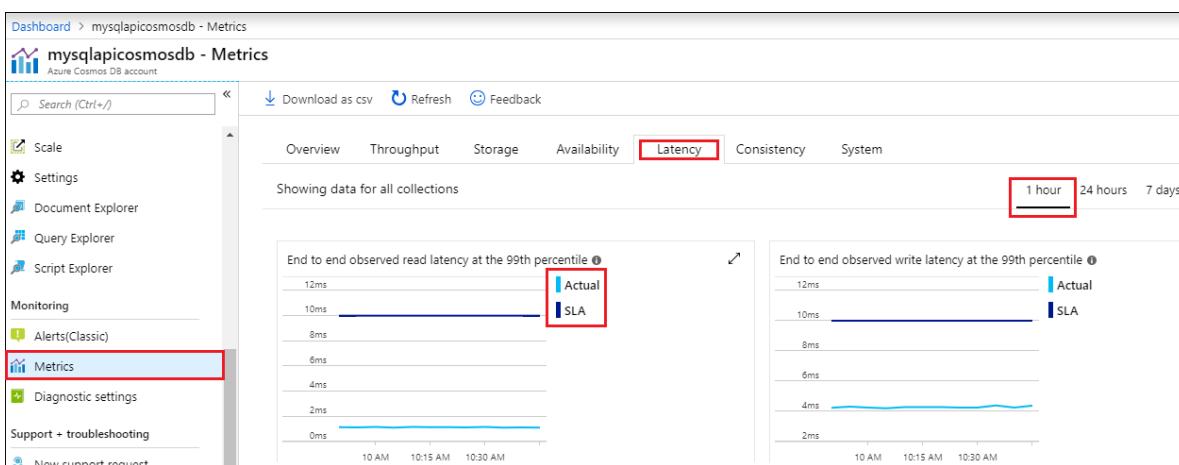
That completes the resource creation part of this tutorial. You can continue to add vertexes to your graph, modify the existing vertexes, or change the queries. Now let's review the metrics Azure Cosmos DB provides, and then clean up the resources.

## 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.

The screenshot shows the Azure portal's 'Resource groups' blade. On the left, a list of resource groups is shown: 'myResourceGroupA', 'DefaultResourceGroup', 'DefaultResourceGroupA', and 'myResourceGroup'. The 'myResourceGroup' item is highlighted with a red box. The main pane displays the details for 'myResourceGroup', including its subscription information ('Contoso Subscription'), subscription ID ('12345678-abcd-abcd-abcd-000000000000'), and tags ('Click here to add tags'). A secondary pane on the right shows the 'Overview' of the resource group, listing 'Subscription (change) : Contoso Subscription', 'Subscription ID : 12345678-abcd-abcd-abcd-000000000000', and 'Tags (change) : Click here to add tags'. It also includes a 'Filter by name...' search bar and a 'Show hidden types' checkbox.

3. On the resource group **Overview** page, select **Delete resource group**.

The screenshot shows the 'myResourceGroup' resource group overview page. The 'Delete resource group' button in the top navigation bar is highlighted with a red box. A confirmation dialog box is open on the right, asking 'Are you sure you want to delete "myResourceGroup"?'. It contains a warning message: 'Warning! Deleting the "myResourceGroup" resource group is irreversible. The action you're about to take can't be undone. Going further will delete this resource group and all the resources it contains.' Below this, a text input field says 'Please enter 'myresourcegroup' to confirm delete.' with 'myResourceGroup' typed in. The dialog also lists 'Affected Resources' with one item: 'mysqlapicosmosdb' (Azure Cosmos DB account, West US). At the bottom are 'Delete' and 'Cancel' buttons, with 'Delete' highlighted with a red box.

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

8/2/2019 • 9 minutes to read • [Edit Online](#)

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:

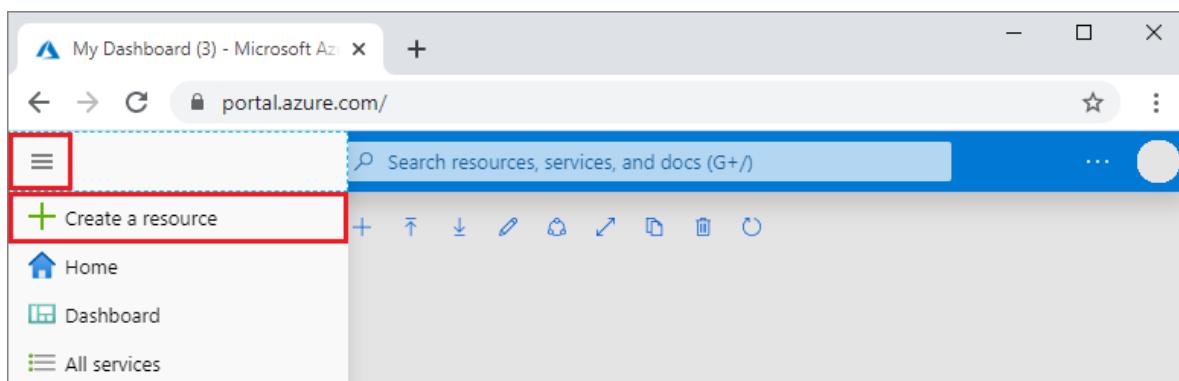
- [PHP 5.6 or newer](#)
- [Composer](#)

## 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**.

New

Search the Marketplace

Azure Marketplace See all

Featured See all

Get started	Azure SQL Managed Instance <a href="#">Quickstart tutorial</a>
Recently created	SQL Database <a href="#">Quickstart tutorial</a>
AI + Machine Learning	Azure Synapse Analytics (formerly SQL DW) <a href="#">Quickstart tutorial</a>
Analytics	Azure Database for MariaDB <a href="#">Learn more</a>
Blockchain	Azure Database for MySQL <a href="#">Quickstart tutorial</a>
Compute	Azure Database for PostgreSQL <a href="#">Quickstart tutorial</a>
Containers	Azure Cosmos DB <a href="#">Quickstart tutorial</a>
<b>Databases</b>	<b>Azure Cosmos DB</b> <a href="#">Quickstart tutorial</a>
Developer Tools	
DevOps	
Identity	
Integration	
Internet of Things	
Media	
Mixed Reality	

- 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 <b>Create new</b> . 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	<p>Enter a unique name to identify your Azure Cosmos DB account. Your account URI will be <i>gremlin.azure.com</i> appended to your unique account name.</p> <p>The account name can use only lowercase letters, numbers, and hyphens (-), and must be between 3 and 31 characters long.</p>

SETTING	VALUE	DESCRIPTION
API	Gremlin (graph)	<p>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.</p> <p>Select <b>Gremlin (graph)</b>, because in this quickstart you are creating a table that works with the Gremlin API.</p> <p><a href="#">Learn more about the Gremlin API.</a></p>
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.

Select **Review+Create**. You can skip the **Network** and **Tags** section.

**Create Azure Cosmos DB Account**

[Basics](#) [Networking](#) [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 *	A Subscription
Resource Group *	<input type="button" value="Select existing..."/> <input type="button" value="Create new"/>

**Instance Details**

Account Name \*

API \*

Apache Spark    [Sign up for Apache Spark preview](#)

Location \*

Geo-Redundancy

Multi-region Writes

\*Up to 33% off multi-region writes is available to qualifying new accounts only. Accounts must be created between December 1, 2019 and February 29, 2020. 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 account was created** page.

The screenshot shows the 'cosmos-db-quickstart - Quick start' page for an Azure Cosmos DB account. On the left, a sidebar lists options like Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Quick start (which is selected), Notifications, Data Explorer, Settings, and Connection String. The main content area starts with a message: 'Congratulations! Your Azure Cosmos DB account was created.' It then asks to 'Choose a platform' with options for .NET, Gremlin Console, and Guided Gremlin Tour. Below this, step 1 'Add a graph' is described with the note: 'In Azure Cosmos DB, you can create containers to store graph elements (vertices and edges).'. A 'Create 'Persons' container' button is shown, along with a note about provisioning: 'Create 'Persons' graph container with 10GB storage capacity and 400 Request Units per second (RU/s) throughput capacity, for up to 400 reads/sec. Estimated hourly bill: \$0.033 USD. Estimated hourly bill: \$0.033 USD'. Step 2 'Download and run your .NET app' is also mentioned.

## 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.

The screenshot shows the 'cdbgraph1 - Data Explorer' page in the Azure portal. The left sidebar is identical to the one in the first screenshot, with 'Data Explorer' selected. In the center, there's a 'GREMLIN API' section. On the right, a modal window titled 'Add Graph' is open. It contains fields for 'Database id': 'sample-database' (radio button selected for 'Create new'), 'Provision database throughput': '1000' (checkbox checked), 'Throughput (400 - 100,000 RU/s)': '1000' (input field), 'Graph id': 'sample-graph' (input field), and 'Partition key': '/pk' (input field). There are also notes about costs and a checkbox for 'My partition key is larger than 100 bytes'. At the bottom of the modal is an 'OK' button.

2. In the **Add graph** page, enter the settings for the new graph.

SETTING	SUGGESTED VALUE	DESCRIPTION
Database ID	sample-database	Enter <i>sample-database</i> 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.

SETTING	SUGGESTED VALUE	DESCRIPTION
Graph ID	sample-graph	Enter <i>sample-graph</i> 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 <a href="#">Graph Data Partitioning article</a> .

- 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.

- Open a command prompt, create a new folder named git-samples, then close the command prompt.

```
md "C:\git-samples"
```

- 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"
```

- 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/<coll1>',
    '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**.

The screenshot shows the Microsoft Azure portal with the URL [portal.azure.com](https://portal.azure.com). The page title is "Microsoft Azure Azure Cosmos DB > test-graph - Data Explorer". On the left sidebar, under "mimi-test-graph - Data Explorer", the "Data Explorer" option is selected and highlighted with a red box. In the main content area, the "GRAPHS" section is expanded, showing a tree view with "sample-database" and "sample-graph". The "sample-graph" node is selected and highlighted with a red box. Under "sample-graph", there are four options: "Graph" (selected and highlighted with a red box), "Scale & Settings", "Stored Procedures", and "User Defined Functions". To the right of the tree view is a "Graph" panel with a search bar containing "g.V()", a "Load graph" button, and an "Apply Filter" button. Below the search bar is a placeholder text "enter query and apply filter".

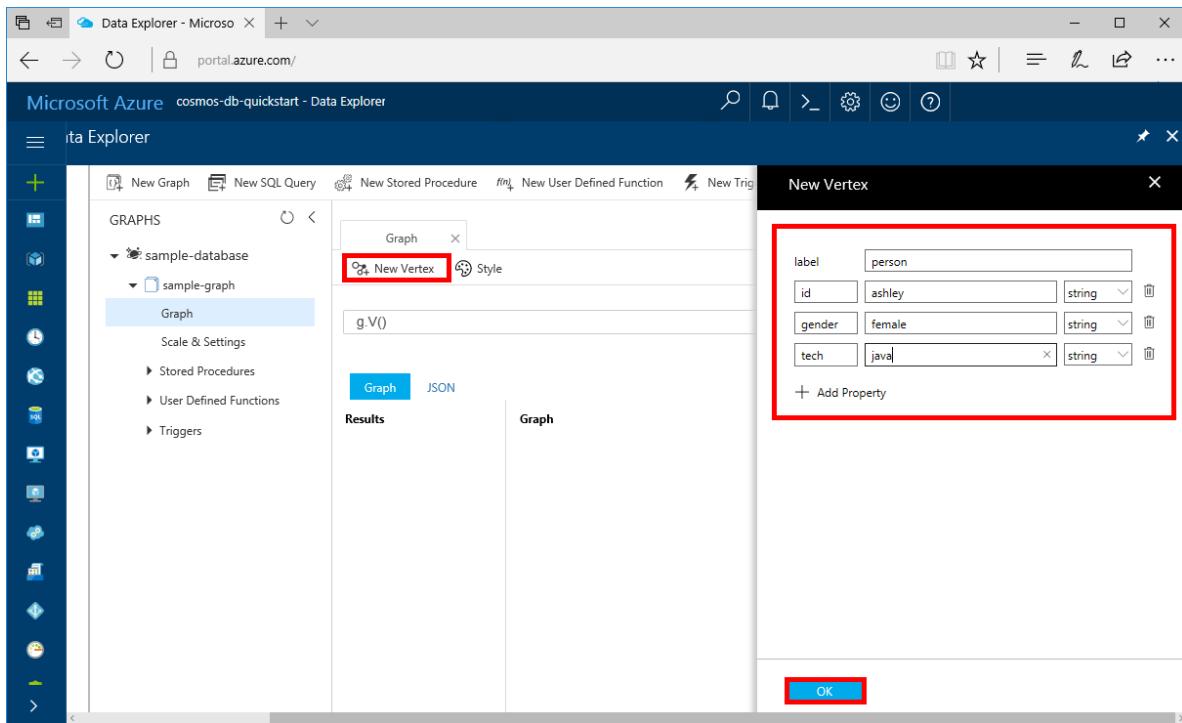
2. In the **Results** list, notice the new users added to the graph. Select **ben** and notice that they're connected to robin. You can move the vertices around by dragging and dropping, zoom in and out by scrolling the wheel of your mouse, and expand the size of the graph with the double-arrow.

The screenshot shows the "Results" pane of the Azure Data Explorer interface. The "Graph" tab is selected. The results list on the left shows three entries: "mary", "ben" (which is selected and highlighted with a red box), and "robin". In the center, there is a graph visualization showing two nodes: "ben" (orange circle) and "robin" (orange circle). They are connected by a single edge. On the right, the details for the selected vertex "ben" are shown in a panel:

- Properties:**

id	ben
label	person
firstName	Ben
lastName	Miller
- Sources:** No sources found.
- Targets:** A table showing one target: "robin" with an edge label "knows".

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
<b>id</b>	ashley	The unique identifier for the vertex. If you don't specify an id, one is generated for you.
<b>gender</b>	female	
<b>tech</b>	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
<b>id</b>	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**.

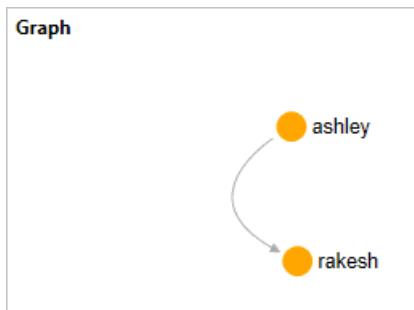
11. Click the **Apply Filter** button with the default `g.V()` filter to display all the values in the graph. All of the users now show in the **Results** list.

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.



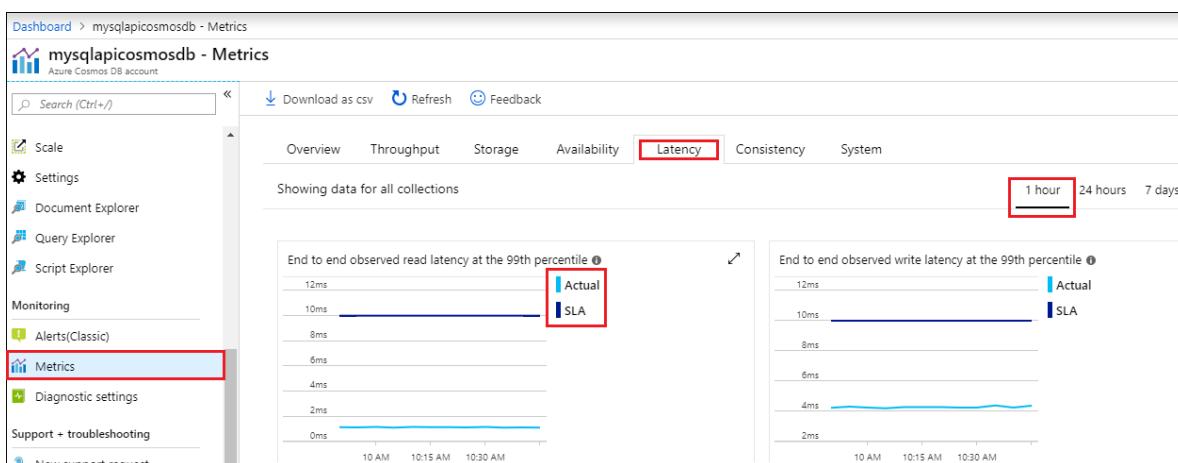
That completes the resource creation part of this quickstart. You can continue to add vertexes to your graph, modify the existing vertexes, or change the queries. Now let's review the metrics Azure Cosmos DB provides, and then clean up the resources.

## 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.

The screenshot shows the Azure portal's Resource groups blade. On the left, there's a list of resource groups: 'myResourceGroupA', 'DefaultResourceGroup', 'DefaultResourceGroupA', and 'myResourceGroup'. The 'myResourceGroup' item is selected and highlighted with a red box. The main area shows the 'Overview' tab of the selected resource group. The subscription information is displayed: 'Subscription (change) : Contoso Subscription', 'Subscription ID : 12345678-abcd-abcd-000000000000', and 'Tags (change) : Click here to add tags'. Below this is a list of items with a single item '1 items' and a checkbox for 'Show hidden types'.

3. On the resource group **Overview** page, select **Delete resource group**.

The screenshot shows a confirmation dialog box titled 'Are you sure you want to delete "myResourceGroup"?'. It includes a warning message: 'Warning! Deleting the "myResourceGroup" resource group is irreversible. The action you're about to take can't be undone. Going further will delete this resource group and all the resources it contains.' Below this is a text input field with the value 'myResourceGroup'. The dialog also displays a table of 'AFFECTED RESOURCES' with one item: 'mysqlapicosmosdb' (Azure Cosmos DB account, West US). At the bottom are two buttons: 'Delete' (highlighted with a red box) and 'Cancel'.

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

12/5/2019 • 2 minutes to read • [Edit Online](#)

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](#)

# Azure Cosmos DB Gremlin server response headers

10/22/2019 • 6 minutes to read • [Edit Online](#)

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
<b>x-ms-request-charge</b>	double	11.3243	Success and Failure	Amount of collection or database throughput consumed in <a href="#">request units (RU/s or RUs)</a> 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.
<b>x-ms-total-request-charge</b>	double	423.987	Success and Failure	Amount of collection or database throughput consumed in <a href="#">request units (RU/s or RUs)</a> 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.

HEADER	TYPE	SAMPLE VALUE	WHEN INCLUDED	EXPLANATION
<b>x-ms-server-time-ms</b>	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.
<b>x-ms-total-server-time-ms</b>	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.
<b>x-ms-status-code</b>	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.

HEADER	TYPE	SAMPLE VALUE	WHEN INCLUDED	EXPLANATION
<b>x-ms-substatus-code</b>	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.
<b>x-ms-retry-after-ms</b>	string (TimeSpan)	"00:00:03.9500000"	Failure Only	This header is a string representation of a .NET <a href="#">TimeSpan</a> type. This value will only be included in requests failed due provisioned throughput exhaustion. Application should resubmit traversal again after instructed period of time.
<b>x-ms-activity-id</b>	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
<b>401</b>	<p>Error message  <code>"Unauthorized: Invalid credentials provided"</code> 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.</p>
<b>404</b>	<p>Concurrent operations that attempt to delete and update the same edge or vertex simultaneously. Error message  <code>"Owner resource does not exist"</code> indicates that specified database or collection is incorrect in connection parameters in <code>/dbs/&lt;database name&gt;/colls/&lt;collection or graph name&gt;</code> format.</p>
<b>408</b>	<p><code>"Server timeout"</code> indicates that traversal took more than <b>30 seconds</b> 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.</p>
<b>409</b>	<p><code>"Conflicting request to resource has been attempted. Retry to avoid conflicts."</code>  This usually happens when vertex or an edge with an identifier already exists in the graph.</p>
<b>412</b>	<p>Status code is complemented with error message  <code>"PreconditionFailedException": One of the specified pre-condition is not met</code>. 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 <code>g.V('identifier').property('name', 'value')</code>. 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.</p>
<b>429</b>	<p>Request was throttled and should be retried after value in <b>x-ms-retry-after-ms</b></p>
<b>500</b>	<p>Error message that contains  <code>"NotFoundException: Entity with the specified id does not exist in the system."</code> 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.</p>
<b>1000</b>	<p>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.</p>

STATUS	EXPLANATION
<b>1001</b>	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.
<b>1003</b>	"Query exceeded memory limit. Bytes Consumed: XXX, Max: YYY" is returned when traversal exceeds allowed memory limit. Memory limit is <b>2 GB</b> per traversal.
<b>1004</b>	This status code indicates malformed graph request. Request can be malformed when it fails deserialization, non-value type is being serialized as value type or unsupported gremlin operation requested. Application should not retry the request because it will not be successful.
<b>1007</b>	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.
<b>1008</b>	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

10/7/2019 • 2 minutes to read • [Edit Online](#)

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
<i>Script length</i>	<b>64 KB</b>	Maximum length of a Gremlin traversal script per request.
<i>Operator depth</i>	<b>400</b>	Total number of unique steps in a traversal. For example, <code>g.V().out()</code> has an operator count of 2: <code>V()</code> and <code>out()</code> , <code>g.V('label').repeat(out()).times(100)</code> has operator depth of 3: <code>V()</code> , <code>repeat()</code> , and <code>out()</code> because <code>.times(100)</code> is a parameter to <code>.repeat()</code> operator.
<i>Degree of parallelism</i>	<b>32</b>	Maximum number of storage partitions queried in a single request to storage layer. Graphs with hundreds of partitions will be impacted by this limit.
<i>Repeat limit</i>	<b>32</b>	Maximum number of iterations a <code>.repeat()</code> operator can execute. Each iteration of <code>.repeat()</code> step in most cases runs breadth-first traversal, which means that any traversal is limited to at most 32 hops between vertices.
<i>Traversal timeout</i>	<b>30 seconds</b>	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 <a href="#">Apache Spark</a> with <a href="#">Graph Data Frames</a> and <a href="#">Cosmos DB Spark Connector</a> .

RESOURCE	DEFAULT LIMIT	EXPLANATION
<i>Idle connection timeout</i>	<b>1 hour</b>	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.
<i>Resource token per hour</i>	<b>100</b>	<p>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,</p> <div style="border: 1px solid black; padding: 5px; margin-left: 20px;">           "Exceeded allowed resource token limit of 100 that can be used concurrently"         </div> <p>will be returned on the next authentication request.</p>

## Next steps

- [Azure Cosmos DB Gremlin response headers](#)
- [Azure Cosmos DB Resource Tokens with Gremlin](#)

# Azure Cosmos DB Gremlin compatibility

1/3/2020 • 2 minutes to read • [Edit Online](#)

Azure Cosmos DB Graph engine closely follows [Apache TinkerPop](#) traversal steps specification but there are differences.

## 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](#).
- `atcch()` allows querying graphs using declarative pattern matching. This capability isn't available.
- ***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.
- ***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.

## Next steps

- Visit [Cosmos DB user voice](#) page to share feedback and help team focus on features that are important to you.

# Graph data modeling for Azure Cosmos DB Gremlin API

12/26/2019 • 5 minutes to read • [Edit Online](#)

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
--------	----------	------	-------

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 <a href="#">graph partitioning</a> .
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

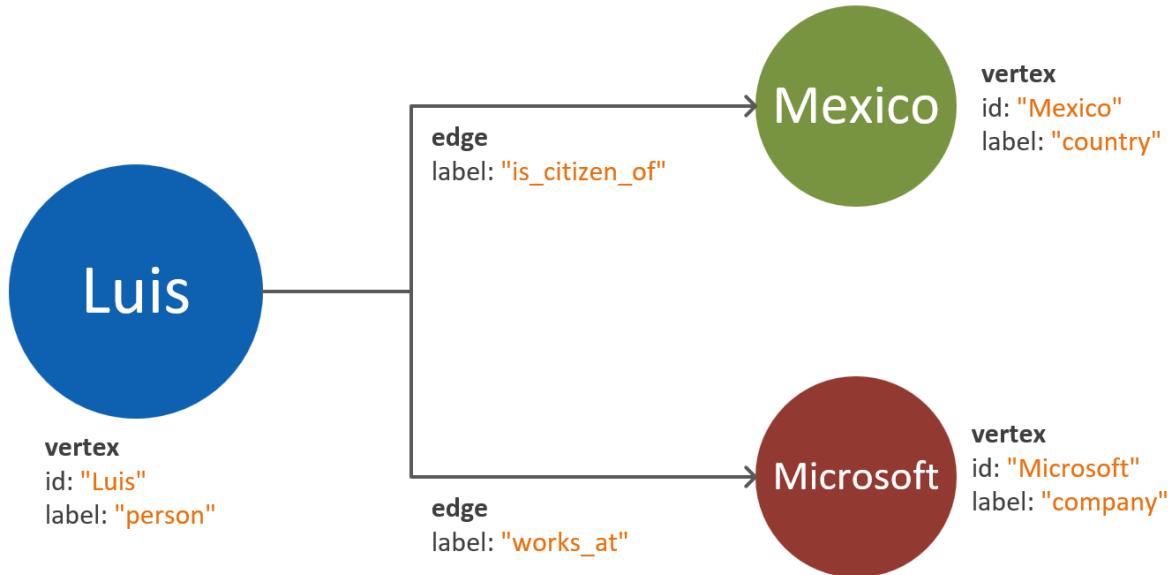
The steps outlined below are presented as recommendations. The final model should be evaluated and tested before its consideration as production-ready. Additionally, the recommendations below are specific to Azure Cosmos DB's Gremlin API implementation.

### 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](#).

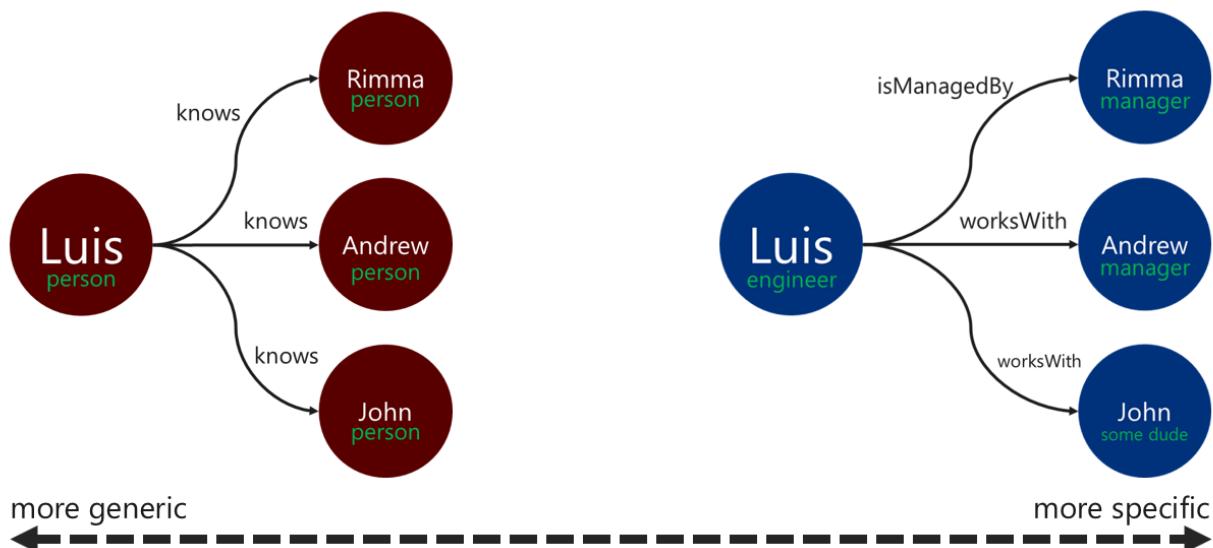
#### NOTE

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.

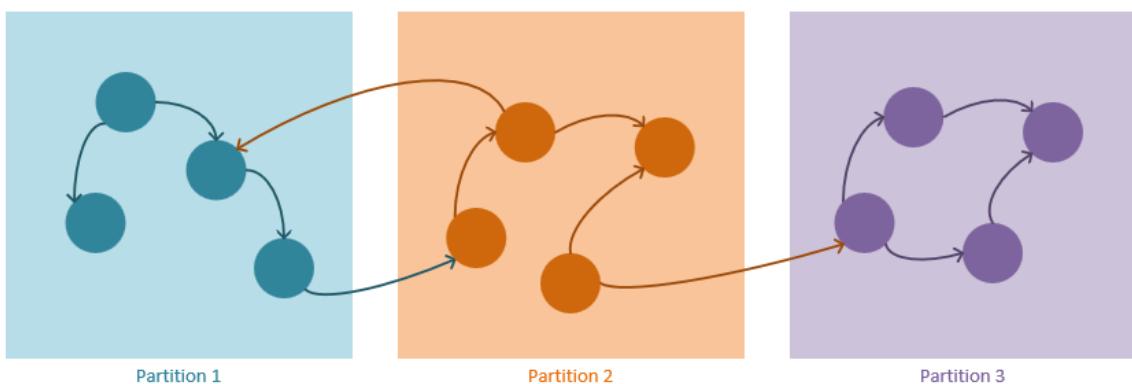
- Evaluate your Gremlin queries using the [Execution Profile step](#).

# Using a partitioned graph in Azure Cosmos DB

2/26/2020 • 3 minutes to read • [Edit Online](#)

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', 'vertex_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('partitio  
nKey_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

12/13/2019 • 6 minutes to read • [Edit Online](#)

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
<code>App.config</code>	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.
<code>Program.cs</code>	This file contains the logic behind creating the <code>DocumentClient</code> collection, handling the cleanups and sending the bulk executor requests.
<code>Util.cs</code>	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
<code>EndPointUrl</code>	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 <code>https://your-graph-database-account.documents.azure.com:443/</code>
<code>AuthorizationKey</code>	This is the Primary or Secondary key listed under your Azure Cosmos DB account. Learn more about <a href="#">Securing Access to Azure Cosmos DB data</a>
<code>DatabaseName</code> , <code>CollectionName</code>	These are the target database and collection names. When <code>ShouldCleanupOnStart</code> is set to <code>true</code> these values, along with <code>CollectionThroughput</code> , will be used to drop them and create a new database and collection. Similarly, if <code>ShouldCleanupOnFinish</code> is set to <code>true</code> , they will be used to delete the database as soon as the ingestion is over. Note that the target collection must be an <b>unlimited collection</b> .
<code>CollectionThroughput</code>	This is used to create a new collection if the <code>ShouldCleanupOnStart</code> option is set to <code>true</code> .
<code>ShouldCleanupOnStart</code>	This will drop the database account and collections before the program is run, and then create new ones with the <code>DatabaseName</code> , <code>CollectionName</code> and <code>CollectionThroughput</code> values.
<code>ShouldCleanupOnFinish</code>	This will drop the database account and collections with the specified <code>DatabaseName</code> and <code>CollectionName</code> after the program is run.
<code>NumberOfDocumentsToImport</code>	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.
<code>NumberOfBatches</code>	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.
<code>CollectionPartitionKey</code>	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 <code>ShouldCleanupOnStart</code> option is set to <code>true</code> .

## 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

12/13/2019 • 7 minutes to read • [Edit Online](#)

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
<code>GetVertices</code>	This step obtains a predicated set of objects from the persistence layer.
<code>GetEdges</code>	This step obtains the edges that are adjacent to a set of vertices. This step can result in one or many storage operations.
<code>GetNeighborVertices</code>	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.
<code>Coalesce</code>	This step accounts for the evaluation of two operations whenever the <code>coalesce()</code> 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 <code>to()</code> or <code>from()</code> 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": {}  
     }  
   ]  
 }
```

```
        },
        "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](#).

# Use Azure Cosmos DB resource tokens with the Gremlin SDK

10/22/2019 • 3 minutes to read • [Edit Online](#)

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/"),
    "<master 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 master 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

- [Role-based access control](#) in Azure Cosmos DB
- [Learn how to secure access to data](#) in Azure Cosmos DB

# Regional endpoints for Azure Cosmos DB Graph account

10/22/2019 • 3 minutes to read • [Edit Online](#)

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-master](#) 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 your 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 "[accountname]-[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 "[accountname]-[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

10/22/2019 • 2 minutes to read • [Edit Online](#)

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

12/26/2019 • 2 minutes to read • [Edit Online](#)

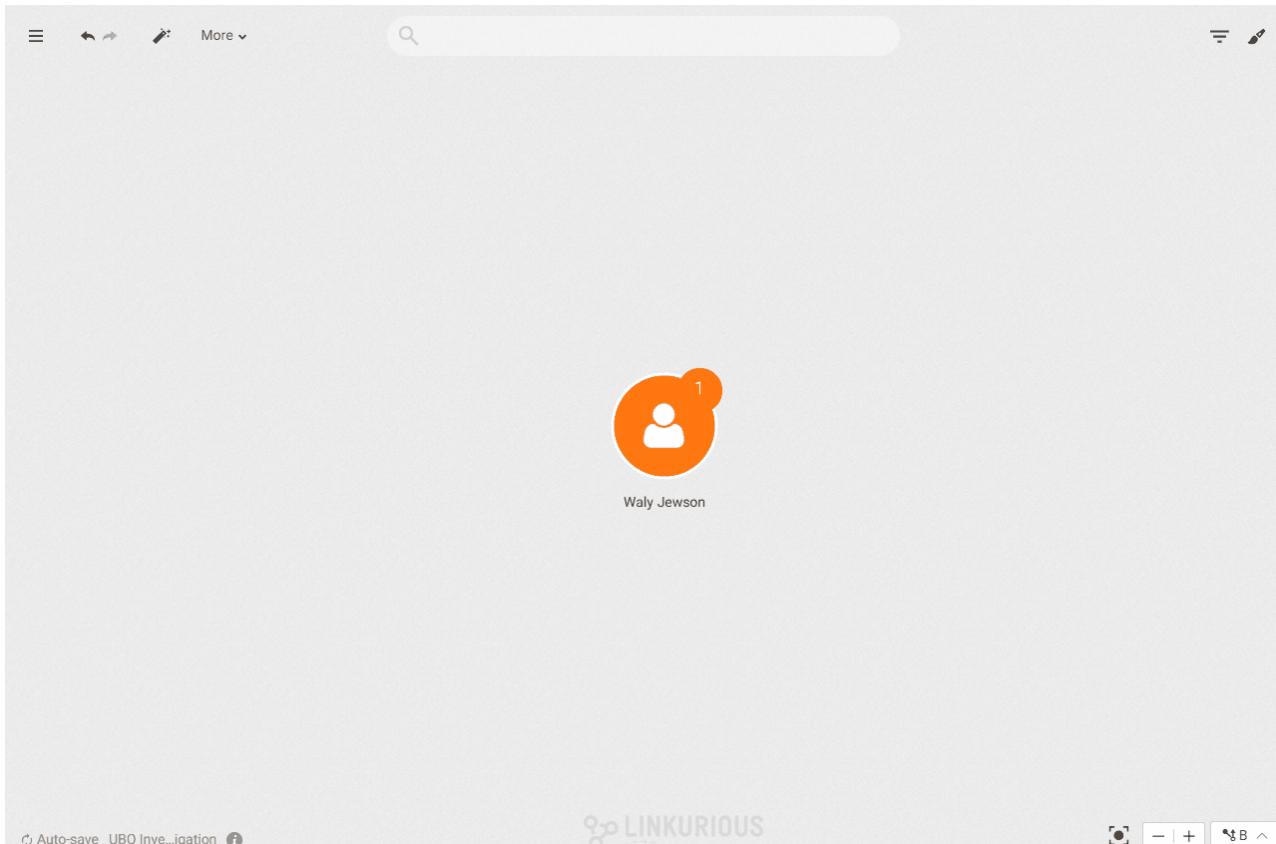
You can visualize data stored in Azure Cosmos DB Gremlin API by using various data visualization solutions. The following solutions are recommended by the [Apache Tinkerpop community](#) for graph data visualization.

## 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'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](#)

The screenshot shows a radial graph visualization centered on the movie 'The Matrix'. Nodes represent actors and crew members, including Carrie-Anne Moss, Anthony Ray Parker, Paul Goddard, Hugo Weaving, Gloria Foster, Matt Doran, Joe Pantoliano, Laurence Fishburne, Keanu Reeves, Marc Aden, Julian Arahanga, David Aston, Marcus Chong, and Belinda McClory. The 'KeyLines' logo is in the top right, and a 'Query' section shows the Cypher query: `g.V().hasLabel("Movie").has("title", "The Matrix").inE("ACTED_IN").outV().hasLabel("Person").path()`. Below it are 'Layouts' options: Standard, Hierarchy, Radial, and Structural.

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.

Philip Alton	Robert Benson	Stacy Blackman	Andrew Black	Jeff Bling	Jonathan Blodgett	Joe Quennet	Goeff Story
John Blawie	Mark Bowles	Jeffrey Blodgett	John Blodgett	John Blodgett	Steven Blodgett	Punk Blodgett	Colleen Nathan-Bloodgood
Christine Blodgett	Danny Blodgett	Mark Blodgett	Mike Blodgett	Annie Blodgett	Ashley Blodgett	Day Blodgett	Julia Blodgett
Scott Blodgett	Terriane Blodgett	Chris Blodgett	Jeff Blodgett	Gwynn Blodgett	Stephanie Blodgett	Andrea Blodgett	Jane Blodgett
John Blodgett	Jay Blodgett	Mark Blodgett	Mike Blodgett	Vince Blodgett	Jean Blodgett	Ebony Blodgett	Paul Blodgett
Bill Blodgett	Michelle Blodgett	Frank Blodgett	Laurie Blodgett	Terry Blodgett	Marc Blodgett	Benjamin Blodgett	Judy Blodgett
John Blodgett	Natalie Blodgett	Joseph Blodgett	Matthew Blodgett	Carrie Blodgett	Joan Blodgett	Karen Blodgett	Melissa Blodgett
Harry Blodgett	Sarah Blodgett	David Blodgett	Jeff Blodgett	Jeff Blodgett	Hannah Blodgett	Mike Blodgett	Myley Blodgett
Kathy Blodgett	Mark Blodgett	Mark Blodgett	Mark Blodgett	Mark Blodgett	Mark Blodgett	Mark Blodgett	Mark Blodgett
Robert Blodgett	Bobby Blodgett	John Blodgett	John Blodgett	John Blodgett	John Blodgett	John Blodgett	John Blodgett
Timothy Blodgett	Christopher Blodgett	William Blodgett	Mark Blodgett	Mark Blodgett	Mark Blodgett	Mark Blodgett	Mark Blodgett
Debra Blodgett	Larry Blodgett	David Blodgett	Mark Blodgett	Paul Blodgett	Stephanie Blodgett	Susan Blodgett	Kimberly Blodgett
Julian Blodgett	Mike Blodgett	Mark Blodgett	Mark Blodgett	Mark Blodgett	Mark Blodgett	Mark Blodgett	Mark Blodgett
Mark Blodgett	Roger Blodgett	Mark Blodgett	Mark Blodgett	Mark Blodgett	Mark Blodgett	Mark Blodgett	Mark Blodgett
Robert Blodgett	Mark Blodgett	Mark Blodgett	Mark Blodgett	Mark Blodgett	Mark Blodgett	Mark Blodgett	Mark Blodgett
Doug Blodgett	Sean Blodgett	Julie Blodgett	Keith Blodgett	Cynthia Blodgett	Francesca Blodgett	Caroline Blodgett	Andy Blodgett
Katheryn Blodgett	Mark Blodgett	Mark Blodgett	Mark Blodgett	Mark Blodgett	Mark Blodgett	Mark Blodgett	Mark Blodgett
Don Blodgett	Kurtis Blodgett	Belinda Blodgett	Michael Blodgett	Leanne Blodgett	Phoebe Blodgett	Carolyn Blodgett	Carolyn Blodgett

## 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

2/24/2020 • 4 minutes to read • [Edit Online](#)

This article describes how to perform different operations to automate management of your Azure Cosmos DB accounts, databases and containers using Azure Resource Manager templates. 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.

## Create Azure Cosmos DB API for MongoDB account, database and collection

Create Azure Cosmos DB resources using an Azure Resource Manager template. This template will create an Azure Cosmos account for Gremlin API with two graphs that share 400 RU/s throughput at the database level. Copy the template and deploy as shown below or visit [Azure Quickstart Gallery](#) and deploy from the Azure portal. You can also download the template to your local computer or create a new template and specify the local path with the `--template-file` parameter.

### NOTE

Account names must be lowercase and 44 or fewer characters. To update RU/s, resubmit the template with updated throughput property values.

```
{
    "$schema": "https://schema.management.azure.com/schemas/2015-01-01/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"
        }
    }
}
```

```
        "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."
        }
    },
    "multipleWriteLocations": {
        "type": "bool",
        "defaultValue": false,
        "allowedValues": [
            true,
            false
        ],
        "metadata": {
            "description": "Enable multi-master to make all regions writable."
        }
    },
    "automaticFailover": {
        "type": "bool",
        "defaultValue": false,
        "allowedValues": [
            true,
            false
        ],
        "metadata": {
            "description": "Enable automatic failover for regions. Ignored when Multi-Master is enabled"
        }
    },
    "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": "[toLowerCase(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": "2019-08-01",
            "location": "[parameters('location')]",
            "kind": "GlobalDocumentDB",
            "properties": {
                "capabilities": [
                    {
                        "name": "EnableGremlin"
                    }
                ],
                "consistencyPolicy": "[variables('consistencyPolicy')[parameters('defaultConsistencyLevel')]]",
                "locations": "[variables('locations')]",
                "databaseAccountOfferType": "Standard",
                "enableAutomaticFailover": "[parameters('automaticFailover')]",
                "enableMultipleWriteLocations": "[parameters('multipleWriteLocations')]"
            }
        },
        {
            "type": "Microsoft.DocumentDB/databaseAccounts/gremlinDatabases",
            "name": "[concat(variables('accountName'), '/', parameters('databaseName'))]",
            "apiVersion": "2019-08-01",
            "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": "2019-08-01",
    "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')]"
            }
        }
    }
}
]
}

```

## Deploy with the Azure CLI

To deploy the Azure Resource Manager template using the Azure CLI, **Copy** the script and select **Try it** to open Azure Cloud Shell. To paste the script, right-click the shell, and then select **Paste**:

```
read -p 'Enter the Resource Group name: ' resourceGroupName
read -p 'Enter the location (i.e. westus2): ' location
read -p 'Enter the account name: ' accountName
read -p 'Enter the primary region (i.e. westus2): ' primaryRegion
read -p 'Enter the secondary region (i.e. eastus2): ' secondaryRegion
read -p 'Enter the database name: ' databaseName
read -p 'Enter the first graph name: ' graph1Name
read -p 'Enter the second graph name: ' graph2Name

az group create --name $resourceGroupName --location $location
az group deployment create --resource-group $resourceGroupName \
    --template-uri https://raw.githubusercontent.com/azure/azure-quickstart-templates/master/101-cosmosdb-
gremlin/azuredeploy.json \
    --parameters accountName=$accountName primaryRegion=$primaryRegion secondaryRegion=$secondaryRegion
databaseName=$databaseName \
    graph1Name=$graph1Name graph2Name=$graph2Name

az cosmosdb show --resource-group $resourceGroupName --name accountName --output tsv
```

The `az cosmosdb show` command shows the newly created Azure Cosmos account after it has been provisioned. If you choose to use a locally installed version of the Azure CLI instead of using Cloud Shell, see the [Azure CLI](#) article.

## 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

12/5/2019 • 2 minutes to read • [Edit Online](#)

The following table includes links to sample Azure PowerShell scripts for Azure Cosmos DB for Gremlin API.

<a href="#">Create an account, database and graph</a>	Creates an Azure Cosmos account, database and graph.
<a href="#">List or get databases or graphs</a>	List or get database or graph.
<a href="#">Get RU/s</a>	Get RU/s for a database or graph.
<a href="#">Update RU/s</a>	Update RU/s for a database or graph.
<a href="#">Update an account or add a region</a>	Add a region to a Cosmos account. Can also be used to modify other account properties but these must be separate from changes to regions.
<a href="#">Change failover priority or trigger failover</a>	Change the regional failover priority of an Azure Cosmos account or trigger a manual failover.
<a href="#">Account keys or connection strings</a>	Get primary and secondary keys, connection strings or regenerate an account key of an Azure Cosmos account.
<a href="#">Create a Cosmos Account with IP Firewall</a>	Create an Azure Cosmos account with IP Firewall enabled.

# Azure CLI samples for Azure Cosmos DB Gremlin API

9/25/2019 • 2 minutes to read • [Edit Online](#)

The following table includes links to sample Azure CLI scripts for Azure Cosmos DB Gremlin API. Reference pages for all Azure Cosmos DB CLI commands are available in the [Azure CLI Reference](#). All Azure Cosmos DB CLI script samples can be found in the [Azure Cosmos DB CLI GitHub Repository](#).

<a href="#">Create an Azure Cosmos account, database and graph</a>	Creates an Azure Cosmos DB account, database, and graph for Gremlin API.
<a href="#">Change throughput</a>	Update RU/s on a database and graph.
<a href="#">Add or failover regions</a>	Add a region, change failover priority, trigger a manual failover.
<a href="#">Account keys and connection strings</a>	List account keys, read-only keys, regenerate keys and list connection strings.
<a href="#">Secure with IP firewall</a>	Create a Cosmos account with IP firewall configured.
<a href="#">Secure new account with service endpoints</a>	Create a Cosmos account and secure with service-endpoints.
<a href="#">Secure existing account with service endpoints</a>	Update a Cosmos account to secure with service-endpoints when the subnet is eventually configured.

# Introduction to Azure Cosmos DB: Table API

8/12/2019 • 2 minutes to read • [Edit Online](#)

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.

## 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:

	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 <a href="#">dedicated reserved throughput per table</a> 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.	<a href="#">Turnkey global distribution</a> from one to any number of regions. Support for <a href="#">automatic and manual failovers</a> at any time, anywhere in the world. Multi-master capability 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.

	AZURE TABLE STORAGE	AZURE COSMOS DB TABLE API
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. <a href="#">Comprehensive SLAs</a> 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

5/21/2019 • 7 minutes to read • [Edit Online](#)

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.

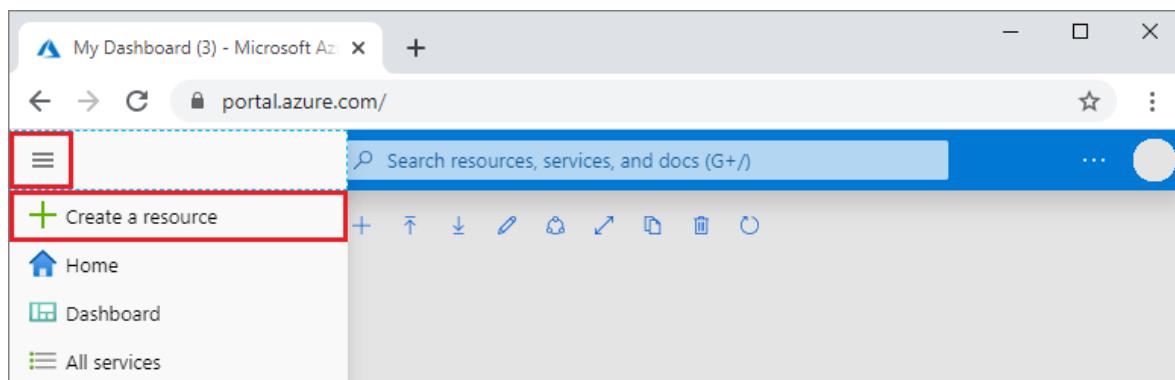
## Prerequisites

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**.

New

Search the Marketplace

Azure Marketplace [See all](#)

Featured [See all](#)

Get started		Azure SQL Managed Instance <a href="#">Quickstart tutorial</a>
Recently created		SQL Database <a href="#">Quickstart tutorial</a>
AI + Machine Learning		Azure Synapse Analytics (formerly SQL DW) <a href="#">Quickstart tutorial</a>
Analytics		Azure Database for MariaDB <a href="#">Learn more</a>
Blockchain		Azure Database for MySQL <a href="#">Quickstart tutorial</a>
Compute		Azure Database for PostgreSQL <a href="#">Quickstart tutorial</a>
Containers		Azure Cosmos DB <a href="#">Quickstart tutorial</a>
Databases		Azure Media Services <a href="#">Learn more</a>
Developer Tools		Azure Internet of Things <a href="#">Quickstart tutorial</a>
DevOps		Azure Media Services <a href="#">Learn more</a>
Identity		Azure Media Services <a href="#">Quickstart tutorial</a>
Integration		Azure Media Services <a href="#">Learn more</a>
Internet of Things		Azure Media Services <a href="#">Quickstart tutorial</a>
Media		Azure Media Services <a href="#">Learn more</a>
Mixed Reality		Azure Cosmos DB <a href="#">Quickstart tutorial</a>

- 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	<b>Create new</b> , then Account Name	Select <b>Create new</b> . 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	<p>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.</p> <p>Select <b>Azure Table</b>, because in this quickstart you are creating a table that works with the Table API.</p> <p><a href="#">Learn more about the Table API.</a></p>
Location	The region closest to your users	<p>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.</p>

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.

**Create Azure Cosmos DB Account**

[Basics](#) [Networking](#) [Tags](#) [Review + create](#)

Azure Cosmos DB is a globally distributed, multi-model, fully managed database service.

**Project Details**

Select the subscription to manage deployed resources and costs. Use resource groups like folders to organize and manage all your resources.

Subscription *	A Subscription
Resource Group *	Select existing... <a href="#">Create new</a>

**Instance Details**

Account Name *	Enter account name
API * ⓘ	Azure Table
Apache Spark ⓘ	Notebooks (preview) Notebooks with Apache Spark (preview) <a href="#">None</a> <a href="#">Sign up for Apache Spark preview</a>
Location *	(US) West US
Geo-Redundancy ⓘ	<a href="#">Enable</a> <a href="#">Disable</a>
Multi-region Writes ⓘ	<a href="#">Enable</a> <a href="#">Disable</a>

[Review + create](#) [Previous](#) [Next: Networking](#)

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**.

The screenshot shows the Azure portal interface for a Cosmos DB account named 'cosmos-db-table-quickstart'. On the left, a sidebar lists various account management options like Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Quick start (which is selected and highlighted in grey), Notifications, Data Explorer, and Settings (Account level throughput, Replicate data globally, Default consistency, Firewall and virtual networks, Private Endpoint Connections, Connection String, Add Azure Function). The main content area displays a success message: 'Congratulations! Your Azure Cosmos DB account was created.' Below it, a section titled 'Choose a platform' offers .NET, Node.js, Java, and Python. A numbered step 1 provides instructions to connect an existing Azure Table storage .NET app to the database, including a code snippet and a connection string input field.

## 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 screenshot shows the Data Explorer tool within the Azure portal for a database named 'ccdbtable'. The 'Data Explorer' option in the sidebar is highlighted with a red box. The main area displays the 'Add Table' dialog, which includes fields for 'Table Id' (set to 'Collection1') and 'Throughput' (set to '400'). A note at the bottom indicates an estimated spend of '\$0.032 hourly / \$0.77 daily'. The 'New Table' button in the top bar is also highlighted with a red box.

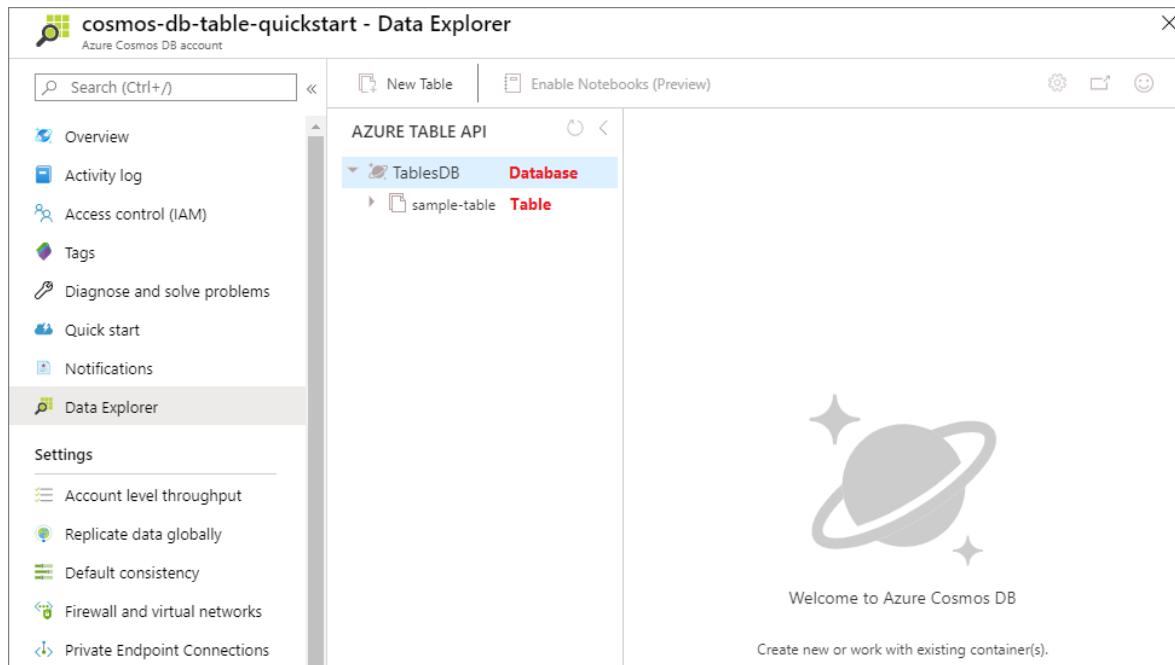
### 2. In the **Add Table** page, enter the settings for the new table.

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.

SETTING	SUGGESTED VALUE	DESCRIPTION
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**.

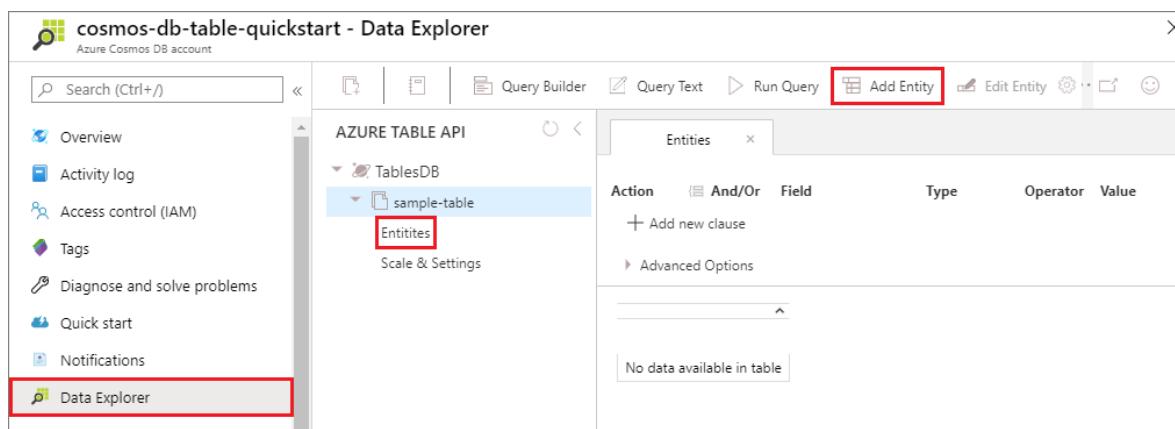
4. Data Explorer displays the new database and table.



## 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**.

The screenshot shows the Azure Data Explorer interface for a Cosmos DB account named 'cosmos-db-table-quickstart'. On the left, there's a sidebar with various navigation links. The main area is titled 'Add Table Entity' and contains a table with two rows: 'PartitionKey' and 'RowKey'. The 'PartitionKey' row has its 'Type' set to 'String' and its 'Value' set to 'Azure Cosmos DB'. The 'RowKey' row has its 'Type' set to 'String' and its 'Value' set to 'Replicate your data globally!'. There are edit icons next to each value. At the bottom of the dialog, there's a blue 'Add Entity' button.

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.

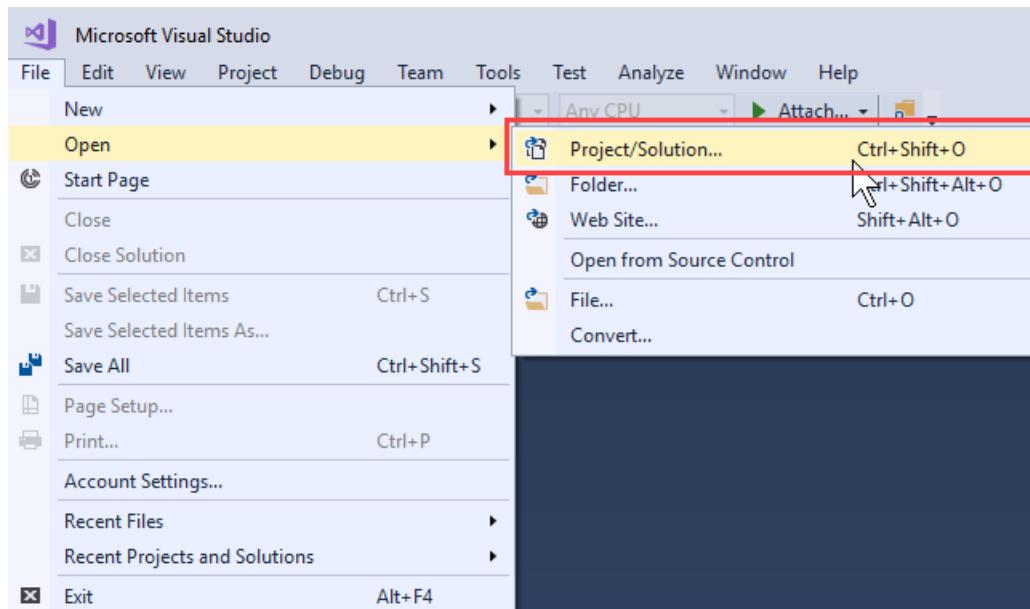
```
cd "C:\git-samples"
```

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
```

## 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.

## 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**.

The screenshot shows the Azure portal interface. The URL in the address bar is `portal.azure.com`. The main content area displays the 'cosmos-db-quickstart - Connection String' page for an 'Azure Cosmos DB account'. On the left, there's a sidebar with various icons and a 'Connection String' item highlighted with a red box. The main panel shows connection details like 'ACCOUNT NAME', 'ENDPOINT', 'PRIMARY KEY', 'SECONDARY KEY', and two connection strings. The 'PRIMARY CONNECTION STRING' field is also highlighted with a red box.

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.

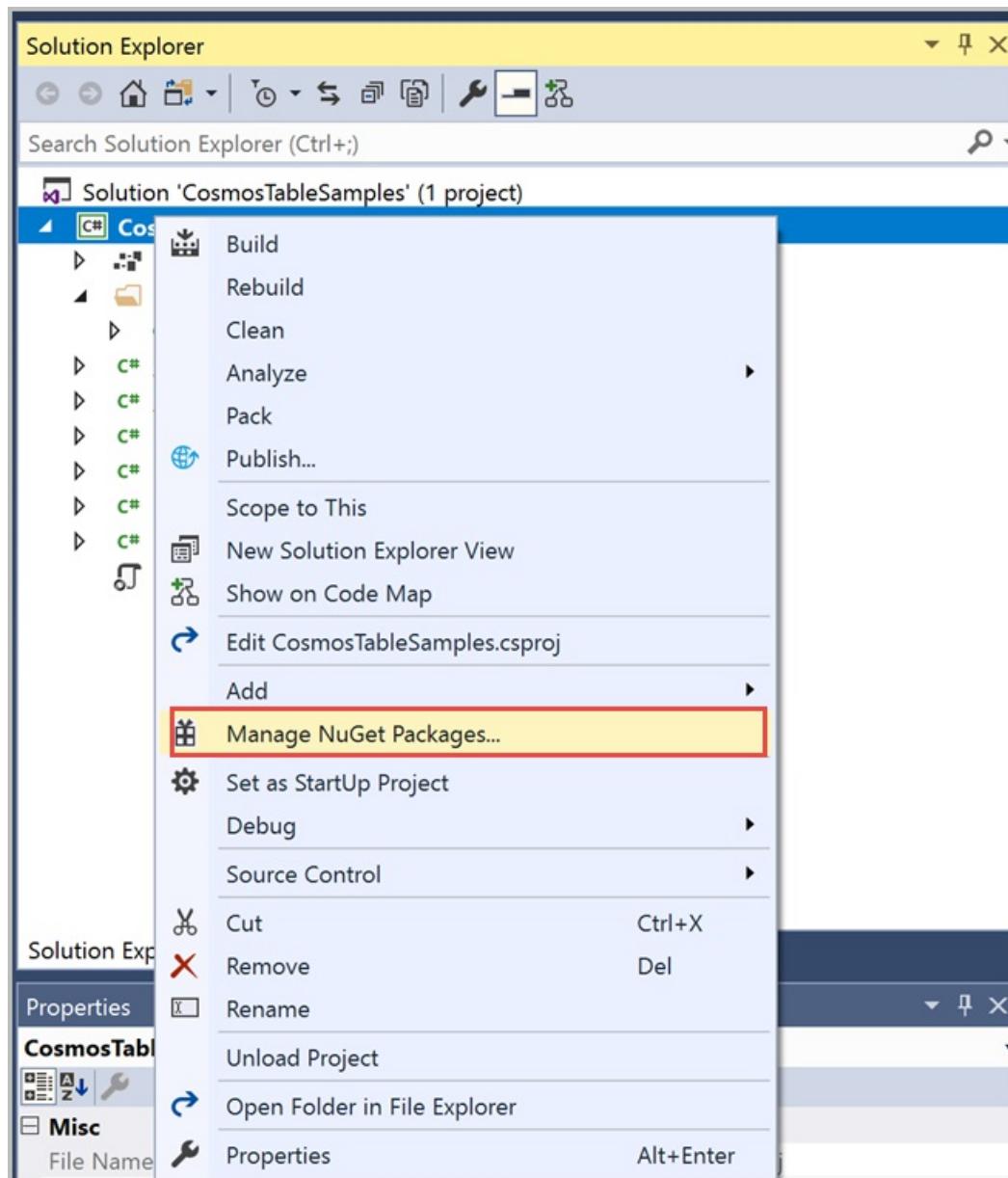
```
{  
    "StorageConnectionString": "<Primary connection string from Azure portal>"  
}
```

4. Press CTRL+S to save the **Settings.json** file.

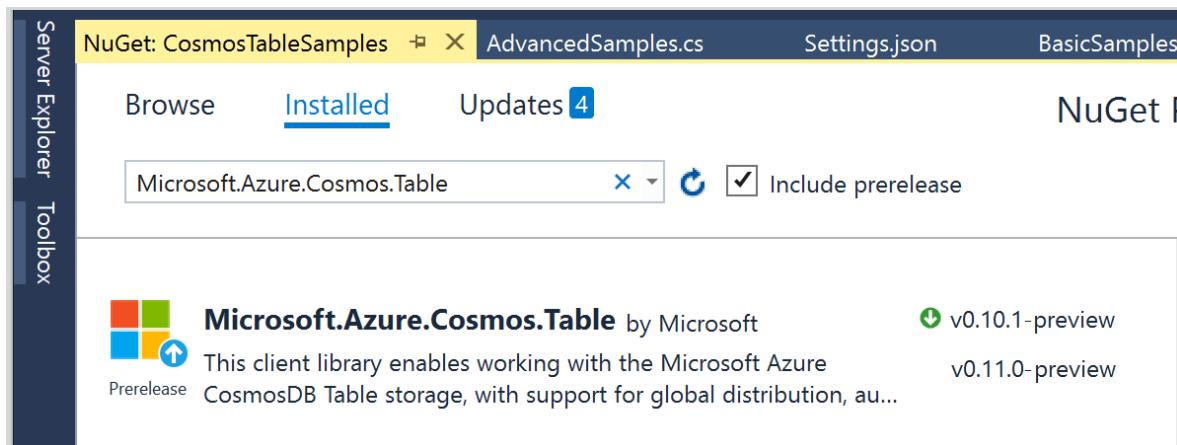
You've now updated your app with all the info it needs to communicate with Azure Cosmos DB.

## Build and deploy the app

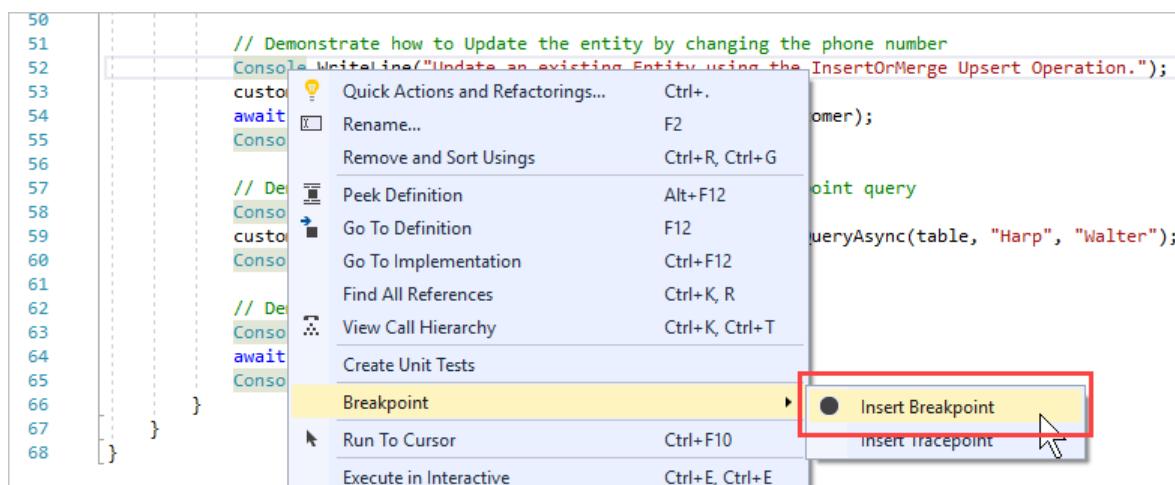
1. In Visual Studio, right-click on the **CosmosTableSamples** project in **Solution Explorer** and then click **Manage NuGet Packages**.



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.

```
C:\Program Files\dotnet\dotnet.exe
Azure Cosmos Table Samples
Azure Cosmos DB Table - Basic Samples

Create a Table for the demo
Created Table named: demoa13b1

Insert an Entity.
Update an existing Entity using the InsertOrMerge Upsert Operation.
```

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.

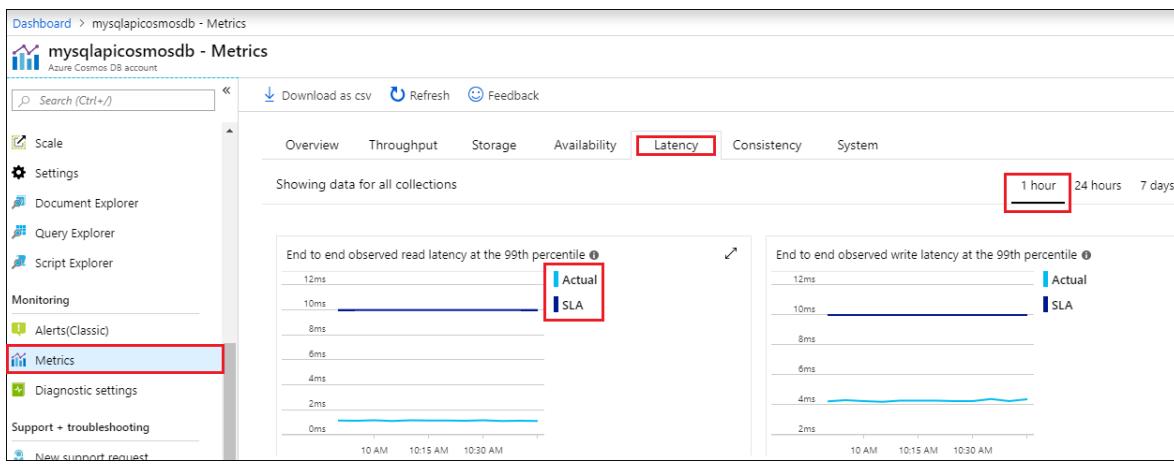
9. In the console window, press Enter to end the execution of the app.

## 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.

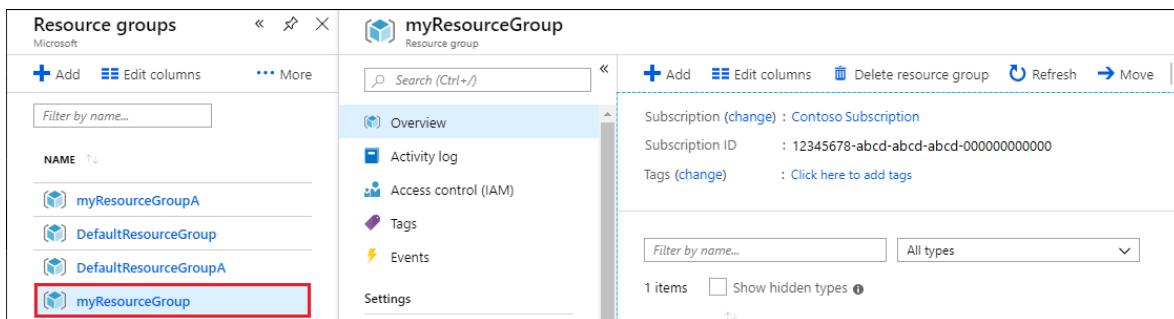


- Review the metrics on the other tabs.

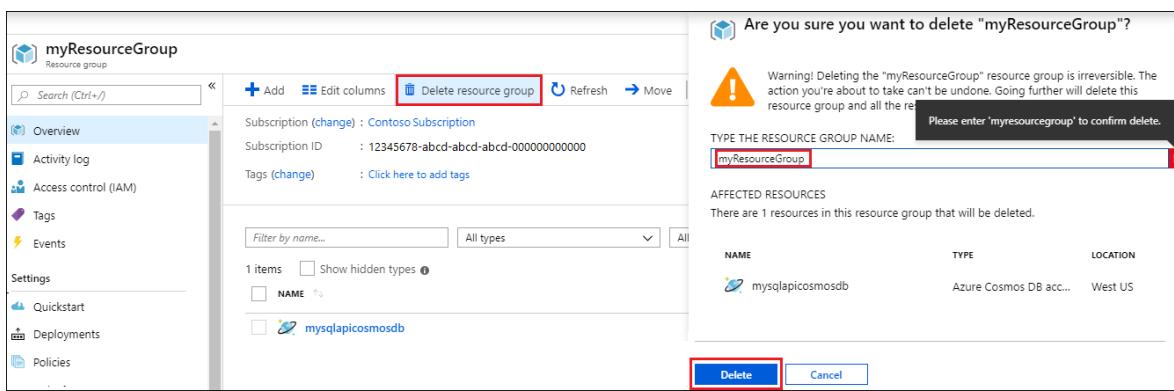
## Clean up resources

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:

- In the Azure portal Search bar, search for and select **Resource groups**.
- From the list, select the resource group you created for this quickstart.



- On the resource group **Overview** page, select **Delete resource group**.



- 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

2/14/2020 • 6 minutes to read • [Edit Online](#)

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+ob0N8A7Cgv30VRDJ1WEHLM+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](#).

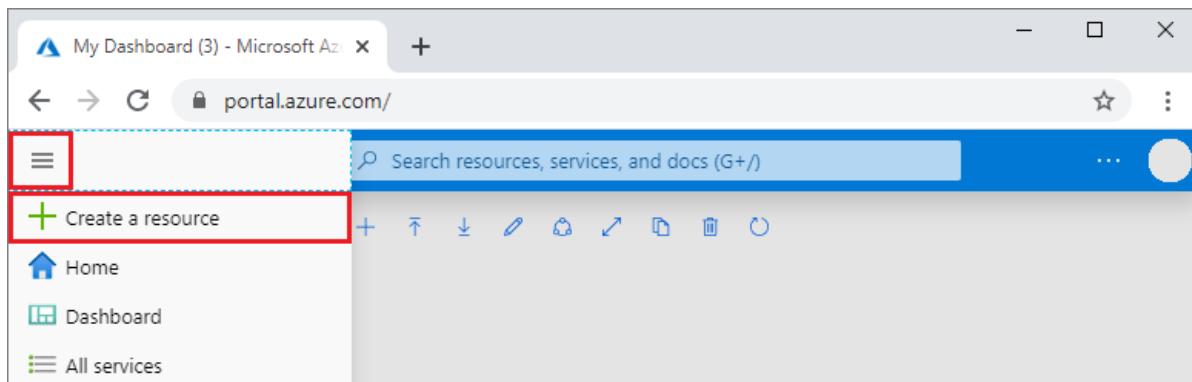
## Create a database account

### 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**.

New

Search the Marketplace

Azure Marketplace See all

Featured See all

Get started	 Azure SQL Managed Instance <a href="#">Quickstart tutorial</a>
Recently created	 SQL Database <a href="#">Quickstart tutorial</a>
AI + Machine Learning	 Azure Synapse Analytics (formerly SQL DW) <a href="#">Quickstart tutorial</a>
Analytics	
Blockchain	 Azure Database for MariaDB <a href="#">Learn more</a>
Compute	 Azure Database for MySQL <a href="#">Quickstart tutorial</a>
Containers	 Azure Database for PostgreSQL <a href="#">Quickstart tutorial</a>
Databases	 Azure Cosmos DB <a href="#">Quickstart tutorial</a>
Developer Tools	
DevOps	 Azure Database for PostgreSQL <a href="#">Quickstart tutorial</a>
Identity	
Integration	 Azure Cosmos DB <a href="#">Quickstart tutorial</a>
Internet of Things	
Media	
Mixed Reality	

- 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	<b>Create new</b> , then Account Name	Select <b>Create new</b> . 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	<p>Enter a unique name to identify your Azure Cosmos DB account.</p> <p>The account name can use only lowercase letters, numbers, and hyphens (-), and must be between 3 and 31 characters long.</p>

SETTING	VALUE	DESCRIPTION
API	Table	<p>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.</p> <p>Select <b>Azure Table</b>, because in this quickstart you are creating a table that works with the Table API.</p> <p><a href="#">Learn more about the Table API.</a></p>
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.

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.

**Create Azure Cosmos DB Account**

[Basics](#) [Networking](#) [Tags](#) [Review + create](#)

Azure Cosmos DB is a globally distributed, multi-model, fully managed database service.

**Project Details**

Select the subscription to manage deployed resources and costs. Use resource groups like folders to organize and manage all your resources.

Subscription *	A Subscription
Resource Group *	Select existing... <a href="#">Create new</a>

**Instance Details**

Account Name *	Enter account name
API * ⓘ	Azure Table
Apache Spark ⓘ	Notebooks (preview) <a href="#">Sign up for Apache Spark preview</a> Notebooks with Apache Spark (preview) <a href="#">None</a>
Location *	(US) West US
Geo-Redundancy ⓘ	Enable <a href="#">Disable</a>
Multi-region Writes ⓘ	Enable <a href="#">Disable</a>

[Review + create](#) [Previous](#) [Next: Networking](#)

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**.

Congratulations! Your Azure Cosmos DB account was created.

**Choose a platform**

- .NET**
- Node.js
- Java
- Python

**1 Connect your existing Azure Table storage .NET app**

You can use your existing Azure Table storage .NET app to work with Azure Cosmos DB. Here is an example:

```
using Microsoft.Azure.Storage;
using Microsoft.Azure.CosmosDB.Table;

var connectionString = "DefaultEndpointsProtocol=https;AccountName=cosmos-db-table-quickstart;AccountKey=J6Ef...";
CloudStorageAccount storageAccount = CloudStorageAccount.Parse(connectionString);
CloudTableClient tableClient = storageAccount.CreateCloudTableClient();
```

**CONNECTION STRING**

DefaultEndpointsProtocol=https;AccountName=cosmos-db-table-quickstart;AccountKey=J6Ef... [Copy](#)

**2 Learn More**

[Documentation](#) [Pricing](#) [Forum](#)

## 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**.

**ccdbtable - Data Explorer**

**New Table**

**Add Table**

\* Table Id

\* Throughput (400 - 1,000,000 RU/s)  - +

Estimated spend (USD): \$0.032 hourly / \$0.77 daily.

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.

SETTING	SUGGESTED VALUE	DESCRIPTION
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**.

The screenshot shows the Azure Cosmos DB Data Explorer interface. On the left, there's a sidebar with various options like Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Quick start, Notifications, and Data Explorer. The Data Explorer option is selected and highlighted with a grey background. The main area is titled 'Add Table Entity' and contains a table with two rows. The first row has 'Property Name' as 'PartitionKey', 'Type' as 'String', and 'Value' as 'Azure Cosmos DB'. The second row has 'Property Name' as 'RowKey', 'Type' as 'String', and 'Value' as 'Replicate your data globally!'. There are edit icons next to each value cell. At the bottom of the table is a blue 'Add Entity' button, which is also highlighted with a red box.

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
```

## 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**.

The screenshot shows the Azure Cosmos DB portal with the title "cosmos-db-table-quickstart - Connection String". On the left, there's a sidebar with various options like Lags, Diagnose and solve problems, Quick start, Notifications, Data Explorer, Settings, Account level throughput, Replicate data globally, Default consistency, Firewall and virtual networks, Private Endpoint Connections, and Connection String. The "Connection String" option is selected and highlighted with a red box. The main pane shows the connection details:

- ACCOUNT NAME: cosmos-db-table-quickstart
- ENDPOINT: https://cosmos-db-table-quickstart.table.cosmos.azure.com:443/
- PRIMARY KEY: J6EfA3AonDx5lAPrsxAjgFAWunTMMhqv3iBDf8Gvf48LnEDEc5A6J8D8Flv9e2Pi==
- SECONDARY KEY: ubQ7JtWozxIsI8pDeM3WDroS5mf7z3LJD0dbg98Za9LJ2kdhDVx3nTSbBOhxL=
- PRIMARY CONNECTION STRING: DefaultEndpointsProtocol=https;AccountName=cosmos-db-table-quickstart;AccountKey=J6EfA3AonDx5lAPrs...
- SECONDARY CONNECTION STRING: DefaultEndpointsProtocol=https;AccountName=cosmos-db-table-quickstart;AccountKey=ubQ7JtWozxIsI8pDe...

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.

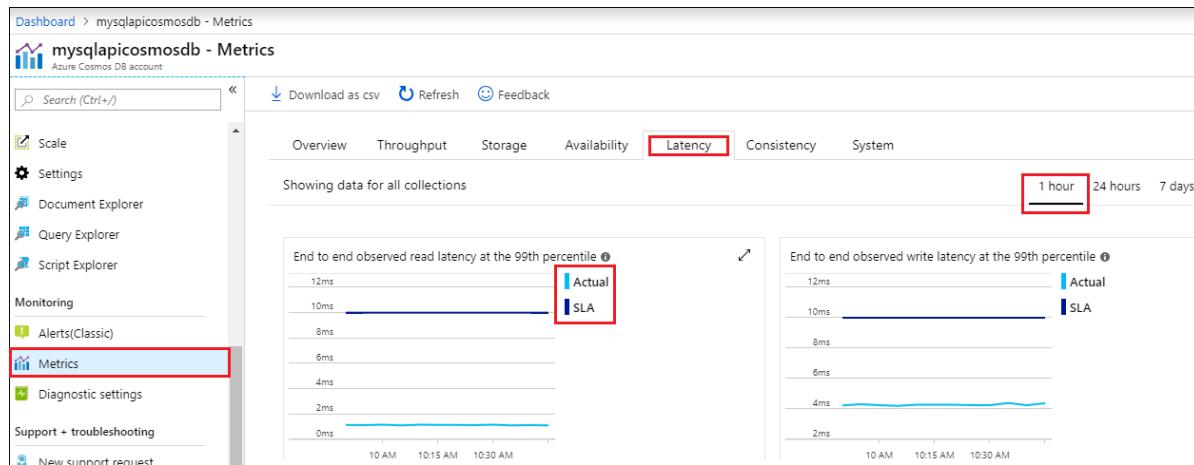
You can now go back to Data Explorer and see, query, modify, and work with this new data.

# 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.

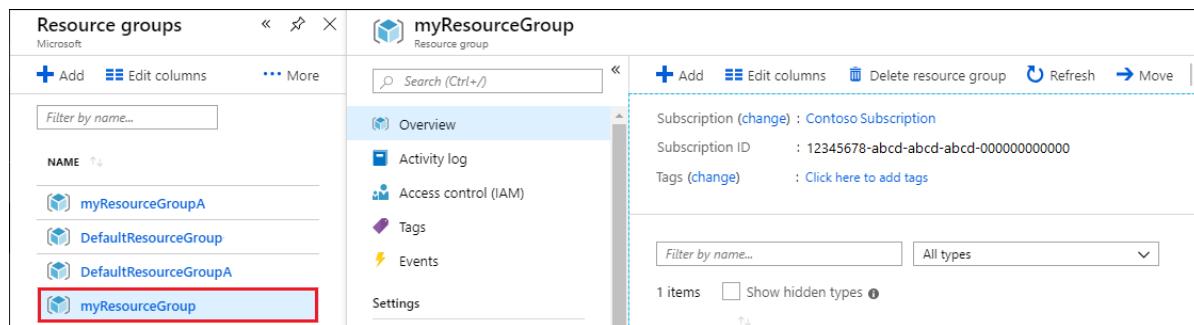


3. Review the metrics on the other tabs.

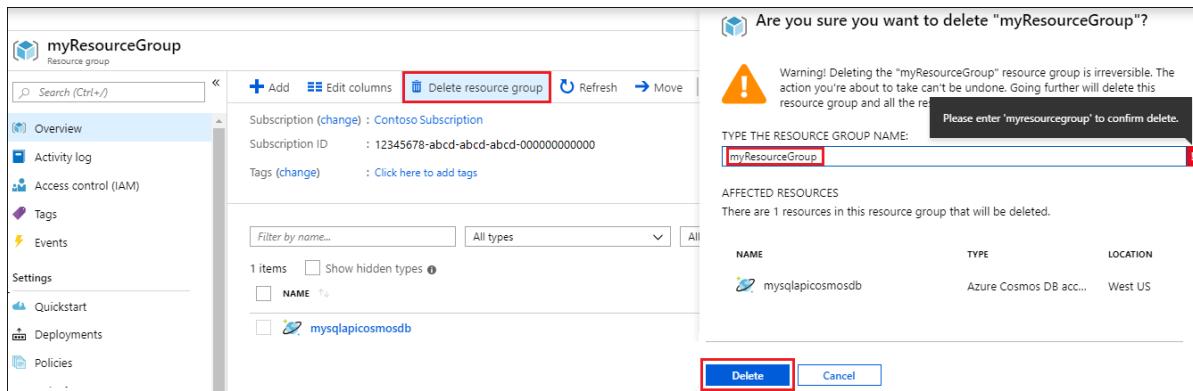
## Clean up resources

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

2/14/2020 • 6 minutes to read • [Edit Online](#)

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

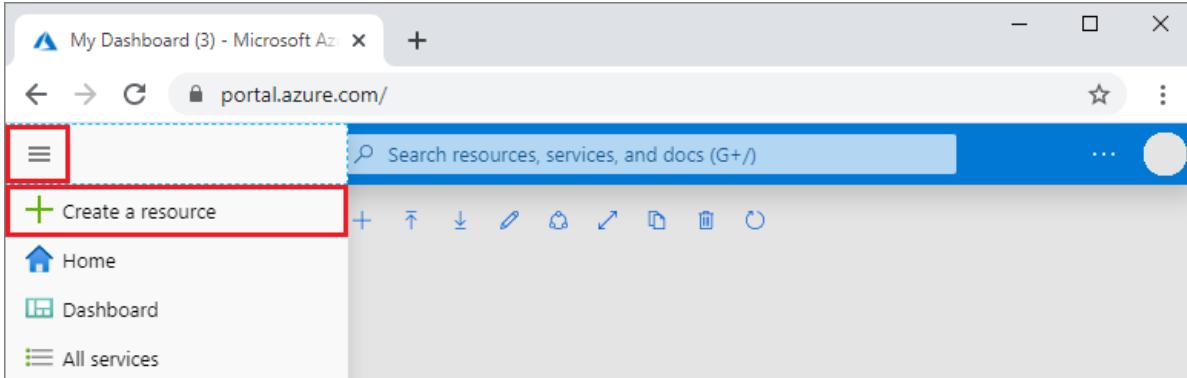
- An Azure account with an active subscription. [Create one for free](#). Or [try Azure Cosmos DB for free](#) without an Azure subscription. You can also use the [Azure Cosmos DB Emulator](#) with a URI of `https://localhost:8081` and the key `c2y6yDjf5/R+ob0N8A7Cgv30VRDJIWEHLM+4QDU5DE2nQ9nDuVTqobD4b8mGGyPMbIZnqyMsEcaGQy67XIw/Jw==`.
- [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**.

New

Search the Marketplace

Azure Marketplace See all

Featured See all

Get started	 Azure SQL Managed Instance <a href="#">Quickstart tutorial</a>
Recently created	 SQL Database <a href="#">Quickstart tutorial</a>
AI + Machine Learning	 Azure Synapse Analytics (formerly SQL DW) <a href="#">Quickstart tutorial</a>
Analytics	 Azure Database for MariaDB <a href="#">Learn more</a>
Blockchain	 Azure Database for MySQL <a href="#">Quickstart tutorial</a>
Compute	 Azure Database for PostgreSQL <a href="#">Quickstart tutorial</a>
Containers	 Azure Cosmos DB <a href="#">Quickstart tutorial</a>
Databases	 Azure Media Services <a href="#">Learn more</a>
Developer Tools	 Azure Mixed Reality <a href="#">Learn more</a>
DevOps	 Azure Identity <a href="#">Learn more</a>
Identity	 Azure Integration <a href="#">Learn more</a>
Integration	 Azure Internet of Things <a href="#">Learn more</a>
Internet of Things	 Azure Media Services <a href="#">Learn more</a>
Media	 Azure Mixed Reality <a href="#">Learn more</a>
Mixed Reality	 Azure Cosmos DB <a href="#">Quickstart tutorial</a>

- 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	<b>Create new</b> , then Account Name	Select <b>Create new</b> . 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	<p>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.</p> <p>Select <b>Azure Table</b>, because in this quickstart you are creating a table that works with the Table API.</p> <p><a href="#">Learn more about the Table API.</a></p>
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.

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.

**Create Azure Cosmos DB Account**

[Basics](#) [Networking](#) [Tags](#) [Review + create](#)

Azure Cosmos DB is a globally distributed, multi-model, fully managed database service.

**Project Details**

Select the subscription to manage deployed resources and costs. Use resource groups like folders to organize and manage all your resources.

Subscription \* [A Subscription](#)

Resource Group \* [Select existing...](#) [Create new](#)

**Instance Details**

Account Name \*

API \* [Azure Table](#) (selected)

Apache Spark [Notebooks \(preview\)](#) [Notebooks with Apache Spark \(preview\)](#) [None](#) [Sign up for Apache Spark preview](#)

Location \* [\(US\) West US](#)

Geo-Redundancy [Enable](#) [Disable](#)

Multi-region Writes [Enable](#) [Disable](#)

[Review + create](#) [Previous](#) [Next: Networking](#)

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**.

The screenshot shows the 'cosmos-db-table-quickstart - Quick start' page for an Azure Cosmos DB account. On the left, a sidebar lists various management options like Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Quick start (which is selected), Notifications, Data Explorer, and Settings. The main content area displays a success message: 'Congratulations! Your Azure Cosmos DB account was created.' Below this, a section titled 'Choose a platform' offers links for .NET (selected), Node.js, Java, and Python. A numbered list '1 Connect your existing Azure Table storage .NET app' provides sample code for connecting to Azure Cosmos DB using the .NET Storage API:

```
using Microsoft.Azure.Storage;
using Microsoft.Azure.CosmosDB.Table;

var connectionString = "DefaultEndpointsProtocol=https;AccountName=cosmos-db-table-quickstart;AccountKey=J6Ef...";
CloudStorageAccount storageAccount = CloudStorageAccount.Parse(connectionString);
CloudTableClient tableClient = storageAccount.CreateCloudTableClient();
```

Below the code, a 'CONNECTION STRING' field contains the same connection string, with a copy icon next to it. A second numbered list '2 Learn More' offers links to Documentation, Pricing, and Forum.

## 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.

The screenshot shows the 'ccdbtable - Data Explorer' page for an Azure Cosmos DB account. The sidebar includes 'Overview', 'Activity log', 'Access control (IAM)', 'Tags', 'Diagnose and solve problems', 'Quick start', 'Notifications', and 'Data Explorer' (which is selected and highlighted with a red box). The main area features a 'New Table' button (also highlighted with a red box) and an 'Azure Table API' section. To the right, the 'Add Table' dialog is open, prompting for 'Table Id' (with a placeholder 'e.g., Collection1') and 'Throughput' (set to 400). A note indicates an estimated spend of '\$0.032 hourly / \$0.77 daily'.

### 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.

SETTING	SUGGESTED VALUE	DESCRIPTION
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.

The screenshot shows the Azure Cosmos DB Data Explorer interface. On the left, there's a sidebar with various navigation options like Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Quick start, Notifications, and Data Explorer. The Data Explorer option is selected and highlighted. In the main area, under the 'AZURE TABLE API' section, the 'TablesDB' database is expanded, showing the 'sample-table' table. A large, stylized planet with a ring and stars is centered in the background. Below it, the text 'Welcome to Azure Cosmos DB' and 'Create new or work with existing container(s.)' is displayed.

## 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**.

This screenshot shows the same Data Explorer interface as the previous one, but with a focus on the 'sample-table' entities. The 'Entities' option under 'sample-table' is highlighted with a red box. In the top right toolbar, the 'Add Entity' button is also highlighted with a red box. The rest of the interface and sidebar are identical to the first screenshot.

2. Now add data to the PartitionKey value box and RowKey value box, and select **Add Entity**.

The screenshot shows the 'cosmos-db-table-quickstart - Data Explorer' window. On the left, there's a sidebar with links like Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Quick start, Notifications, and Data Explorer (which is selected). The main area is titled 'Add Table Entity' and shows two properties: 'PartitionKey' (String type) and 'RowKey' (String type). The 'PartitionKey' value is set to 'Azure Cosmos DB', and the 'RowKey' value is 'Replicate your data globally!'. There are edit icons next to each value. At the bottom right of the main area is a blue 'Add Entity' button, which is also highlighted with a red box.

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
```

## 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**.

The screenshot shows the Azure portal interface for a Cosmos DB account named "cosmos-db-table-quickstart". On the left, there's a sidebar with various options like "lags", "Diagnose and solve problems", "Quick start", "Notifications", "Data Explorer", "Settings", "Account level throughput", "Replicate data globally", "Default consistency", "Firewall and virtual networks", "Private Endpoint Connections", and "Connection String". The "Connection String" option is highlighted with a red box. The main pane shows the connection details:

- ACCOUNT NAME: cosmos-db-table-quickstart
- ENDPOINT: https://cosmos-db-table-quickstart.table.cosmos.azure.com:443/
- PRIMARY KEY: J6EfA3AonDx5lAprsxAjgFAWunTMMhqv3iBDf8Gvf48LnEDEc5A6J8D8Flv9e2Pi==
- SECONDARY KEY: ubQ7JtWozxIsI8pDeM3WDroS5mf7z3UDOdgb98Za9LJ2kdhDVx3nTSbBohxL==
- PRIMARY CONNECTION STRING: DefaultEndpointsProtocol=https;AccountName=cosmos-db-table-quickstart;AccountKey=J6EfA3AonDx5lAprs...
- SECONDARY CONNECTION STRING: DefaultEndpointsProtocol=https;AccountName=cosmos-db-table-quickstart;AccountKey=ubQ7JtWozxIsI8pDe...

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.

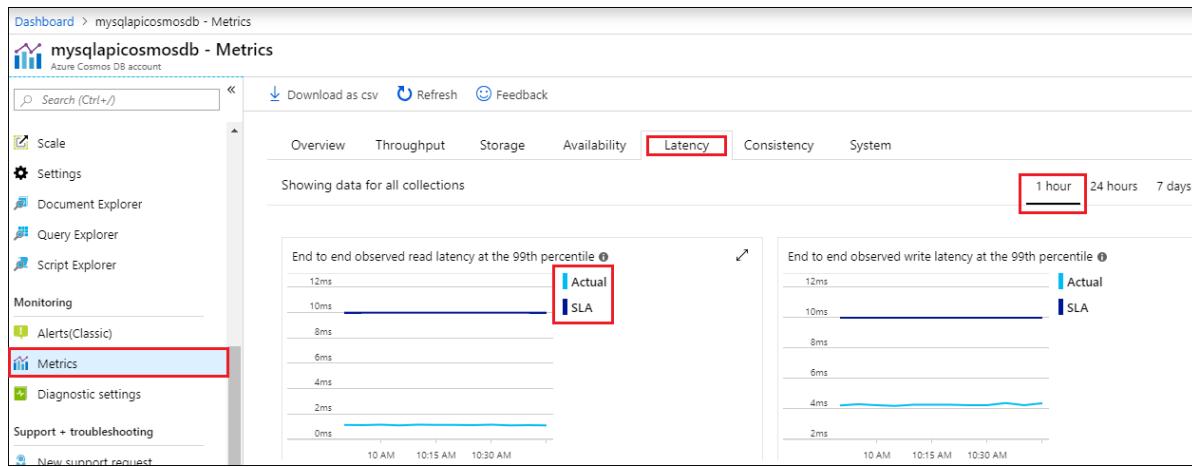
You can now go back to Data Explorer and see, query, modify, and work with this new data.

## 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.

The screenshot shows the 'Resource groups' blade in the Azure portal. On the left, a list of resource groups includes 'myResourceGroupA', 'DefaultResourceGroup', 'DefaultResourceGroupA', and 'myResourceGroup' (which is highlighted with a red box). On the right, the details for 'myResourceGroup' are shown, including the subscription ID and tags. A second window titled 'myResourceGroup' shows the 'Overview' tab with subscription information and a list of resources. The 'Delete resource group' button is highlighted with a red box.

3. On the resource group **Overview** page, select **Delete resource group**.

The screenshot shows a confirmation dialog for deleting the 'myResourceGroup'. It asks 'Are you sure you want to delete "myResourceGroup"?'. A warning message states: 'Warning! Deleting the "myResourceGroup" resource group is irreversible. The action you're about to take can't be undone. Going further will delete this resource group and all the resources it contains.' A text input field says 'Please enter 'myresourcegroup' to confirm delete.' The resource group name 'myResourceGroup' is typed into this field. Below, a table lists 'AFFECTED RESOURCES' with one item: 'mysqlapicosmosdb' (Azure Cosmos DB account, West US). The 'Delete' button is highlighted with a red box.

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](#)

# Quickstart: Build a Table API app with Python and Azure Cosmos DB

2/14/2020 • 5 minutes to read • [Edit Online](#)

In this quickstart, you create and manage an Azure Cosmos DB Table API account from the Azure portal, and from Visual Studio 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

- 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+ob0N8A7Cgv30VRDJTWEHLM+4QDU5DE2nQ9nDuVTqobD4b8mGGyPMbIZnqyMsEcaGQy67XIw/Jw==`.
- [Visual Studio 2019](#), with the **Azure development** and **Python development** workloads selected during setup.
- [Git](#).

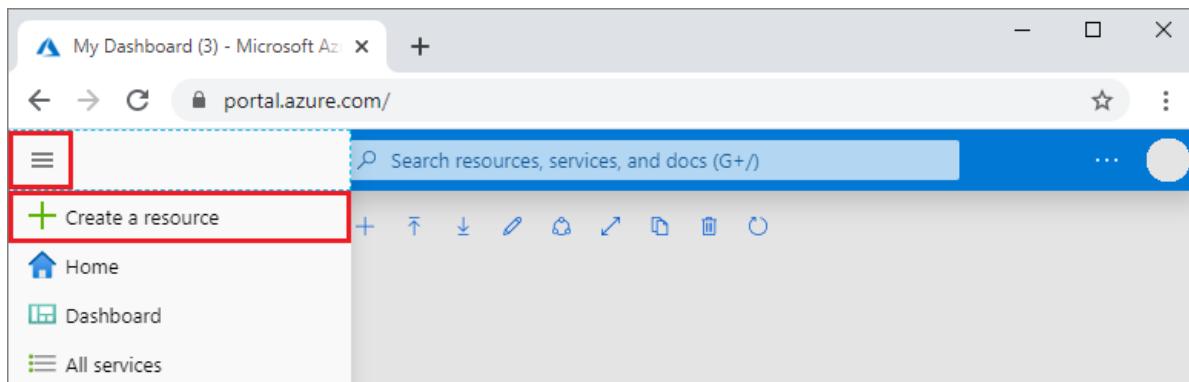
## Create a database account

### 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**.

New

Search the Marketplace

Azure Marketplace [See all](#)

Featured [See all](#)

Get started	 Azure SQL Managed Instance <a href="#">Quickstart tutorial</a>
Recently created	 SQL Database <a href="#">Quickstart tutorial</a>
AI + Machine Learning	 Azure Synapse Analytics (formerly SQL DW) <a href="#">Quickstart tutorial</a>
Analytics	
Blockchain	 Azure Database for MariaDB <a href="#">Learn more</a>
Compute	
Containers	 Azure Database for MySQL <a href="#">Quickstart tutorial</a>
Databases	 Azure Database for PostgreSQL <a href="#">Quickstart tutorial</a>
Developer Tools	 Azure Cosmos DB <a href="#">Quickstart tutorial</a>
DevOps	
Identity	
Integration	 Azure Cosmos DB <a href="#">Quickstart tutorial</a>
Internet of Things	
Media	
Mixed Reality	

- 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	<b>Create new</b> , then Account Name	Select <b>Create new</b> . 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	<p>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.</p> <p>Select <b>Azure Table</b>, because in this quickstart you are creating a table that works with the Table API.</p> <p><a href="#">Learn more about the Table API.</a></p>
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.

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.

**Create Azure Cosmos DB Account**

[Basics](#) [Networking](#) [Tags](#) [Review + create](#)

Azure Cosmos DB is a globally distributed, multi-model, fully managed database service.

**Project Details**

Select the subscription to manage deployed resources and costs. Use resource groups like folders to organize and manage all your resources.

Subscription *	A Subscription
Resource Group *	<input type="button" value="Select existing..."/> <input type="button" value="Create new"/>

**Instance Details**

Account Name *	Enter account name
API * ⓘ	Azure Table
Apache Spark ⓘ	<input type="button" value="Notebooks (preview)"/> <input type="button" value="Notebooks with Apache Spark (preview)"/> <input type="button" value="None"/> <a href="#">Sign up for Apache Spark preview</a>
Location *	(US) West US
Geo-Redundancy ⓘ	<input type="button" value="Enable"/> <input type="button" value="Disable"/>
Multi-region Writes ⓘ	<input type="button" value="Enable"/> <input type="button" value="Disable"/>

[Review + create](#) [Previous](#) [Next: Networking](#)

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**.

The screenshot shows the 'cosmos-db-table-quickstart - Quick start' page for an Azure Cosmos DB account. The left sidebar lists navigation options: Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Quick start (selected), Notifications, Data Explorer, Settings, Account level throughput, Replicate data globally, Default consistency, Firewall and virtual networks, Private Endpoint Connections, Connection String, and Add Azure Function. The main content area displays a success message: 'Congratulations! Your Azure Cosmos DB account was created.' Below it, a section titled 'Choose a platform' offers .NET, Node.js, Java, and Python. A step-by-step guide starts with '1 Connect your existing Azure Table storage .NET app', providing sample code for .NET:

```
using Microsoft.Azure.Storage;
using Microsoft.Azure.CosmosDB.Table;

var connectionString = "DefaultEndpointsProtocol=https;AccountName=cosmos-db-table-quickstart;AccountKey=J6Ef...";
CloudStorageAccount storageAccount = CloudStorageAccount.Parse(connectionString);
CloudTableClient tableClient = storageAccount.CreateCloudTableClient();
```

Below the code, a 'CONNECTION STRING' field contains the same connection string, with a copy icon next to it. Step 2, 'Learn More', includes links to Documentation, Pricing, and Forum.

## 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.

The screenshot shows the 'ccdbtable - Data Explorer' page for an Azure Cosmos DB account. The left sidebar has 'Data Explorer' selected. The main area shows the 'AZURE TABLE API' section and the 'Add Table' dialog. The 'New Table' button is highlighted with a red box. The 'Add Table' dialog fields include 'Table Id' (set to 'e.g., Collection1') and 'Throughput (400 - 1,000,000 RU/s)' (set to '400'). A note below states 'Estimated spend (USD): \$0.032 hourly / \$0.77 daily.'

### 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.

SETTING	SUGGESTED VALUE	DESCRIPTION
Throughput	400 RU/s	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**.

The screenshot shows the Azure Cosmos DB Data Explorer interface. On the left, there's a sidebar with various options like Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Quick start, Notifications, and Data Explorer. The Data Explorer option is selected and highlighted with a red box. The main area is titled 'Add Table Entity' and shows two properties: 'PartitionKey' and 'RowKey'. Both properties have their 'Type' set to 'String'. The 'Value' for 'PartitionKey' is 'Azure Cosmos DB', which is also highlighted with a red box. Below the properties, there's a note: 'Replicate your data globally!'. At the bottom of the dialog, there's a blue 'Add Entity' button, which is also highlighted with a red box.

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-python-getting-started.git
```

4. Then open the solution file in Visual Studio.

## 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**.

The screenshot shows the Azure portal's Connection String page for a Cosmos DB account. The 'Connection String' section is highlighted with a red box. It contains the following information:

- ACCOUNT NAME: cosmos-db-table-quickstart
- ENDPOINT: https://cosmos-db-table-quickstart.table.cosmos.azure.com:443/
- PRIMARY KEY: J6EfA3AonDx5lAPrsxAjgFAWunTMMhqv3iBDf8Gvf48LnEDEc5A6J8D8Flvf9e2Pi==
- SECONDARY KEY: ubQ7JtWozxIsI8pDeM3WDroS5mf7z3LJD0dbg98Za9LJ2kdhDVx3nTSbBOhxL==
- PRIMARY CONNECTION STRING: DefaultEndpointsProtocol=https;AccountName=cosmos-db-table-quickstart;AccountKey=J6EfA3AonDx5lAPrs...
- SECONDARY CONNECTION STRING: DefaultEndpointsProtocol=https;AccountName=cosmos-db-table-quickstart;AccountKey=ubQ7JtWozxIsI8pDe...

2. Copy the ACCOUNT NAME using the button on the right side.
3. Open the *config.py* file, and paste the ACCOUNT NAME from the portal into the STORAGE\_ACCOUNT\_NAME value on line 19.
4. Go back to the portal and copy the PRIMARY KEY.
5. Paste the PRIMARY KEY from the portal into the STORAGE\_ACCOUNT\_KEY value on line 20.
6. Save the *config.py* file.

## Run the app

1. In Visual Studio, right-click on the project in **Solution Explorer**.
2. Select the current Python environment, then right click.
3. Select **Install Python Package**, then enter *azure-storage-table*.
4. Press F5 to run the application. Your app displays in your browser.

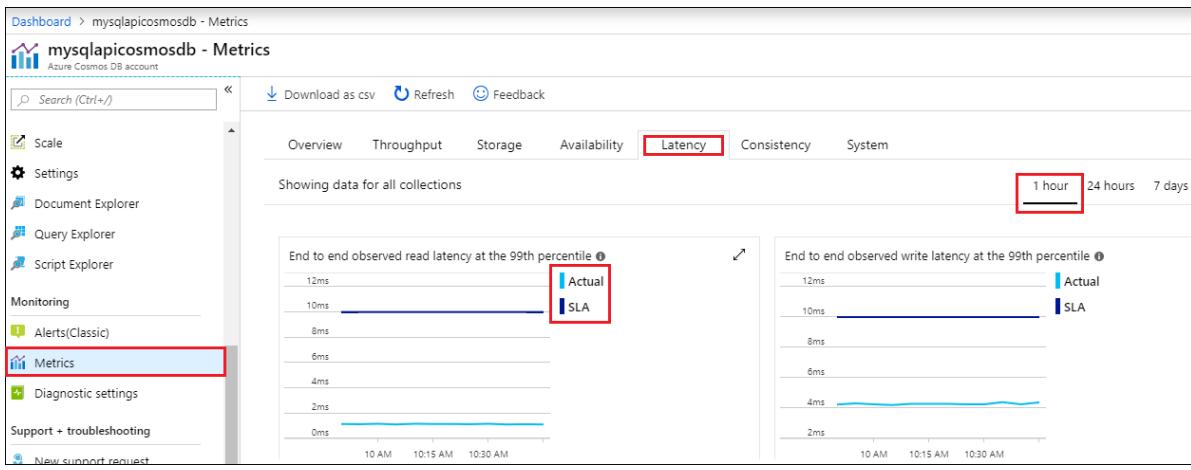
You can now go back to Data Explorer and see, query, modify, and work with this new data.

## 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.



- 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:

- In the Azure portal Search bar, search for and select **Resource groups**.
- From the list, select the resource group you created for this quickstart.

- On the resource group **Overview** page, select **Delete resource group**.

- 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 a Python app in Visual Studio 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 Cosmos DB Table API and Azure Table storage using the .NET SDK

1/26/2020 • 12 minutes to read • [Edit Online](#)

## 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

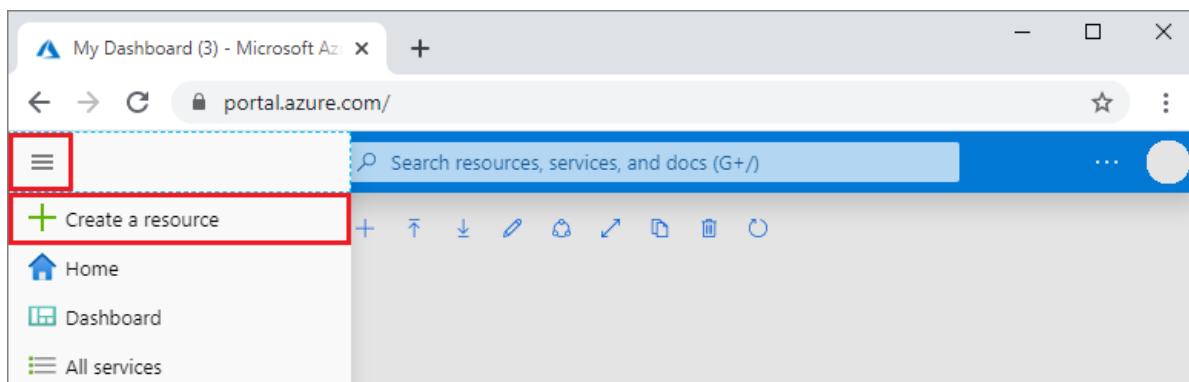
You need the following to complete this sample successfully:

- [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**.

New

Search the Marketplace

Azure Marketplace [See all](#)

Featured [See all](#)

Get started	 Azure SQL Managed Instance <a href="#">Quickstart tutorial</a>
Recently created	 SQL Database <a href="#">Quickstart tutorial</a>
AI + Machine Learning	 Azure Synapse Analytics (formerly SQL DW) <a href="#">Quickstart tutorial</a>
Analytics	
Blockchain	 Azure Database for MariaDB <a href="#">Learn more</a>
Compute	
Containers	 Azure Database for MySQL <a href="#">Quickstart tutorial</a>
Databases	 Azure Database for PostgreSQL <a href="#">Quickstart tutorial</a>
Developer Tools	
DevOps	 Azure Database for PostgreSQL <a href="#">Quickstart tutorial</a>
Identity	
Integration	 Azure Cosmos DB <a href="#">Quickstart tutorial</a>
Internet of Things	
Media	
Mixed Reality	 Azure Cosmos DB <a href="#">Quickstart tutorial</a>

- 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	<b>Create new</b> , then Account Name	Select <b>Create new</b> . 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	<p>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.</p> <p>Select <b>Azure Table</b>, because in this quickstart you are creating a table that works with the Table API.</p> <p><a href="#">Learn more about the Table API.</a></p>
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.

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.

**Create Azure Cosmos DB Account**

[Basics](#) [Networking](#) [Tags](#) [Review + create](#)

Azure Cosmos DB is a globally distributed, multi-model, fully managed database service.

**Project Details**

Select the subscription to manage deployed resources and costs. Use resource groups like folders to organize and manage all your resources.

Subscription *	A Subscription
Resource Group *	<input type="button" value="Select existing..."/> <input type="button" value="Create new"/>

**Instance Details**

Account Name *	Enter account name
API * ⓘ	Azure Table
Apache Spark ⓘ	<input type="button" value="Notebooks (preview)"/> <input type="button" value="Notebooks with Apache Spark (preview)"/> <input type="button" value="None"/> <a href="#">Sign up for Apache Spark preview</a>
Location *	(US) West US
Geo-Redundancy ⓘ	<input type="button" value="Enable"/> <input type="button" value="Disable"/>
Multi-region Writes ⓘ	<input type="button" value="Enable"/> <input type="button" value="Disable"/>

[Review + create](#) [Previous](#) [Next: Networking](#)

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**.

The screenshot shows the 'cosmos-db-table-quickstart - Quick start' page in the Azure portal. The left sidebar lists various options like Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Quick start (which is selected), Notifications, Data Explorer, Settings, Account level throughput, Replicate data globally, Default consistency, Firewall and virtual networks, Private Endpoint Connections, Connection String, and Add Azure Function. The main content area says 'Congratulations! Your Azure Cosmos DB account was created.' and 'Choose a platform' with tabs for .NET (selected), Node.js, Java, and Python. Step 1, 'Connect your existing Azure Table storage .NET app', provides sample code for .NET:

```
using Microsoft.Azure.Storage;
using Microsoft.Azure.CosmosDB.Table;

var connectionString = "DefaultEndpointsProtocol=https;AccountName=cosmos-db-table-quickstart;AccountKey=J6Ef...";
CloudStorageAccount storageAccount = CloudStorageAccount.Parse(connectionString);
CloudTableClient tableClient = storageAccount.CreateCloudTableClient();
```

Step 2, 'Learn More', includes links to Documentation, Pricing, and Forum.

## 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;

        // Get the request units consumed by the current operation. RequestCharge of a TableResult is only
        applied to Azure Cosmos DB
        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);
        }

        // Get the request units consumed by the current operation. RequestCharge of a TableResult is only
        applied to Azure Cosmos DB
        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);

        // Get the request units consumed by the current operation. RequestCharge of a TableResult is only applied
        to Azure Cosmos DB
        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 you wish to delete the entity and table at the end of the project remove the comments from `table.DeleteIfExistsAsync()` and `SamplesUtils.DeleteEntityAsync(table, customer)` methods from the following code:

```

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 from [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

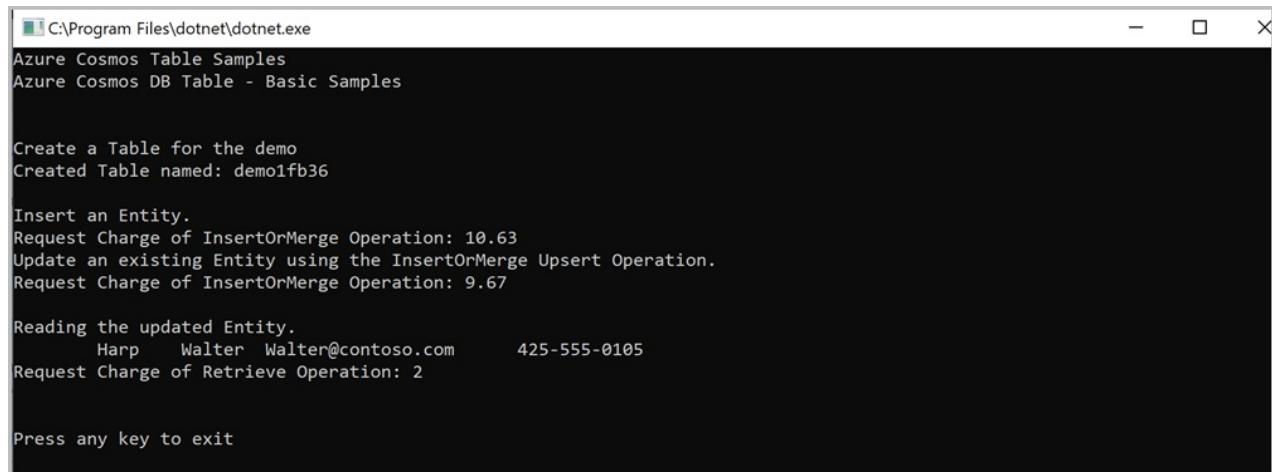
From your project **CosmosTableSamples**. Open the class named **Program.cs** and add the following code to it for calling BasicSamples when the project runs.

```
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();

            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:



```
C:\Program Files\dotnet\dotnet.exe
Azure Cosmos Table Samples
Azure Cosmos DB Table - Basic Samples

Create a Table for the demo
Created Table named: demo1fb36

Insert an Entity.
Request Charge of InsertOrMerge Operation: 10.63
Update an existing Entity using the InsertOrMerge Upsert Operation.
Request Charge of InsertOrMerge Operation: 9.67

Reading the updated Entity.
Harp Walter Walter@contoso.com 425-555-0105
Request Charge of Retrieve Operation: 2

Press any key to exit
```

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.

The screenshot shows the Azure Table API Entities blade. On the left, there's a navigation tree with 'TablesDB' selected, and 'demo8344a' is highlighted with a red box. The main area has tabs for 'Entities', 'Query Builder', 'Query Text', 'Run', 'Add Entity', 'Edit Entity', and 'Delete Entities'. Below these tabs, there's a query builder interface with two clauses: 'PartitionKey = Harp' and 'RowKey = Walter'. A button '+ Add new clause' is available. Below the clauses, there are columns for 'PartitionKey', 'RowKey', 'Timestamp', 'Email', and 'PhoneNumber'. A single row is displayed: 'Harp' (highlighted with a red box), 'Walter', 'Mon, 11 Mar 2019 23:06:10 GMT', 'Walter@contoso.com', and '425-555-0105'.

## Next steps

You can now proceed to the next tutorial and learn how to migrate data to Azure Cosmos DB Table API account.

[How to query data](#)

# Migrate your data to Azure Cosmos DB Table API account

5/23/2019 • 6 minutes to read • [Edit Online](#)

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>]
```

Options for the 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**.

NAME	KEY	CONNECTION STRING
key1	<key 1>	DefaultEndpointsProtocol=https;AccountName=functionsintegration;AccountKey=<key 1>;EndpointSuffix=core.windows.net
key2	<key 2>	DefaultEndpointsProtocol=https;AccountName=functionsintegration;AccountKey=<key 2>;EndpointSuffix=core.windows.net

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

1/26/2020 • 2 minutes to read • [Edit Online](#)

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

## 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

1/30/2020 • 3 minutes to read • [Edit Online](#)

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.

andrl - Replicate data globally  
Azure Cosmos DB account

Search (Ctrl+ /)

- Overview
- Activity log
- Access control (IAM)
- Tags
- Diagnose and solve problems
- Quick start
- Data Explorer

**SETTINGS**

- Keys
- Replicate data globally**
- Default consistency
- Firewall
- Add Azure Search
- Properties
- Locks
- Automation script

**MONITORING**

- Metrics
- Alert rules

Save Discard Manual Failover Failover Priorities

Click on a location to add or remove regions from your Azure Cosmos DB account  
\* Each region is billable based on the throughput and storage for the account. [Learn more](#)

**WRITE REGION**

- Central US

**READ REGIONS**

- Australia East
- Australia Southeast
- Brazil South
- Canada Central

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 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 a multi master account, 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

2 minutes to read

# How to use Azure Table storage or Azure Cosmos DB Table API from Java

1/26/2020 • 17 minutes to read • [Edit Online](#)

## 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.

## Overview

This article demonstrates how to perform common scenarios using the Azure Table storage service and Azure Cosmos DB. 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 Azure Table storage or Azure Cosmos DB. To learn more about the differences between the services, see [Table offerings](#). You'll need to create an account for the service you're going to use.

### 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 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://your
_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 4MB 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 for each 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 **TableQuery** 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 **EntityResolver**, 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]. Note that 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 **CloudTable** 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. Note that 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. For about 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][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

1/26/2020 • 13 minutes to read • [Edit Online](#)

## 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.

## Overview

This article shows how to perform common scenarios using Azure Storage Table service or Azure Cosmos DB in a Node.js application.

## Create an Azure service account

You can work with tables using Azure Table storage or Azure Cosmos DB. To learn more about the differences between the services, see [Table offerings](#). You'll need to create an account for the service you're going to use.

### 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 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.

## 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. Note that **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

1/26/2020 • 7 minutes to read • [Edit Online](#)

## 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.

Azure Table storage and 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 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 Table storage or 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 Azure Table storage or Azure Cosmos DB. To learn more about the differences between the services, see [Table offerings](#). You'll need to create an account for the service you're going to use.

### 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;TableEn  
dpoint=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

1/26/2020 • 11 minutes to read • [Edit Online](#)

## 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.

## Overview

This guide shows you how to perform common scenarios using the Azure Storage Table service and 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 Azure Table storage or Azure Cosmos DB. To learn more about the differences between the services, see [Table offerings](#). You'll need to create an account for the service you're going to use.

### 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 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 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/]";
```

## Add a Storage emulator connection

To access the emulator storage:

```
UseDevelopmentStorage = true
```

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++

1/26/2020 • 14 minutes to read • [Edit Online](#)

## 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 Azure Table storage or Azure Cosmos DB. To learn more about the differences between the services, see [Table offerings](#). You'll need to create an account for the service you're going to use.

### 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](#).

[Linux](#) page.

- **Windows:** On Windows, use [vcppkg](#) as the dependency manager. Follow the [quick-start](#) to initialize vcpkg. Then, use the following command to install the library:

```
.\\vcppkg.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

1/26/2020 • 6 minutes to read • [Edit Online](#)

## 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.

## Overview

This guide shows you how to perform common scenarios using Azure Table service and the Azure Cosmos DB Table API. The samples are written in Ruby and use the [Azure Storage Table Client Library for Ruby](#). The scenarios covered include **creating and deleting a table, and inserting and querying entities in a table**.

## Create an Azure service account

You can work with tables using Azure Table storage or Azure Cosmos DB. To learn more about the differences between the services, see [Table offerings](#). You'll need to create an account for the service you're going to use.

### 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 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 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](#)

# Developing with Azure Cosmos DB Table API and Azure Table storage

1/26/2020 • 3 minutes to read • [Edit Online](#)

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:

	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 <a href="#">dedicated reserved throughput per table</a> 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.	<a href="#">Turnkey global distribution</a> from one to 30+ regions. Support for <a href="#">automatic and manual failovers</a> 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.	<a href="#">Five well-defined consistency levels</a> 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 <a href="#">Industry-leading comprehensive SLAs</a> 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:

- [WindowsAzure.Storage .NET SDK](#). This library enables you to work with the storage Table service.
- [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.

# Azure Table storage table design guide: Scalable and performant tables

1/26/2020 • 74 minutes to read • [Edit Online](#)

## 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 border="1"> <thead> <tr> <th>FIRSTNAME</th><th>LASTNAME</th><th>AGE</th><th>EMAIL</th></tr> </thead> <tbody> <tr> <td>Don</td><td>Hall</td><td>34</td><td>donh@contoso.com</td></tr> </tbody> </table>	FIRSTNAME	LASTNAME	AGE	EMAIL	Don	Hall	34	donh@contoso.com
FIRSTNAME	LASTNAME	AGE	EMAIL								
Don	Hall	34	donh@contoso.com								
Marketing	00002	2014-08-22T00:50:34Z	<table border="1"> <thead> <tr> <th>FIRSTNAME</th><th>LASTNAME</th><th>AGE</th><th>EMAIL</th></tr> </thead> <tbody> <tr> <td>Jun</td><td>Caio</td><td>47</td><td>junc@contoso.com</td></tr> </tbody> </table>	FIRSTNAME	LASTNAME	AGE	EMAIL	Jun	Caio	47	junc@contoso.com
FIRSTNAME	LASTNAME	AGE	EMAIL								
Jun	Caio	47	junc@contoso.com								
Marketing	Department	2014-08-22T00:50:30Z	<table border="1"> <thead> <tr> <th>DEPARTMENTNAME</th><th>EMPLOYEECOUNT</th></tr> </thead> <tbody> <tr> <td>Marketing</td><td>153</td></tr> </tbody> </table>	DEPARTMENTNAME	EMPLOYEECOUNT	Marketing	153				
DEPARTMENTNAME	EMPLOYEECOUNT										
Marketing	153										
Sales	00010	2014-08-22T00:50:44Z	<table border="1"> <thead> <tr> <th>FIRSTNAME</th><th>LASTNAME</th><th>AGE</th><th>EMAIL</th></tr> </thead> <tbody> <tr> <td>Ken</td><td>Kwok</td><td>23</td><td>kenk@contoso.com</td></tr> </tbody> </table>	FIRSTNAME	LASTNAME	AGE	EMAIL	Ken	Kwok	23	kenk@contoso.com
FIRSTNAME	LASTNAME	AGE	EMAIL								
Ken	Kwok	23	kenk@contoso.com								

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 <code>PartitionKey</code> , <code>RowKey</code> , and <code>Timestamp</code> ).
Size of the <code>PartitionKey</code>	A string up to 1 KB in size.

<b>TOTAL CAPACITY OF AN AZURE STORAGE ACCOUNT</b>	<b>500 TB</b>
Size of the <code>RowKey</code>	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
<code>PartitionKey</code> (Department name)	String
<code>RowKey</code> (Employee ID)	String
<code>FirstName</code>	String
<code>LastName</code>	String
<code>Age</code>	Integer
<code>EmailAddress</code>	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:

Department entity	Employee entity
PartitionKey (Department name) RowKey ("Department") ManagerName (string) ManagerEmailAddress (string)	PartitionKey (Department name) RowKey (Email address) FirstName (string) LastName (string) Age (integer) Id (string)

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.

Employee entity
PartitionKey (Department name) RowKey (Email address) FirstName (string) LastName (string) Age (integer) Id (string) ManagerName (string) ManagerEmailAddress (string)

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	<ul style="list-style-type: none"> <li>You can update a department entity with a single operation.</li> <li>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.</li> </ul>	<ul style="list-style-type: none"> <li>You might need to retrieve both an employee and a department entity for some client activities.</li> <li>Storage operations happen in the same partition. At high transaction volumes, this can result in a hotspot.</li> <li>You can't move an employee to a new department by using an EGT.</li> </ul>
Separate entity types, different partitions, or tables or storage accounts	<ul style="list-style-type: none"> <li>You can update a department entity or employee entity with a single operation.</li> <li>At high transaction volumes, this can help spread the load across more partitions.</li> </ul>	<ul style="list-style-type: none"> <li>You might need to retrieve both an employee and a department entity for some client activities.</li> <li>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.</li> <li>You can't move an employee to a new department by using an EGT.</li> </ul>
Denormalize into single entity type	<ul style="list-style-type: none"> <li>You can retrieve all the information you need with a single request.</li> </ul>	<ul style="list-style-type: none"> <li>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).</li> </ul>

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

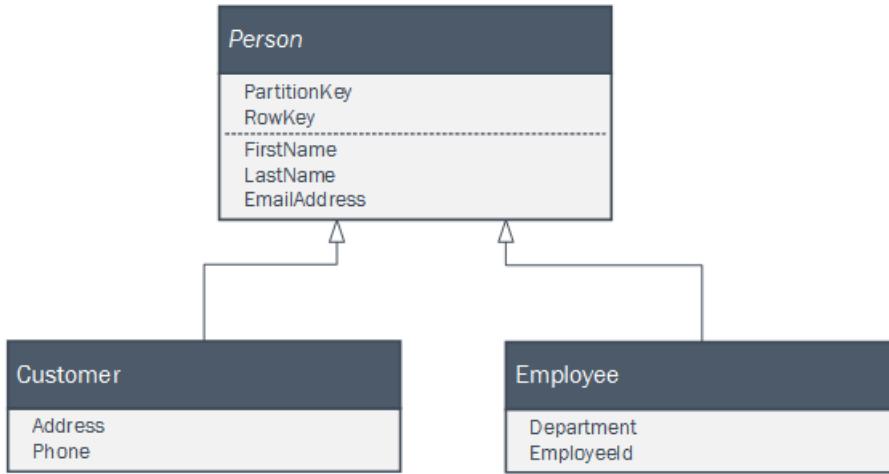
Although there are ways to model relationships in Table storage, don't forget that the two prime reasons for using Table storage are scalability and performance. If you find you are modeling many relationships that compromise the performance and scalability of your solution, you should ask yourself if 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

PartitionKey  
RowKey  
PersonType ("Customer")  
FirstName (string)  
LastName (string)  
EmailAddress (string)  
Address (string)  
Phone (string)

### Employee entity

PartitionKey  
RowKey  
PersonType ("Employee")  
FirstName (string)  
LastName (string)  
EmailAddress (string)  
Department (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.

Employee entity	Employee entity
<code>PartitionKey (Department name)</code> <code>RowKey ("empid_" + Employee Id)</code> ----- <code>FirstName (string)</code> <code>LastName (string)</code> <code>Age (integer)</code> <code>EmailAddress (string)</code>	<code>PartitionKey (Department name)</code> <code>RowKey ("email_" + Email address)</code> ----- <code>FirstName (string)</code> <code>LastName (string)</code> <code>Age (integer)</code> <code>EmployeeId (string)</code>

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:

Employee entity
<code>PartitionKey (Department name)</code> <code>RowKey ("email_" + Email address)</code> ----- <code>FirstName (string)</code> <code>LastName (string)</code> <code>EmployeeId (string)</code>

- 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](#)

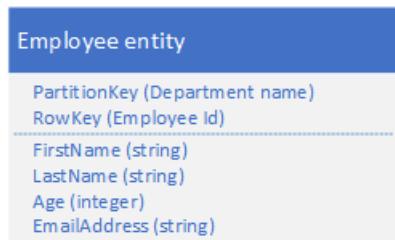
#### Inter-partition secondary index pattern

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.



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.

### Employee entity (primary index)

PartitionKey ("empid\_" + Department name)  
RowKey (Employee Id)  
FirstName (string)  
LastName (string)  
Age (integer)  
EmailAddress (string)

### Employee entity (secondary index)

PartitionKey ("email\_" + Department name)  
RowKey (Email address)  
FirstName (string)  
LastName (string)  
Age (integer)  
EmployeeId (string)

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:

### Employee entity (secondary index)

PartitionKey ("email\_" + Department name)  
RowKey (Email address)  
FirstName (string)  
LastName (string)  
EmployeeId (string)

- 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:

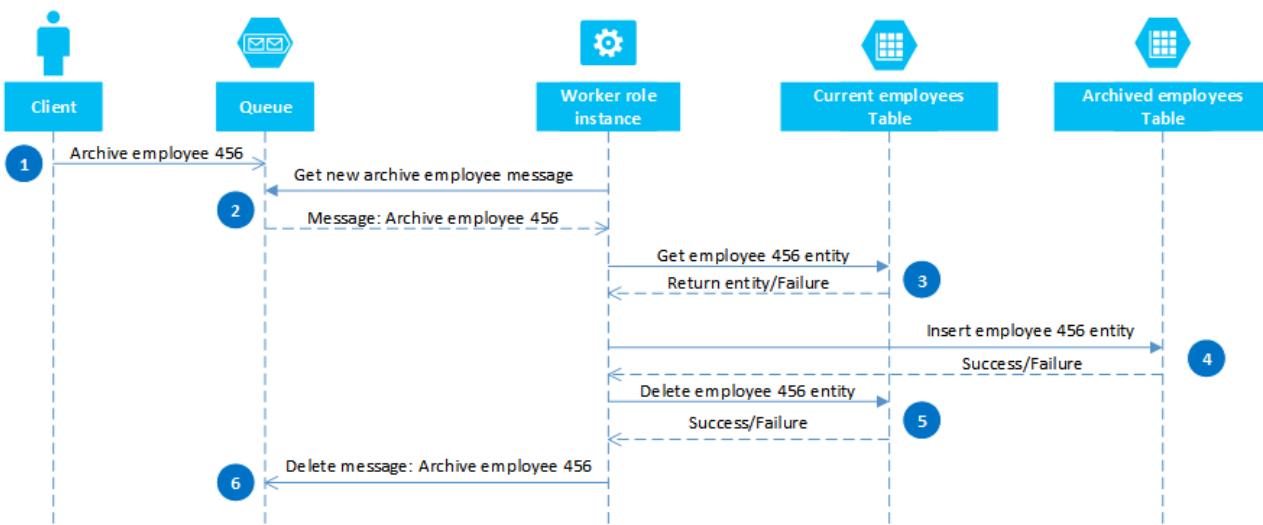
- Entities stored in two different partitions in the same table, in different tables, or in different storage accounts.
- An entity stored in Table storage and a blob stored in Blob storage.
- An entity stored in Table storage and a file in a file system.
- An entity stored in Table storage, yet indexed 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`.

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.

Department entity	Employee entity
<pre>PartitionKey (Department name) RowKey ("Department") ----- DepartmentName (string) EmployeeCount (integer) ManagerId (string - employee id of manager)</pre>	<pre>PartitionKey (Department name) RowKey (Employee Id) ----- FirstName (string) LastName (string) Age (integer) EmailAddress (string)</pre>

#### 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:

Department entity
<pre>PartitionKey (Department name) RowKey ("Department") ----- DepartmentName (string) EmployeeCount (integer) ManagerName (string) ManagerEmailAddress (string)</pre>

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:

## Employee entity

PartitionKey (Department name)  
RowKey (Employee Id + Year)  
FirstName (string)  
LastName (string)  
Age (integer)  
EmailAddress (string)  
ManagerRating (integer)  
PeerRating (integer)  
Comments (string)

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:

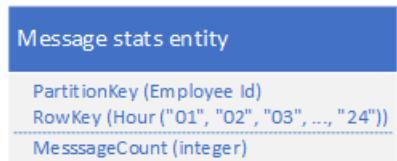
- [Entity group transactions](#)
- [Modify entities](#)

### Data series pattern

Store complete data series in a single entity to minimize the number of requests you make.

#### Context and problem

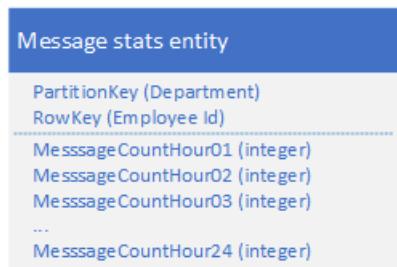
A common scenario is for an application to store a series of data that it typically needs to retrieve all at once. For example, your application might record how many IM messages each employee sends every hour, and then use this information to plot how many messages each user sent over the preceding 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:

Message stats entity	Message stats entity
<pre>PartitionKey (Department) RowKey (Employee Id + ".01") MessageCountDay01 (integer) MessageCountDay02 (integer) MessageCountDay03 (integer) ... MessageCountDay252 (integer)</pre>	<pre>PartitionKey (Department) RowKey (Employee Id + ".02") MessageCountDay253 (integer) MessageCountDay254 (integer) MessageCountDay255 (integer) ... MessageCountDay365 (integer)</pre>

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

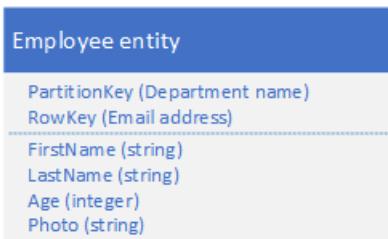
Use Blob storage to store large property values.

## 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:

## Employee entity

PartitionKey (Year + Month + Day + Hour)  
RowKey (Employee Id + Event Id)

FirstName (string)  
LastName (string)  
EventType (string)  
EventTimestamp (datetime)  
EventText (string)

### Solution

The following alternative entity structure avoids a hotspot on any particular partition, as the application logs events:

## Employee entity

PartitionKey (Department name + Employee Id)  
RowKey (Year + Month + Day + Hour + Event Id)

FirstName (string)  
LastName (string)  
EventType (string)  
EventTimestamp (datetime)  
EventText (string)

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
<code>PartitionKey</code> (Date and hour of log message)
<code>RowKey</code> (Time of log message and unique log message id (GUID))
<code>Severity</code> (integer)
<code>Source</code> (string)
<code>Message</code> (string)
<code>Department</code> (string)
<code>ClientIP</code> (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
<code>PartitionKey</code> (Source)
<code>RowKey</code> (Time of log message and unique log message id (GUID))
<code>Severity</code> (integer)
<code>Message</code> (string)
<code>Department</code> (string)
<code>ClientIP</code> (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.GreaterThanOrEqualTo, "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
--------------	--------	-----------


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	EN TIT YT YP E	FIR ST NA ME	LA ST NA ME	AG E	EM AIL
			E m p l o y e e				

	<table border="1"> <thead> <tr> <th>EN</th><th>FIR</th><th>LA</th><th>AG</th><th>EM</th></tr> <tr> <th>TIT</th><th>ST</th><th>ST</th><th>E</th><th>AIL</th></tr> <tr> <th>YT</th><th>NA</th><th>NA</th><th></th><th></th></tr> <tr> <th>YP</th><th>ME</th><th>ME</th><th></th><th></th></tr> <tr> <th>E</th><th></th><th></th><th></th><th></th></tr> <tr> <td>Employee</td><td></td><td></td><td></td><td></td></tr> </thead> </table>	EN	FIR	LA	AG	EM	TIT	ST	ST	E	AIL	YT	NA	NA			YP	ME	ME			E					Employee						
EN	FIR	LA	AG	EM																													
TIT	ST	ST	E	AIL																													
YT	NA	NA																															
YP	ME	ME																															
E																																	
Employee																																	

ENTITY TYPE	DEPARTMENTNAME	EMPLOYEECOUNT
Department		

EN	FIR	LA	AG	EM
TIT	ST	ST	E	AIL
YT	NA	NA		
YP	ME	ME		
E				
Employee				

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 about [Inheritance 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.GreaterThanOrEqualTo, "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

2/24/2020 • 4 minutes to read • [Edit Online](#)

This article describes how to perform different operations to automate management of your Azure Cosmos DB accounts, databases and containers using Azure Resource Manager templates. 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.

## Create Azure Cosmos account and table

Create Azure Cosmos DB resources using an Azure Resource Manager template. This template will create an Azure Cosmos account for Table API with one table at 400 RU/s throughput. Copy the template and deploy as shown below or visit [Azure Quickstart Gallery](#) and deploy from the Azure portal. You can also download the template to your local computer or create a new template and specify the local path with the `--template-file` parameter.

### NOTE

Account names must be lowercase and 44 or fewer characters. To update RU/s, resubmit the template with updated throughput property values.

```
{  
  "$schema": "https://schema.management.azure.com/schemas/2015-01-01/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",  
      "metadata": {  
        "description": "The default consistency level for the Cosmos DB account."  
      }  
    }  
  },  
  "resources": [  
    {  
      "type": "Microsoft.DocumentDB/databaseAccounts",  
      "name": "[parameters('accountName')]",  
      "apiVersion": "2017-02-28",  
      "location": "[parameters('location')]",  
      "properties": {  
        "consistencyPolicy": {  
          "consistentPrefix": true  
        },  
        "locations": [ {  
          "name": "[parameters('primaryRegion')]",  
          "latency": 0,  
          "type": "Primary"  
        }, {  
          "name": "[parameters('secondaryRegion')]",  
          "latency": 100,  
          "type": "Secondary"  
        } ]  
      }  
    },  
    {  
      "type": "Microsoft.DocumentDB/databaseAccounts/containers",  
      "name": "[concat(parameters('accountName'), '/Tables')]",  
      "apiVersion": "2017-02-28",  
      "dependsOn": [ "[resourceId('Microsoft.DocumentDB/databaseAccounts', parameters('accountName'))]" ],  
      "properties": {  
        "partitionKeyPath": "/id",  
        "throughput": 400  
      }  
    }  
  ]  
}
```

```
"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."
    }
},
},
"multipleWriteLocations": {
    "type": "bool",
    "defaultValue": false,
    "allowedValues": [ true, false ],
    "metadata": {
        "description": "Enable multi-master to make all regions writable."
    }
},
},
"automaticFailover": {
    "type": "bool",
    "defaultValue": false,
    "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": "2019-08-01",
  "location": "[parameters('location')]",
  "kind": "GlobalDocumentDB",
  "properties": {
    "capabilities": [{ "name": "EnableTable" }],
    "consistencyPolicy": "[variables('consistencyPolicy')[parameters('defaultConsistencyLevel')]]",
    "locations": "[variables('locations')]",
    "databaseAccountOfferType": "Standard",
    "enableAutomaticFailover": "[parameters('automaticFailover')]",
    "enableMultipleWriteLocations": "[parameters('multipleWriteLocations')]"
  }
},
{
  "type": "Microsoft.DocumentDB/databaseAccounts/tables",
  "name": "[concat(variables('accountName'), '/', parameters('tableName'))]",
  "apiVersion": "2019-08-01",
  "dependsOn": [ "[resourceId('Microsoft.DocumentDB/databaseAccounts/', variables('accountName'))]" ],
  "properties": {
    "resource": {
      "id": "[parameters('tableName')]"
    },
    "options": { "throughput": "[parameters('throughput')]" }
  }
}
]
}

```

## Deploy via PowerShell

To deploy the Resource Manager template using PowerShell, **Copy** the script and select **Try it** to open Azure Cloud Shell. To paste the script, right-click the shell, and then select **Paste**:

```

$resourceGroupName = Read-Host -Prompt "Enter the Resource Group name"
$accountName = Read-Host -Prompt "Enter the account name"
$location = Read-Host -Prompt "Enter the location (i.e. westus2)"
$primaryRegion = Read-Host -Prompt "Enter the primary region (i.e. westus2)"
$secondaryRegion = Read-Host -Prompt "Enter the secondary region (i.e. eastus2)"
$tableName = Read-Host -Prompt "Enter the table name"

New-AzResourceGroup -Name $resourceGroupName -Location $location
New-AzResourceGroupDeployment ` 
    -ResourceGroupName $resourceGroupName ` 
    -TemplateUri "https://raw.githubusercontent.com/Azure/azure-quickstart-templates/master/101-cosmosdb-` 
    table/azuredeploy.json" ` 
    -primaryRegion $primaryRegion ` 
    -secondaryRegion $secondaryRegion ` 
    -tableName $tableName

(Get-AzResource --ResourceType "Microsoft.DocumentDb/databaseAccounts" --ApiVersion "2015-04-08" --` 
ResourceGroupName $resourceGroupName).name

```

If you choose to use a locally installed version of PowerShell instead of from the Azure Cloud shell, you have to [install](#) the Azure PowerShell module. Run `Get-Module -ListAvailable Az` to find the version.

## Deploy via the Azure CLI

To deploy the Azure Resource Manager template using the Azure CLI, [Copy](#) the script and select **Try it** to open Azure Cloud Shell. To paste the script, right-click the shell, and then select **Paste**:

```

read -p 'Enter the Resource Group name: ' resourceGroupName
read -p 'Enter the location (i.e. westus2): ' location
read -p 'Enter the account name: ' accountName
read -p 'Enter the primary region (i.e. westus2): ' primaryRegion
read -p 'Enter the secondary region (i.e. eastus2): ' secondaryRegion
read -p 'Enter the table name: ' tableName

az group create --name $resourceGroupName --location $location
az group deployment create --resource-group $resourceGroupName \
    --template-uri https://raw.githubusercontent.com/Azure/azure-quickstart-templates/master/101-cosmosdb-` 
    table/azuredeploy.json \
    --parameters accountName=$accountName primaryRegion=$primaryRegion secondaryRegion=$secondaryRegion
tableName=$tableName

az cosmosdb show --resource-group $resourceGroupName --name accountName --output tsv

```

The `az cosmosdb show` command shows the newly created Azure Cosmos account after it has been provisioned. If you choose to use a locally installed version of the Azure CLI instead of using Cloud Shell, see the [Azure CLI](#) article.

## 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

12/5/2019 • 2 minutes to read • [Edit Online](#)

The following table includes links to sample Azure PowerShell scripts for Azure Cosmos DB for Table API.

<a href="#">Create an account and table</a>	Creates an Azure Cosmos account and table.
<a href="#">List or get tables</a>	List or get tables.
<a href="#">Get RU/s</a>	Get RU/s for a table.
<a href="#">Update RU/s</a>	Update RU/s for a table.
<a href="#">Update an account or add a region</a>	Add a region to a Cosmos account. Can also be used to modify other account properties but these must be separate from changes to regions.
<a href="#">Change failover priority or trigger failover</a>	Change the regional failover priority of an Azure Cosmos account or trigger a manual failover.
<a href="#">Account keys or connection strings</a>	Get primary and secondary keys, connection strings or regenerate an account key of an Azure Cosmos account.
<a href="#">Create a Cosmos Account with IP Firewall</a>	Create an Azure Cosmos account with IP Firewall enabled.

# Azure CLI samples for Azure Cosmos DB Table API

9/25/2019 • 2 minutes to read • [Edit Online](#)

The following table includes links to sample Azure CLI scripts for Azure Cosmos DB Table API. Reference pages for all Azure Cosmos DB CLI commands are available in the [Azure CLI Reference](#). All Azure Cosmos DB CLI script samples can be found in the [Azure Cosmos DB CLI GitHub Repository](#).

<a href="#">Create an Azure Cosmos account and table</a>	Creates an Azure Cosmos DB account and table for Table API.
<a href="#">Change throughput</a>	Update RU/s on a table.
<a href="#">Add or failover regions</a>	Add a region, change failover priority, trigger a manual failover.
<a href="#">Account keys and connection strings</a>	List account keys, read-only keys, regenerate keys and list connection strings.
<a href="#">Secure with IP firewall</a>	Create a Cosmos account with IP firewall configured.
<a href="#">Secure new account with service endpoints</a>	Create a Cosmos account and secure with service-endpoints.
<a href="#">Secure existing account with service endpoints</a>	Update a Cosmos account to secure with service-endpoints when the subnet is eventually configured.

# Azure Cosmos DB Table .NET API: Download and release notes

1/26/2020 • 4 minutes to read • [Edit Online](#)

<b>SDK download</b>	<a href="#">NuGet</a>
<b>Quickstart</b>	<a href="#">Azure Cosmos DB: Build an app with .NET and the Table API</a>
<b>Tutorial</b>	<a href="#">Azure Cosmos DB: Develop with the Table API in .NET</a>
<b>Current supported framework</b>	<a href="#">Microsoft .NET Framework 4.5.1</a>

## 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.

Any requests to Azure Cosmos DB using a retired SDK are rejected by the service.

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

### **1. How will customers be notified of the retiring SDK?**

Microsoft will provide 12 month advance notification to the end of support of the retiring SDK in order to facilitate a smooth transition to a supported SDK. Further, customers will be notified through various communication channels – Azure Management Portal, Developer Center, blog post, and direct communication to assigned service administrators.

### **2. Can customers author applications using a "to-be" retired Azure Cosmos DB SDK during the 12 month period?**

Yes, customers will have full access to author, deploy and modify applications using the "to-be" retired Azure Cosmos DB SDK during the 12 month grace period. During the 12 month grace period, customers are advised to migrate to a newer supported version of Azure Cosmos DB SDK as appropriate.

### **3. Can customers author and modify applications using a retired Azure Cosmos DB SDK after the 12 month notification period?**

After the 12 month notification period, the SDK will be retired. Any access to Azure Cosmos DB by an applications using a retired SDK will not be permitted by the Azure Cosmos DB platform. Further, Microsoft will not provide customer support on the retired SDK.

### **4. What happens to Customer's running applications that are using unsupported Azure Cosmos DB SDK version?**

Any attempts made to connect to the Azure Cosmos DB service with a retired SDK version will be rejected.

### **5. Will new features and functionality be applied to all non-retired SDKs?**

New features and functionality will only be added to new versions. If you are using an old, non-retired, version of the SDK your requests to Azure Cosmos DB will still function as previous but you will not have access to any new capabilities.

### **6. What should I do if I cannot update my application before a cut-off date?**

We recommend that you upgrade to the latest SDK as early as possible. Once an SDK has been tagged for retirement you will have 12 months to update your application. If, for whatever reason, you cannot complete your application update within this timeframe then please contact the [Cosmos DB Team](#) and request their assistance before the cutoff date.

## 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

1/26/2020 • 4 minutes to read • [Edit Online](#)

<b>SDK download</b>	<a href="#">NuGet</a>
<b>Sample</b>	<a href="#">Cosmos DB Table API .NET Sample</a>
<b>Quickstart</b>	<a href="#">Quickstart</a>
<b>Tutorial</b>	<a href="#">Tutorial</a>
<b>Current supported framework</b>	<a href="#">Microsoft .NET Standard 2.0</a>
<b>Report Issue</b>	<a href="#">Report Issue</a>

## 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](#).

### **1.0.5**

- 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

### **1. How will customers be notified of the retiring SDK?**

Microsoft will provide 12 month advance notification to the end of support of the retiring SDK in order to facilitate a smooth transition to a supported SDK. Further, customers will be notified through various communication channels – Azure Management Portal, Developer Center, blog post, and direct communication to assigned service administrators.

### **2. Can customers author applications using a "to-be" retired Azure Cosmos DB SDK during the 12 month period?**

Yes, customers will have full access to author, deploy and modify applications using the "to-be" retired Azure Cosmos DB SDK during the 12 month grace period. During the 12 month grace period, customers are advised to migrate to a newer supported version of Azure Cosmos DB SDK as appropriate.

### **3. Can customers author and modify applications using a retired Azure Cosmos DB SDK after the 12 month notification period?**

After the 12 month notification period, the SDK will be retired. Any access to Azure Cosmos DB by an applications using a retired SDK will not be permitted by the Azure Cosmos DB platform. Further, Microsoft will not provide customer support on the retired SDK.

### **4. What happens to Customer's running applications that are using unsupported Azure Cosmos DB SDK version?**

Any attempts made to connect to the Azure Cosmos DB service with a retired SDK version will be rejected.

### **5. Will new features and functionality be applied to all non-retired SDKs?**

New features and functionality will only be added to new versions. If you are using an old, non-retired, version of the SDK your requests to Azure Cosmos DB will still function as previous but you will not have access to any new

capabilities.

## **6. What should I do if I cannot update my application before a cut-off date?**

We recommend that you upgrade to the latest SDK as early as possible. Once an SDK has been tagged for retirement you will have 12 months to update your application. If, for whatever reason, you cannot complete your application update within this timeframe then please contact the [Cosmos DB Team](#) and request their assistance before the cutoff date.

## 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

1/26/2020 • 2 minutes to read • [Edit Online](#)

<b>SDK download</b>	<a href="#">Download Options</a>
<b>API documentation</b>	<a href="#">Java API reference documentation</a>
<b>Contribute to SDK</b>	<a href="#">GitHub</a>

## 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

### 1. How will customers be notified of the retiring SDK?

Microsoft will provide 12 month advance notification to the end of support of the retiring SDK in order to facilitate a smooth transition to a supported SDK. Further, customers will be notified through various communication channels – Azure Management Portal, Developer Center, blog post, and direct communication to assigned service administrators.

### 2. Can customers author applications using a "to-be" retired Azure Cosmos DB SDK during the 12 month period?

Yes, customers will have full access to author, deploy and modify applications using the "to-be" retired Azure Cosmos DB SDK during the 12 month grace period. During the 12 month grace period, customers are advised to migrate to a newer supported version of Azure Cosmos DB SDK as appropriate.

### **3. Can customers author and modify applications using a retired Azure Cosmos DB SDK after the 12 month notification period?**

After the 12 month notification period, the SDK will be retired. Any access to Azure Cosmos DB by an applications using a retired SDK will not be permitted by the Azure Cosmos DB platform. Further, Microsoft will not provide customer support on the retired SDK.

### **4. What happens to Customer's running applications that are using unsupported Azure Cosmos DB SDK version?**

Any attempts made to connect to the Azure Cosmos DB service with a retired SDK version will be rejected.

### **5. Will new features and functionality be applied to all non-retired SDKs?**

New features and functionality will only be added to new versions. If you are using an old, non-retired, version of the SDK your requests to Azure Cosmos DB will still function as previous but you will not have access to any new capabilities.

### **6. What should I do if I cannot update my application before a cut-off date?**

We recommend that you upgrade to the latest SDK as early as possible. Once an SDK has been tagged for retirement you will have 12 months to update your application. If, for whatever reason, you cannot complete your application update within this timeframe then please contact the [Cosmos DB Team](#) and request their assistance before the cutoff date.

## 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

1/26/2020 • 2 minutes to read • [Edit Online](#)

<b>SDK download</b>	<a href="#">NPM</a>
<b>API documentation</b>	<a href="#">Node.js API reference documentation</a>
<b>Contribute to SDK</b>	<a href="#">GitHub</a>

## 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

### 1. How will customers be notified of the retiring SDK?

Microsoft will provide 12 month advance notification to the end of support of the retiring SDK in order to facilitate a smooth transition to a supported SDK. Further, customers will be notified through various communication channels – Azure Management Portal, Developer Center, blog post, and direct communication to assigned service administrators.

### 2. Can customers author applications using a "to-be" retired Azure Cosmos DB SDK during the 12 month period?

Yes, customers will have full access to author, deploy and modify applications using the "to-be" retired Azure Cosmos DB SDK during the 12 month grace period. During the 12 month grace period, customers are advised to migrate to a newer supported version of Azure Cosmos DB SDK as appropriate.

### **3. Can customers author and modify applications using a retired Azure Cosmos DB SDK after the 12 month notification period?**

After the 12 month notification period, the SDK will be retired. Any access to Azure Cosmos DB by an applications using a retired SDK will not be permitted by the Azure Cosmos DB platform. Further, Microsoft will not provide customer support on the retired SDK.

### **4. What happens to Customer's running applications that are using unsupported Azure Cosmos DB SDK version?**

Any attempts made to connect to the Azure Cosmos DB service with a retired SDK version will be rejected.

### **5. Will new features and functionality be applied to all non-retired SDKs?**

New features and functionality will only be added to new versions. If you are using an old, non-retired, version of the SDK your requests to Azure Cosmos DB will still function as previous but you will not have access to any new capabilities.

### **6. What should I do if I cannot update my application before a cut-off date?**

We recommend that you upgrade to the latest SDK as early as possible. Once an SDK has been tagged for retirement you will have 12 months to update your application. If, for whatever reason, you cannot complete your application update within this timeframe then please contact the [Cosmos DB Team](#) and request their assistance before the cutoff date.

## 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

1/26/2020 • 2 minutes to read • [Edit Online](#)

<b>SDK download</b>	<a href="#">PyPI</a>
<b>API documentation</b>	<a href="#">Python API reference documentation</a>
<b>SDK installation instructions</b>	<a href="#">Python SDK installation instructions</a>
<b>Contribute to SDK</b>	<a href="#">GitHub</a>
<b>Current supported platform</b>	<a href="#">Python 2.7</a> or <a href="#">Python 3.3, 3.4, 3.5, or 3.6</a>

## 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

### 1. How will customers be notified of the retiring SDK?

Microsoft will provide 12 month advance notification to the end of support of the retiring SDK in order to facilitate a smooth transition to a supported SDK. Further, customers will be notified through various communication channels – Azure Management Portal, Developer Center, blog post, and direct communication to assigned service administrators.

**2. Can customers author applications using a "to-be" retired Azure Cosmos DB SDK during the 12 month period?**

Yes, customers will have full access to author, deploy and modify applications using the "to-be" retired Azure Cosmos DB SDK during the 12 month grace period. During the 12 month grace period, customers are advised to migrate to a newer supported version of Azure Cosmos DB SDK as appropriate.

**3. Can customers author and modify applications using a retired Azure Cosmos DB SDK after the 12 month notification period?**

After the 12 month notification period, the SDK will be retired. Any access to Azure Cosmos DB by an applications using a retired SDK will not be permitted by the Azure Cosmos DB platform. Further, Microsoft will not provide customer support on the retired SDK.

**4. What happens to Customer's running applications that are using unsupported Azure Cosmos DB SDK version?**

Any attempts made to connect to the Azure Cosmos DB service with a retired SDK version will be rejected.

**5. Will new features and functionality be applied to all non-retired SDKs?**

New features and functionality will only be added to new versions. If you are using an old, non-retired, version of the SDK your requests to Azure Cosmos DB will still function as previous but you will not have access to any new capabilities.

**6. What should I do if I cannot update my application before a cut-off date?**

We recommend that you upgrade to the latest SDK as early as possible. Once an SDK has been tagged for retirement you will have 12 months to update your application. If, for whatever reason, you cannot complete your application update within this timeframe then please contact the [Cosmos DB Team](#) and request their assistance before the cutoff date.

## See also

To learn more about Cosmos DB, see [Microsoft Azure Cosmos DB service page](#).

# Introduction to the Azure Cosmos DB etcd API (preview)

5/6/2019 • 2 minutes to read • [Edit Online](#)

Azure Cosmos DB is Microsoft's globally distributed, multi-model database service for mission-critical applications. It offers turnkey global distribution, elastic scaling of throughput and storage, single-digit millisecond latencies at the 99th percentile, and guaranteed high availability, all backed by industry-leading SLA's.

**Etcd** is a distributed key/value store. In [Kubernetes](#), etcd is used to store the state and the configuration of the Kubernetes clusters. Ensuring availability, reliability, and performance of etcd is crucial to the overall cluster health, scalability, elasticity availability, and performance of a Kubernetes cluster.

The etcd API in Azure Cosmos DB allows you to use Azure Cosmos DB as the backend store for [Azure Kubernetes](#). etcd API in Azure Cosmos DB is currently in preview. Azure Cosmos DB implements the etcd wire protocol. With etcd API in Azure Cosmos DB, developers will automatically get highly reliable, [available, globally distributed](#) Kubernetes. This API allows developers to scale Kubernetes state management on a fully managed cloud native PaaS service.

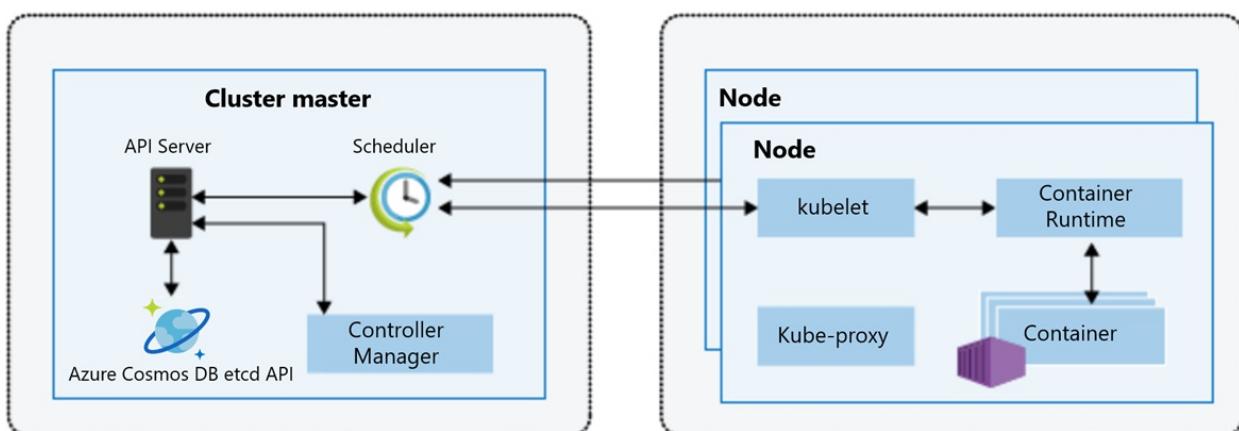
## NOTE

Unlike other APIs in Azure Cosmos DB, you cannot provision an etcd API account through the Azure portal, CLI or SDKs. You can provision an etcd API account by deploying the Resource Manager template only; for detailed steps, see [How to provision Azure Kubernetes with Azure Cosmos DB](#) article. Azure Cosmos DB etcd API is currently in limited preview. You can [sign-up for the preview](#), by filling out the sign-up form.

## Wire level compatibility

Azure Cosmos DB implements the wire-protocol of etcd version 3, and allows the [master node's](#) API servers to use Azure Cosmos DB just like it would do in a locally installed etcd environment. The etcd API supports TLS mutual authentication.

The following diagram shows the components of a Kubernetes cluster. In the cluster master, the API Server uses Azure Cosmos DB etcd API, instead of locally installed etcd.



## Key benefits

## No etcd operations management

As a fully managed native cloud service, Azure Cosmos DB removes the need for Kubernetes developers to set up and manage etcd. The etcd API in Azure Cosmos DB is scalable, highly available, fault tolerant, and offers high performance. The overhead of setting up replication across multiple nodes, performing rolling updates, security patches, and monitoring the etcd health are handled by Azure Cosmos DB.

## Global distribution & high availability

By using etcd API, Azure Cosmos DB guarantees 99.99% availability for data reads and writes in a single region, and 99.999% availability across multiple regions.

## Elastic scalability

Azure Cosmos DB offers elastic scalability for read and write requests across different regions. As the Kubernetes cluster grows, the etcd API account in Azure Cosmos DB elastically scales without any downtime. Storing etcd data in Azure Cosmos DB, instead of the Kubernetes master nodes also enables more flexible master node scaling.

## Security & enterprise readiness

When etcd data is stored in Azure Cosmos DB, Kubernetes developers automatically get the [built-in encryption at rest, certifications and compliance](#), and [backup and restore capabilities](#) supported by Azure Cosmos DB.

## Next steps

- [How to use Azure Kubernetes with Azure Cosmos DB](#)
- [Key benefits of Azure Cosmos DB](#)
- [AKS engine Quickstart guide](#)

# How to use Azure Kubernetes with Azure Cosmos DB (preview)

8/28/2019 • 5 minutes to read • [Edit Online](#)

The etcd API in Azure Cosmos DB allows you to use Azure Cosmos DB as the backend store for Azure Kubernetes. Azure Cosmos DB implements the etcd wire protocol, which allows the master node's API servers to use Azure Cosmos DB just like it would access a locally installed etcd. etcd API in Azure Cosmos DB is currently in preview. When you use Azure Cosmos etcd API as the backing store for Kubernetes, you get the following benefits:

- No need to manually configure and manage etcd.
- High availability of etcd, guaranteed by Cosmos (99.99% in single region, 99.999% in multiple regions).
- Elastic scalability of etcd.
- Secure by default & enterprise ready.
- Industry-leading, comprehensive SLAs.

To learn more about etcd API in Azure Cosmos DB, see the [overview](#) article. This article shows you how to use [Azure Kubernetes Engine](#) (aks-engine) to bootstrap a Kubernetes cluster on Azure that uses [Azure Cosmos DB](#) instead of a locally installed and configured etcd.

## Prerequisites

1. Install the latest version of [Azure CLI](#). You can download Azure CLI specific to your operating system and install.
2. Install the [latest version](#) of Azure Kubernetes Engine. The installation instructions for different operating systems are available in [Azure Kubernetes Engine](#) page. You just need the steps from **Install AKS Engine** section of the linked doc. After downloading, extract the zip file.

The Azure Kubernetes Engine (**aks-engine**) generates Azure Resource Manager templates for Kubernetes clusters on Azure. The input to aks-engine is a cluster definition file that describes the desired cluster, including orchestrator, features, and agents. The structure of the input files is similar to the public API for Azure Kubernetes Service.

3. The etcd API in Azure Cosmos DB is currently in preview. Sign up to use the preview version at: <https://aka.ms/cosmosetcdapi-signup>. After you submit the form, your subscription will be whitelisted to use the Azure Cosmos etcd API.

## Deploy the cluster with Azure Cosmos DB

1. Open a command prompt window, and sign into Azure with the following command:

```
az login
```

2. If you have more than one subscription, switch to the subscription that has been whitelisted for Azure Cosmos DB etcd API. You can switch to the required subscription using the following command:

```
az account set --subscription "<Name of your subscription>"
```

3. Next create a new resource group where you will deploy the resources required by the Azure Kubernetes cluster. Make sure to create the resource group in the "centralus" region. It's not mandatory for the resource group to be in "centralus" region however, Azure Cosmos etcd API is currently available to deploy in "centralus" region only. So it's best to have the Kubernetes cluster to be colocated with the Cosmos etcd instance:

```
az group create --name <Name> --location "centralus"
```

4. Next create a service principal for the Azure Kubernetes cluster so that it can communicate with the resources that are a part of the same resource group. You can create a service principal using Azure CLI, PowerShell or Azure portal, in this example you will use CLI to create it.

```
az ad sp create-for-rbac --role="Contributor" --  
scopes="/subscriptions/<Your_Azure_subscription_ID>/resourceGroups/<Your_resource_group_name>"
```

This command outputs the details of a service principal, for example:

```
Retrying role assignment creation: 1/36  
{  
  "appId": "8415a4e9-4f83-46ca-a704-107457b2e3ab",  
  "displayName": "azure-cli-2019-04-19-19-01-46",  
  "name": "http://azure-cli-2019-04-19-19-01-46",  
  "password": "102aec3-5e37-4f3d-8738-2ac348c2e6a7",  
  "tenant": "72f988bf-86f1-41af-91ab-2d7cd011db47"  
}
```

Make a note of the **appId** and the **password** fields, as you will use these parameters in the next steps.

5. From the command prompt, navigate to the folder where the Azure Kubernetes Engine executable is located. For example, on your command prompt you can navigate to the folder as:

```
cd "\aks-engine-v0.36.3-windows-amd64\aks-engine-v0.36.3-windows-amd64"
```

6. Open a text editor of your choice and define a Resource Manager template that deploys the Azure Kubernetes cluster with Azure Cosmos DB etcd API. Copy the following JSON definition to your text editor and save the file as `apiModel.json`:

```
{
  "apiVersion": "vlabs",
  "properties": {
    "orchestratorProfile": {
      "orchestratorType": "Kubernetes",
      "kubernetesConfig": {
        "useManagedIdentity": false
      }
    },
    "masterProfile": {
      "count": 1,
      "dnsPrefix": "",
      "vmSize": "Standard_D2_v3",
      "cosmosEtcd": true
    },
    "agentPoolProfiles": [
      {
        "name": "agent",
        "count": 1,
        "vmSize": "Standard_D2_v3",
        "availabilityProfile": "AvailabilitySet"
      }
    ],
    "linuxProfile": {
      "adminUsername": "azureuser",
      "ssh": {
        "publicKeys": [
          {
            "keyData": ""
          }
        ]
      }
    }
  }
}
```

In the JSON/cluster definition file, the key parameter to note is "**cosmosEtcd**: true". This parameter is in the "masterProfile" properties and it indicates the deployment to use Azure Cosmos etcd API instead of regular etcd.

## 7. Deploy the Azure Kubernetes cluster that uses Azure Cosmos DB with the following command:

```
aks-engine deploy \
--subscription-id <Your_Azure_subscription_ID> \
--client-id <Service_principal_appId> \
--client-secret <Service_principal_password> \
--dns-prefix <Region_unique_dns_name> \
--location centralus \
--resource-group <Resource_Group_Name> \
--api-model <Fully_qualified_path_to_the_template_file> \
--force-overwrite
```

Azure Kubernetes Engine consumes a cluster definition that outlines the desired shape, size, and configuration of the Azure Kubernetes. There are several features that can be enabled through the cluster definition. In this example you will use the following parameters:

- **subscription-id**: Azure subscription ID that has Azure Cosmos DB etcd API enabled.
- **client-id**: The service principal's appId. The `appId` was returned as output in step 4.
- **Client-secret**: The service principal's password or a randomly generated password. This value was returned as output in the 'password' parameter in step 4.

- **dnsPrefix:** A region-unique DNS name. This value will form part of the hostname (example values are-myprod1, staging).
- **location:** Location where the cluster should be deployed to, currently only "centralus" is supported.

**NOTE**

Azure Cosmos etcd API is currently available to deploy in "centralus" region only.

- **api-model:** Fully qualified path to the template file.
- **force-overwrite:** This option is used to automatically overwrite existing files in the output directory.

The following command shows an example deployment:

```
aks-engine deploy \
--subscription-id 1234fc61-1234-1234-1234-be1234d12c1b \
--client-id 1234a4e9-4f83-46ca-a704-107457b2e3ab \
--client-secret 123aecd3-5e37-4f3d-8738-2ac348c2e6a7 \
--dns-prefix aks-sg-test \
--location centralus \
--api-model "C:\Users\demouser\Downloads\apiModel.json" \
--force-overwrite
```

## Verify the deployment

The template deployment takes several minutes to complete. After the deployment is successfully completed, you will see the following output in the commands prompt:

```
WARN[0006] apimodel: missing masterProfile.dnsPrefix will use "aks-sg-test"
WARN[0006] --resource-group was not specified. Using the DNS prefix from the apimodel as the resource group
name: aks-sg-test
INFO[0025] Starting ARM Deployment (aks-sg-test-546247491). This will take some time...
INFO[0587] Finished ARM Deployment (aks-sg-test-546247491). Succeeded
```

The resource group now contains resources such as- virtual machine, Azure Cosmos account(etcd API), virtual network, availability set, and other resources required by the Kubernetes cluster.

The Azure Cosmos account's name will match your specified DNS prefix appended with k8s. Your Azure Cosmos account will be automatically provisioned with a database named **EtcDB** and a container named **EtcData**. The container will store all the etcd related data. The container is provisioned with a certain number of request units and you can [scale \(increase/decrease\) the throughput](#) based on your workload.

## Next steps

- Learn how to [work with Azure Cosmos database, containers, and items](#)
- Learn how to [optimize provisioned throughput costs](#)

# Manage an Azure Cosmos account

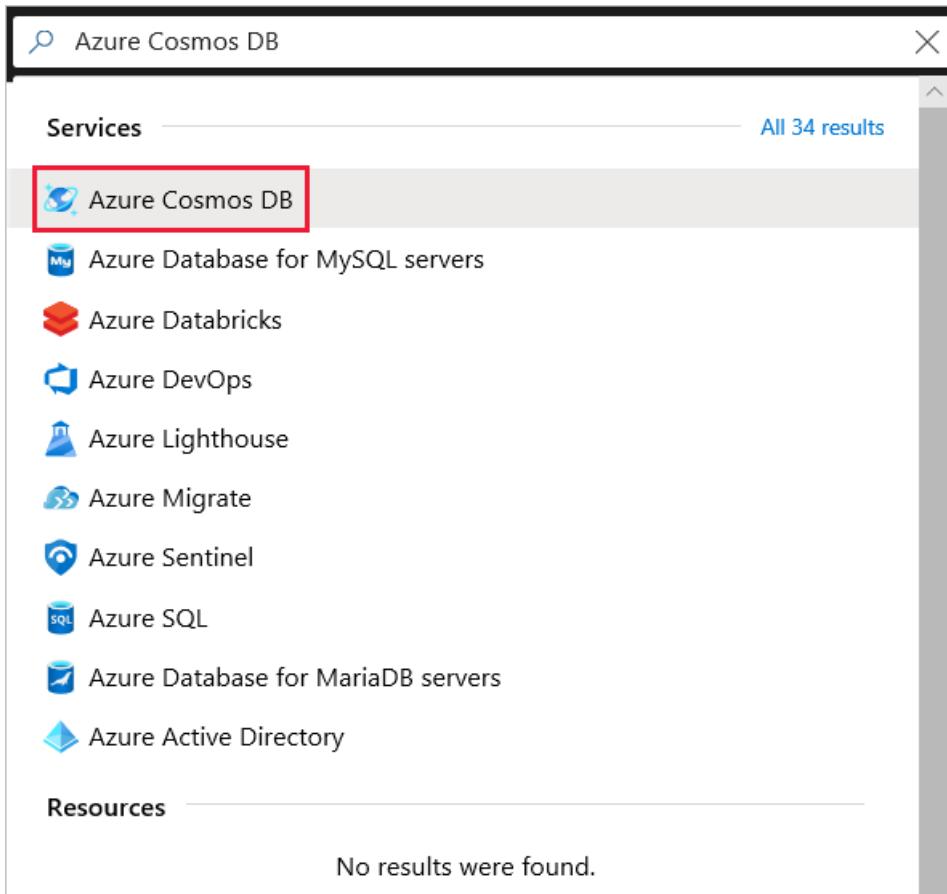
12/5/2019 • 6 minutes to read • [Edit Online](#)

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. Go to the [Azure portal](#) to create an Azure Cosmos DB account. Search for and select **Azure Cosmos DB**.



The screenshot shows the Azure portal's search interface. A search bar at the top contains the text "Azure Cosmos DB". Below the search bar, a list of services is displayed under the heading "Services". The "Azure Cosmos DB" item is highlighted with a red box. Other listed services include Azure Database for MySQL servers, Azure Databricks, Azure DevOps, Azure Lighthouse, Azure Migrate, Azure Sentinel, Azure SQL, Azure Database for MariaDB servers, and Azure Active Directory. A separate section titled "Resources" shows the message "No results were found."

2. Select **Add**.
3. 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 <b>Create new</b> , then enter a unique name for the new resource group.

Setting	Value	Description
Account Name	A unique name	<p>Enter a name to identify your Azure Cosmos account. Because <i>documents.azure.com</i> is appended to the name that you provide to create your URI, use a unique name.</p> <p>The name can only contain lowercase letters, numbers, and the hyphen (-) character. It must be between 3-31 characters in length.</p>
API	The type of account to create	<p>Select <b>Core (SQL)</b> to create a document database and query by using SQL syntax.</p> <p>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.</p> <p><a href="#">Learn more about the SQL API.</a></p>
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.

## Create Azure Cosmos DB Account

[Basics](#) [Network](#) [Tags](#) [Review + create](#)

Azure Cosmos DB is a fully managed globally distributed, multi-model database service, transparently replicating your data across any number of Azure regions. You can elastically scale throughput and storage, and take advantage of fast, single-digit-millisecond data access using the API of your choice backed by 99.999 SLA. [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.

<p>* Subscription</p> <p>Resource Group</p>	<div style="border: 1px solid #ccc; padding: 2px;">Contoso Subscription</div> <div style="border: 1px solid #ccc; padding: 2px;">(New) myResourceGroup</div> <a href="#">Create new</a>
<p><b>INSTANCE DETAILS</b></p> <p>* Account Name</p> <p>mysqlapicosmosdb <span style="color: green;">✓</span> documents.azure.com</p> <p>* API <span style="color: blue;">i</span></p> <p>Core (SQL)</p> <p>* Location</p> <p>West US</p> <p>Geo-Redundancy <span style="color: blue;">i</span></p> <p><span style="background-color: #0078d4; color: white; padding: 2px 10px;">Enable</span> <span style="border: 1px solid #ccc; padding: 2px;">Disable</span></p> <p>Multi-region Writes <span style="color: blue;">i</span></p> <p><span style="background-color: #0078d4; color: white; padding: 2px 10px;">Enable</span> <span style="border: 1px solid #ccc; padding: 2px;">Disable</span></p>	

[Review + create](#)

[Previous](#)

[Next: Network](#)

4. Select **Review + create**. You can skip the **Network** and **Tags** sections.

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**.

Dashboard > Microsoft.Azure.CosmosDB-20190321000000 - Overview

**Microsoft.Azure.CosmosDB-20190321000000 - Overview**

Deployment

Search (Ctrl+/  
)

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	<a href="#">Operation details</a>

6. Select **Go to resource** to go to the Azure Cosmos DB account page.

The screenshot shows the Azure portal interface for a Cosmos DB account named 'mysqlapicosmosdb'. The left sidebar contains navigation links: Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Quick start (which is selected and highlighted in blue), Notifications, Data Explorer, Settings, Replicate data globally, Default consistency, and Firewall and virtual networks. The main content area starts with a message: 'Congratulations! Your Azure Cosmos DB account was created.' It then instructs the user to connect to it using a sample app. A section titled 'Choose a platform' offers links for .NET, .NET Core, Xamarin, Java, Node.js, and Python. Below this, two numbered steps are listed: '1 Add a collection' (described as creating an 'Items' collection with 10GB storage capacity and 400 Request Units per second (RU/s) throughput capacity) and '2 Download and run your .NET app' (described as downloading a sample .NET app connected to the collection). A 'Create 'Items' collection' button is located between the two steps.

## 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

This Azure Resource Manager template will create an Azure Cosmos account for SQL API configured with two regions and options to select consistency level, automatic failover, and multi-master. To deploy this template, click on Deploy to Azure on the readme page, [Create Azure Cosmos account](#)

## 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**.

Click on a location to add or remove regions from your Azure Cosmos DB account.  
\* Each region is billable based on the throughput and storage for the account. [Learn more](#)

**Configure regions**

Configure the regions available for reads and writes.

**WRITE REGION**

West US

**READ REGIONS**

East US  
Southeast Asia  
North Europe  
Central India  
Japan West  
Brazil South  
UK West  
East US 2

Search for a region

OK Cancel

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.

cdbsm1 - Replicate data globally

Save Discard Manual Failover Automatic Failover

Click on a location to add or remove regions from your Azure Cosmos DB account.  
\* Each region is billable based on the throughput and storage for the account. [Learn more](#)

**Configure regions**

Enable multi-region writes [?](#)

[Disable](#) [Enable](#)

Configure the regions available for reads and writes. + Add region

REGIONS	READS ENABLED	WRITES ENABLED
Australia East	✓	✓
East US 2	✓	✓

## 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-master to multi-master 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

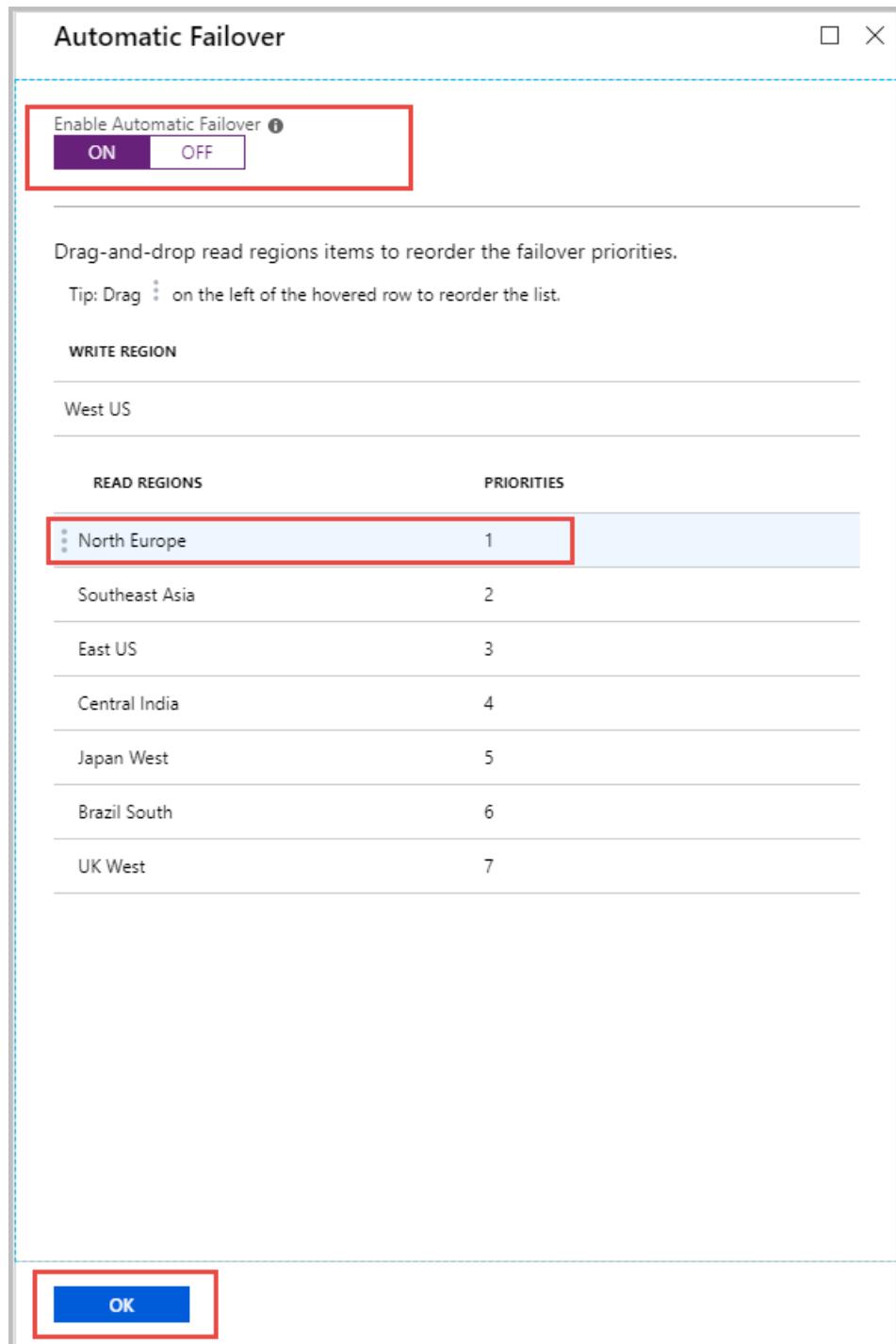
1. From your Azure Cosmos account, open the **Replicate data globally** pane.

2. At the top of the pane, select **Automatic Failover**.

The screenshot shows the Azure portal interface for managing an Azure Cosmos DB account. The left sidebar contains navigation links like Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Quick start, Data Explorer, Notifications, Settings, Replicate data globally (which is highlighted in blue), Default consistency, Firewall and virtual networks, Keys, and Add Azure Function. The main pane has tabs for Save, Discard, Manual Failover, and Automatic Failover (which is selected). A world map in the center allows users to click on locations to add or remove regions. Below the map, a note says: "Click on a location to add or remove regions from your Azure Cosmos DB account. \* Each region is billable based on the throughput and storage for the account. Learn more". To the right, there are sections for "Configure regions", "WRITE REGION" (listing West US), and "READ REGIONS" (listing East US, Southeast Asia, North Europe, Central India, Japan West, Brazil South, and UK West). There are also "Add region" buttons for both sections.

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**.

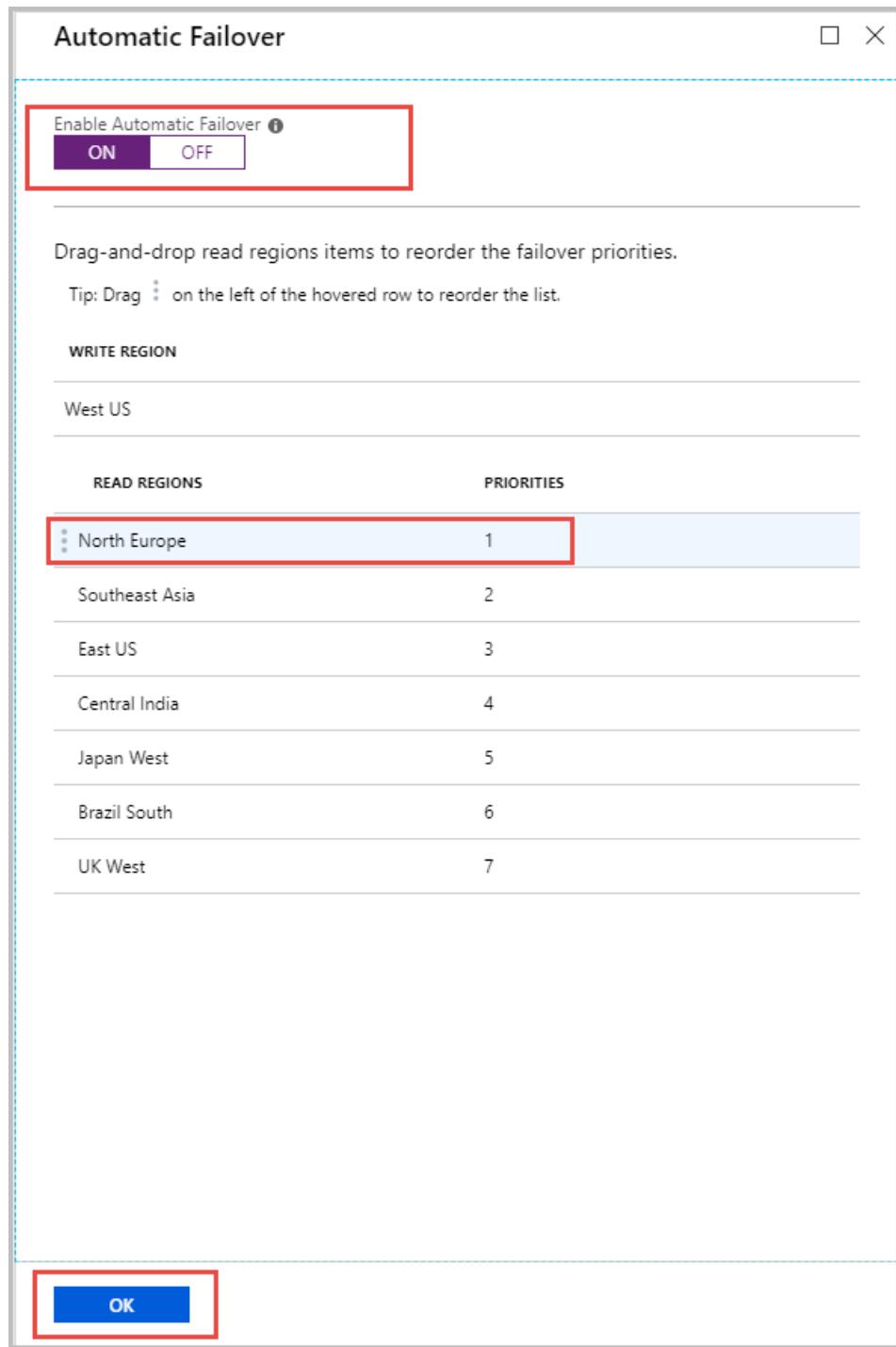
The screenshot shows the 'Replicate data globally' pane in the Azure Cosmos DB settings. The 'Automatic Failover' tab is selected at the top. A world map in the center displays various regions marked with blue dots. To the right, there are sections for 'Configure regions', 'WRITE REGION', and 'READ REGIONS', each listing specific Azure regions with checkboxes. The 'Settings' sidebar on the left has 'Replicate data globally' selected.

Region Type	Regions
WRITE REGION	West US, Southeast Asia, North Europe, Central India, Japan West, Brazil South, UK West
READ REGIONS	East US, Southeast Asia, North Europe, Central India, Japan West, Brazil South, UK West

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

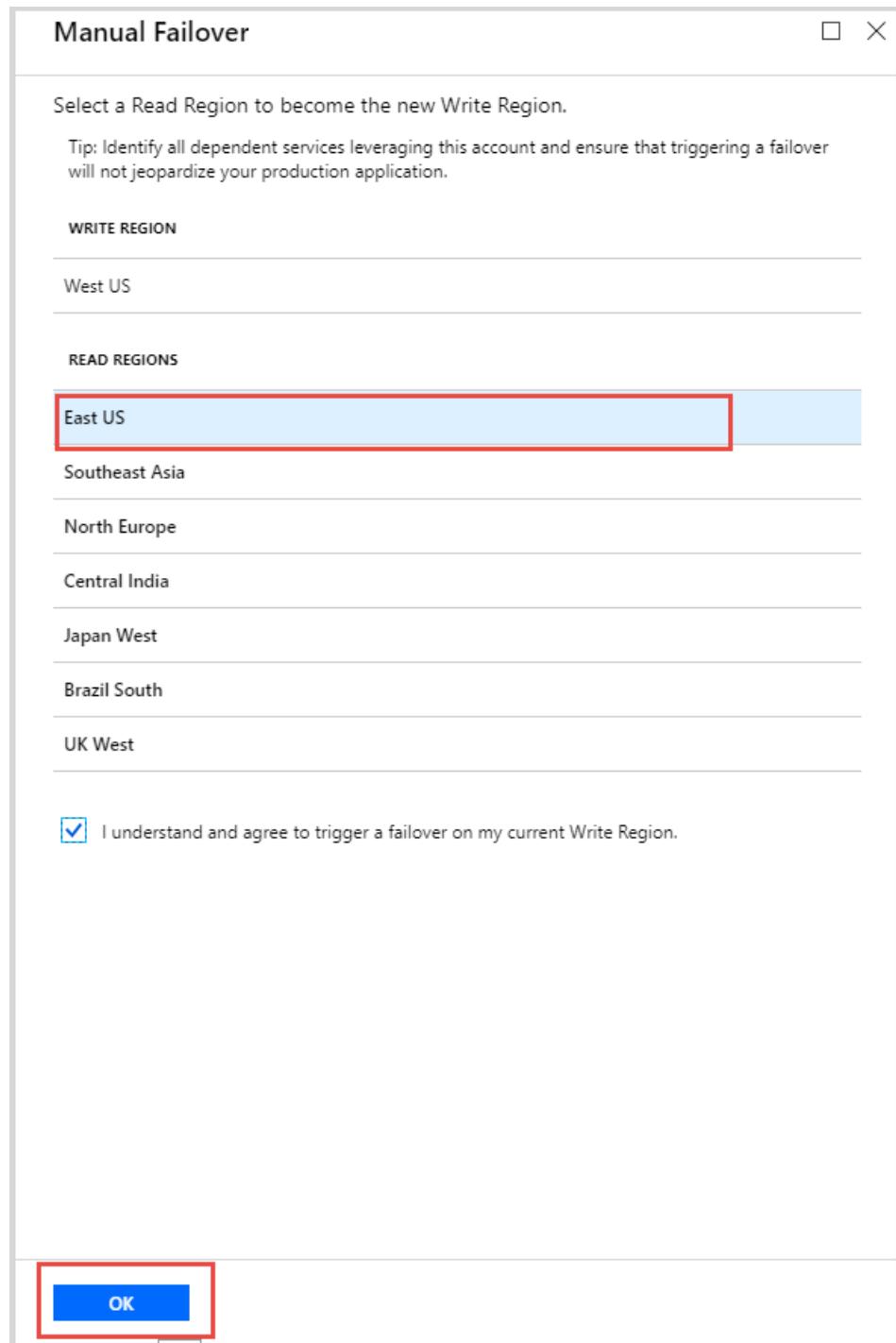
Multi-master accounts cannot be manually failed over. For applications using the Azure Cosmos SDK, the SDK will detect when a region becomes unavailable, then redirect automatically to the next closest region if using multi-homing API in the SDK.

## 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 SQL API resources using PowerShell

10/24/2019 • 14 minutes to read • [Edit Online](#)

The following guide describes how to use PowerShell to script and automate management of Azure Cosmos DB resources, including account, database, container, and throughput. Management of Azure Cosmos DB is handled through the `AzResource` cmdlet directly to the Azure Cosmos DB resource provider. To view all of the properties that can be managed using PowerShell for the Azure Cosmos DB resource provider, see [Azure Cosmos DB resource provider schema](#)

For cross-platform management of Azure Cosmos DB, you can use [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.

- If you would like to execute the following commands without requiring user confirmation, append the `-Force` flag to the command.
- All the following commands are synchronous.

## 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](#)

### Create an Azure Cosmos account

This command creates an Azure Cosmos database account with [multiple-regions](#), bounded-staleness [consistency policy](#).

```

# Create an Azure Cosmos Account for Core (SQL) API
$resourceGroupName = "myResourceGroup"
$location = "West US 2"
$accountName = "mycosmosaccount" # must be lowercase and < 31 characters .

$locations = @(
    @{ "locationName"="West US 2"; "failoverPriority"=0 },
    @{ "locationName"="East US 2"; "failoverPriority"=1 }
)

$consistencyPolicy = @{
    "defaultConsistencyLevel"="BoundedStaleness";
    "maxIntervalInSeconds"=300;
    "maxStalenessPrefix"=100000
}

$CosmosDBProperties = @{
    "databaseAccountOfferType"="Standard";
    "locations"=$locations;
    "consistencyPolicy"=$consistencyPolicy;
    "enableMultipleWriteLocations"="false"
}

New-AzResource -ResourceType "Microsoft.DocumentDb/databaseAccounts" ` 
    -ApiVersion "2015-04-08" -ResourceGroupName $resourceGroupName -Location $location ` 
    -Name $accountName -PropertyObject $CosmosDBProperties

```

- `$accountName` The name for the Azure Cosmos account. Must be lowercase, accepts alphanumeric and the '-' character, and between 3 and 31 characters.
- `$location` The location for the Azure Cosmos account resource.
- `$locations` The replica regions for the database account. There must be one write region per database account with a failover priority value of 0.
- `$consistencyPolicy` The default consistency level of the Azure Cosmos account. For more information, see [Consistency Levels in Azure Cosmos DB](#).
- `$CosmosDBProperties` The property values passed to the Cosmos DB Azure Resource Manager Provider to provision the account.

Azure Cosmos accounts can be configured with IP Firewall as well as Virtual Network service end points. For information on how to configure the IP Firewall for Azure Cosmos DB, see [Configure IP Firewall](#). For more information on how to enable service endpoints for Azure Cosmos DB, see [Configure access from virtual Networks](#).

### List all Azure Cosmos accounts in a subscription

This command allows you to list all Azure Cosmos accounts in a subscription.

```

# List Azure Cosmos Accounts

Get-AzResource -ResourceType Microsoft.DocumentDb/databaseAccounts | ft

```

### Get the properties of an Azure Cosmos account

This command allows you to get the properties of an existing Azure Cosmos account.

```
# Get the properties of an Azure Cosmos Account

$resourceGroupName = "myResourceGroup"
$accountName = "mycosmosaccount"

Get-AzResource -ResourceType "Microsoft.DocumentDb/databaseAccounts" ` 
    -ApiVersion "2015-04-08" -ResourceGroupName $resourceGroupName ` 
    -Name $accountName | Select-Object Properties
```

## Update an Azure Cosmos account

This command allows you to update your Azure Cosmos 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-master

### 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 [Modify failover priority](#) and [Trigger manual failover](#).

```

# Create an account with 2 regions
$resourceGroupName = "myResourceGroup"
$resourceType = "Microsoft.DocumentDb/databaseAccounts"
$accountName = "mycosmosaccount" # must be lower case and < 31 characters

$locations = @(
    @{ "locationName"="West US 2"; "failoverPriority"=0, "isZoneRedundant"=$false },
    @{ "locationName"="East US 2"; "failoverPriority"=1, "isZoneRedundant"=$false }
)
$consistencyPolicy = @{ "defaultConsistencyLevel"="Session" }
$CosmosDBProperties = @{
    "databaseAccountOfferType"="Standard";
    "locations"=$locations;
    "consistencyPolicy"=$consistencyPolicy
}
New-AzResource -ResourceType $resourceType ` 
    -ApiVersion "2015-04-08" -ResourceGroupName $resourceGroupName ` 
    -Name $accountName -PropertyObject $CosmosDBProperties

# Add a region
$account = Get-AzResource -ResourceType $resourceType ` 
    -ApiVersion "2015-04-08" -ResourceGroupName $resourceGroupName ` 
    -Name $accountName

$locations = @(
    @{ "locationName"="West US 2"; "failoverPriority"=0, "isZoneRedundant"=$false },
    @{ "locationName"="East US 2"; "failoverPriority"=1, "isZoneRedundant"=$false },
    @{ "locationName"="South Central US"; "failoverPriority"=2, "isZoneRedundant"=$false }
)

$account.Properties.locations = $locations
$CosmosDBProperties = $account.Properties

Set-AzResource -ResourceType $resourceType ` 
    -ApiVersion "2015-04-08" -ResourceGroupName $resourceGroupName ` 
    -Name $accountName -PropertyObject $CosmosDBProperties

# Azure Resource Manager does not wait on the resource update
Write-Host "Confirm region added before continuing..."

# Remove a region
$account = Get-AzResource -ResourceType $resourceType ` 
    -ApiVersion "2015-04-08" -ResourceGroupName $resourceGroupName ` 
    -Name $accountName

$locations = @(
    @{ "locationName"="West US 2"; "failoverPriority"=0, "isZoneRedundant"=$false },
    @{ "locationName"="East US 2"; "failoverPriority"=1, "isZoneRedundant"=$false }
)

$account.Properties.locations = $locations
$CosmosDBProperties = $account.Properties

Set-AzResource -ResourceType $resourceType ` 
    -ApiVersion "2015-04-08" -ResourceGroupName $resourceGroupName ` 
    -Name $accountName -PropertyObject $CosmosDBProperties

```

## Enable multiple write regions for an Azure Cosmos account

```

# Update an Azure Cosmos account from single to multi-master
$resourceGroupName = "myResourceGroup"
$accountName = "mycosmosaccount"
$resourceType = "Microsoft.DocumentDb/databaseAccounts"

$account = Get-AzResource -ResourceType $resourceType ` 
    -ApiVersion "2015-04-08" -ResourceGroupName $resourceGroupName ` 
    -Name $accountName

$account.Properties.enableMultipleWriteLocations = "true"
$CosmosDBProperties = $account.Properties

Set-AzResource -ResourceType $resourceType ` 
    -ApiVersion "2015-04-08" -ResourceGroupName $resourceGroupName ` 
    -Name $accountName -PropertyObject $CosmosDBProperties

```

## Delete an Azure Cosmos account

This command allows you to delete an existing Azure Cosmos account.

```

# Delete an Azure Cosmos Account
$resourceGroupName = "myResourceGroup"
$accountName = "mycosmosaccount"

Remove-AzResource -ResourceType "Microsoft.DocumentDb/databaseAccounts" ` 
    -ApiVersion "2015-04-08" -ResourceGroupName $resourceGroupName ` 
    -Name $accountName

```

## Update Tags of an Azure Cosmos account

The following example describes how to set [Azure resource tags](#) for an Azure Cosmos account.

### NOTE

This command can be combined with the create or update commands by appending the `-Tags` flag with the corresponding parameter.

```

# Update tags for an Azure Cosmos Account

$resourceGroupName = "myResourceGroup"
$accountName = "mycosmosaccount"

$tags = @{
    "dept" = "Finance";
    "environment" = "Production"
}

Set-AzResource -ResourceType "Microsoft.DocumentDb/databaseAccounts" ` 
    -ApiVersion "2015-04-08" -ResourceGroupName $resourceGroupName ` 
    -Name $accountName -Tags $tags

```

## List Account Keys

When you create an Azure Cosmos DB account, the service generates two master access keys that can be used for authentication when the Azure Cosmos DB account is accessed. By providing two access keys, Azure Cosmos DB enables you to regenerate the keys with no interruption to your Azure Cosmos DB account. Read-only keys for authenticating read-only operations are also available. There are two read-write keys (primary and secondary) and two read-only keys (primary and secondary).

```
# List keys for an Azure Cosmos Account

$resourceGroupName = "myResourceGroup"
$accountName = "mycosmosaccount"

$keys = Invoke-AzResourceAction -Action listKeys ` 
    -ResourceType "Microsoft.DocumentDb/databaseAccounts" -ApiVersion "2015-04-08" ` 
    -ResourceGroupName $resourceGroupName -Name $accountName

Write-Host "PrimaryKey =" $keys.primaryMasterKey
Write-Host "SecondaryKey =" $keys.secondaryMasterKey
```

## List Connection Strings

For MongoDB accounts, the connection string to connect your MongoDB app to the database account can be retrieved using the following command.

```
# List connection strings for an Azure Cosmos Account

$resourceGroupName = "myResourceGroup"
$accountName = "mycosmosaccount"

$keys = Invoke-AzResourceAction -Action listConnectionStrings ` 
    -ResourceType "Microsoft.DocumentDb/databaseAccounts" -ApiVersion "2015-04-08" ` 
    -ResourceGroupName $resourceGroupName -Name $accountName

Select-Object $keys
```

## Regenerate Account Keys

Access keys to an Azure Cosmos account should be periodically regenerated to help keep connections more secure. A primary and secondary access keys are assigned to the account. This allows clients to maintain access while the other is regenerated. There are four types of keys for an Azure Cosmos account (Primary, Secondary, Primary Readonly, and Secondary Readonly)

```
# Regenerate the primary key for an Azure Cosmos Account

$resourceGroupName = "myResourceGroup"
$accountName = "mycosmosaccount"

$keyKind = @{ "keyKind"="Primary" }

$keys = Invoke-AzResourceAction -Action regenerateKey ` 
    -ResourceType "Microsoft.DocumentDb/databaseAccounts" -ApiVersion "2015-04-08" ` 
    -ResourceGroupName $resourceGroupName -Name $accountName -Parameters $keyKind

Select-Object $keys
```

## Enable automatic failover

Enables a Cosmos account to failover to its secondary region should the primary region become unavailable.

```

$resourceGroupName = "myResourceGroup"
$accountName = "mycosmosaccount"
$resourceType = "Microsoft.DocumentDb/databaseAccounts"

$account = Get-AzResource -ResourceType $resourceType ` 
    -ApiVersion "2015-04-08" -ResourceGroupName $resourceGroupName ` 
    -Name $accountName

$account.Properties.enableAutomaticFailover="true";
$CosmosDBProperties = $account.Properties;

Set-AzResource -ResourceType $resourceType ` 
    -ApiVersion "2015-04-08" -ResourceGroupName $resourceGroupName ` 
    -Name $accountName -PropertyObject $CosmosDBProperties

```

## 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, `West US 2 = 0`, `East US 2 = 1`, `South Central US = 2`.

**Caution**

Changing `locationName` for `failoverPriority=0` will trigger a manual failover for an Azure Cosmos account. Any other priority changes will not trigger a failover.

```

# Change the failover priority for an Azure Cosmos Account
# Assume existing priority is "West US 2" = 0, "East US 2" = 1, "South Central US" = 2

$resourceGroupName = "myResourceGroup"
$accountName = "mycosmosaccount"

$failoverRegions = @(
    @{ "locationName"="West US 2"; "failoverPriority"=0 },
    @{ "locationName"="South Central US"; "failoverPriority"=1 },
    @{ "locationName"="East US 2"; "failoverPriority"=2 }
)

$failoverPolicies = @{
    "failoverPolicies"= $failoverRegions
}

Invoke-AzResourceAction -Action failoverPriorityChange ` 
    -ResourceType "Microsoft.DocumentDb/databaseAccounts" -ApiVersion "2015-04-08" ` 
    -ResourceGroupName $resourceGroupName -Name $accountName -Parameters $failoverPolicies

```

## Trigger Manual Failover

For accounts configured with Manual Failover, you can failover 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.

```

# Change the failover priority for an Azure Cosmos Account
# Assume existing priority is "West US 2" = 0, "East US 2" = 1, "South Central US" = 2

$resourceGroupName = "myResourceGroup"
$accountName = "mycosmosaccount"

$failoverRegions = @(
    @{ "locationName"="South Central US"; "failoverPriority"=0 },
    @{ "locationName"="East US 2"; "failoverPriority"=1 },
    @{ "locationName"="West US 2"; "failoverPriority"=2 }
)

$failoverPolicies = @{
    "failoverPolicies"= $failoverRegions
}

Invoke-AzResourceAction -Action failoverPriorityChange ` 
    -ResourceType "Microsoft.DocumentDb/databaseAccounts" -ApiVersion "2015-04-08" ` 
    -ResourceGroupName $resourceGroupName -Name $accountName -Parameters $failoverPolicies

```

## Azure Cosmos Database

The following sections demonstrate how to manage the Azure Cosmos database, including:

- [Create an Azure Cosmos database](#)
- [Create an Azure Cosmos database with shared throughput](#)
- [Get the throughput of an Azure Cosmos database](#)
- [List all Azure Cosmos databases in an account](#)
- [Get a single Azure Cosmos database](#)
- [Delete an Azure Cosmos database](#)

### Create an Azure Cosmos database

```

# Create an Azure Cosmos database
$resourceGroupName = "myResourceGroup"
$accountName = "mycosmosaccount"
$databaseName = "database1"
$resourceName = $accountName + "/sql/" + $databaseName

$DataBaseProperties = @{
    "resource"=@{ "id"=$databaseName}
}

New-AzResource -ResourceType "Microsoft.DocumentDb/databaseAccounts/apis/databases" ` 
    -ApiVersion "2015-04-08" -ResourceGroupName $resourceGroupName ` 
    -Name $resourceName -PropertyObject $DataBaseProperties

```

### Create an Azure Cosmos database with shared throughput

```

$resourceGroupName = "myResourceGroup"
$accountName = "mycosmosaccount"
$databaseName = "database2"
$resourceName = $accountName + "/sql/" + $databaseName

$DataBaseProperties = @{
    "resource"=@{ "id"=$databaseName };
    "options"=@{ "Throughput"="400" }
}

New-AzResource -ResourceType "Microsoft.DocumentDb/databaseAccounts/apis/databases" ` 
    -ApiVersion "2015-04-08" -ResourceGroupName $resourceGroupName ` 
    -Name $resourceName -PropertyObject $DataBaseProperties

```

## Get the throughput of an Azure Cosmos database

```

$resourceGroupName = "myResourceGroup"
$accountName = "mycosmosaccount"
$databaseName = "database1"
$containerName = "container1"
$databaseThroughputResourceType = "Microsoft.DocumentDb/databaseAccounts/apis/databases/settings"
$databaseThroughputResourceName = $accountName + "/sql/" + $databaseName + "/throughput"

Get-AzResource -ResourceType $databaseThroughputResourceType ` 
    -ApiVersion "2015-04-08" -ResourceGroupName $resourceGroupName ` 
    -Name $databaseThroughputResourceName | Select-Object Properties

```

## Get all Azure Cosmos databases in an account

```

# Get all databases in an Azure Cosmos account
$resourceGroupName = "myResourceGroup"
$accountName = "mycosmosaccount"
$resourceName = $accountName + "/sql/"

Get-AzResource -ResourceType "Microsoft.DocumentDb/databaseAccounts/apis/databases" ` 
    -ApiVersion "2015-04-08" -ResourceGroupName $resourceGroupName ` 
    -Name $resourceName | Select-Object Properties

```

## Get a single Azure Cosmos database

```

# Get a single database in an Azure Cosmos account
$resourceGroupName = "myResourceGroup"
$accountName = "mycosmosaccount"
$databaseName = "database1"
$resourceName = $accountName + "/sql/" + $databaseName

Get-AzResource -ResourceType "Microsoft.DocumentDb/databaseAccounts/apis/databases" ` 
    -ApiVersion "2015-04-08" -ResourceGroupName $resourceGroupName ` 
    -Name $resourceName | Select-Object Properties

```

## Delete an Azure Cosmos database

```

# Delete a database in an Azure Cosmos account
$resourceGroupName = "myResourceGroup"
$accountName = "mycosmosaccount"
$databaseName = "database1"
$resourceName = $accountName + "/sql/" + $databaseName

Remove-AzResource -ResourceType "Microsoft.DocumentDb/databaseAccounts/apis/databases" ` 
    -ApiVersion "2015-04-08" -ResourceGroupName $resourceGroupName -Name $resourceName

```

# Azure Cosmos Container

The following sections demonstrate how to manage the Azure Cosmos container, including:

- [Create an Azure Cosmos container](#)
- [Create an Azure Cosmos container with a large partition key](#)
- [Get the throughput of an Azure Cosmos container](#)
- [Create an Azure Cosmos container with shared throughput](#)
- [Create an Azure Cosmos container with custom indexing](#)
- [Create an Azure Cosmos container with indexing turned off](#)
- [Create an Azure Cosmos container with unique key and TTL](#)
- [Create an Azure Cosmos container with conflict resolution](#)
- [List all Azure Cosmos containers in a database](#)
- [Get a single Azure Cosmos container in a database](#)
- [Delete an Azure Cosmos container](#)

## Create an Azure Cosmos container

```
# Create an Azure Cosmos container with default indexes and throughput at 400 RU
$resourceGroupName = "myResourceGroup"
$accountName = "mycosmosaccount"
$databaseName = "database1"
$containerName = "container1"
$resourceName = $accountName + "/sql/" + $databaseName + "/" + $containerName

$ContainerProperties = @{
    "resource"=@{
        "id"=$containerName;
        "partitionKey"=@{
            "paths"=@("/myPartitionKey");
            "kind"="Hash"
        }
    };
    "options"=@{ "Throughput"="400" }
}

New-AzResource -ResourceType "Microsoft.DocumentDb/databaseAccounts/apis/databases/containers" ` 
    -ApiVersion "2015-04-08" -ResourceGroupName $resourceGroupName ` 
    -Name $resourceName -PropertyObject $ContainerProperties
```

## Create an Azure Cosmos container with a large partition key size

```

# Create an Azure Cosmos container with a large partition key value (version = 2)
$resourceGroupName = "myResourceGroup"
$accountName = "mycosmosaccount"
$databaseName = "database1"
$containerName = "container1"
$resourceName = $accountName + "/sql/" + $databaseName + "/" + $containerName

$ContainerProperties = @{
    "resource"=@{
        "id"=$containerName;
        "partitionKey"=@{
            "paths"=@("/myPartitionKey");
            "kind"="Hash";
            "version" = 2
        }
    };
    "options"=@{ "Throughput"="400" }
}

New-AzResource -ResourceType "Microsoft.DocumentDb/databaseAccounts/apis/databases/containers" ` 
    -ApiVersion "2015-04-08" -ResourceGroupName $resourceGroupName ` 
    -Name $resourceName -PropertyObject $ContainerProperties

```

## Get the throughput of an Azure Cosmos container

```

$resourceGroupName = "myResourceGroup"
$accountName = "mycosmosaccount"
$databaseName = "database1"
$containerName = "container1"
$containerThroughputResourceType = "Microsoft.DocumentDb/databaseAccounts/apis/databases/containers/settings"
$containerThroughputResourceName = $accountName + "/sql/" + $databaseName + "/" + $containerName +
"/throughput"

Get-AzResource -ResourceType $containerThroughputResourceType ` 
    -ApiVersion "2015-04-08" -ResourceGroupName $resourceGroupName ` 
    -Name $containerThroughputResourceName | Select-Object Properties

```

## Create an Azure Cosmos container with shared throughput

```

$resourceGroupName = "myResourceGroup"
$accountName = "mycosmosaccount"
$databaseName = "database1"
$containerName = "container2"
$resourceName = $accountName + "/sql/" + $databaseName + "/" + $containerName

$ContainerProperties = @{
    "resource"=@{
        "id"=$containerName;
        "partitionKey"=@{
            "paths"=@("/myPartitionKey");
            "kind"="Hash"
        }
    };
    "options"=@{}
}

New-AzResource -ResourceType "Microsoft.DocumentDb/databaseAccounts/apis/databases/containers" ` 
    -ApiVersion "2015-04-08" -ResourceGroupName $resourceGroupName ` 
    -Name $resourceName -PropertyObject $ContainerProperties

```

## Create an Azure Cosmos container with custom index policy

```

# Create a container with a custom indexing policy
$resourceGroupName = "myResourceGroup"
$accountName = "mycosmosaccount"
$databaseName = "database1"
$containerName = "container3"
$resourceName = $accountName + "/sql/" + $databaseName + "/" + $containerName

$ContainerProperties = @{
    "resource"=@{
        "id"=$containerName;
        "partitionKey"=@{
            "paths"=@("/myPartitionKey");
            "kind"="Hash"
        };
        "indexingPolicy"=@{
            "indexingMode"="Consistent";
            "includedPaths"= @(@{
                "path"="/";
                "indexes"= @()
            });
            "excludedPaths"= @(@{
                "path"="/myPathToNotIndex/*"
            })
        }
    };
    "options"=@{ "Throughput"="400" }
}

New-AzResource -ResourceType "Microsoft.DocumentDb/databaseAccounts/apis/databases/containers" ` 
-ApiVersion "2015-04-08" -ResourceGroupName $resourceGroupName ` 
-Name $resourceName -PropertyObject $ContainerProperties

```

## Create an Azure Cosmos container with indexing turned off

```

# Create an Azure Cosmos container with no indexing
$resourceGroupName = "myResourceGroup"
$accountName = "mycosmosaccount"
$databaseName = "database1"
$containerName = "container4"
$resourceName = $accountName + "/sql/" + $databaseName + "/" + $containerName

$ContainerProperties = @{
    "resource"=@{
        "id"=$containerName;
        "partitionKey"=@{
            "paths"=@("/myPartitionKey");
            "kind"="Hash"
        };
        "indexingPolicy"=@{
            "indexingMode"="none"
        }
    };
    "options"=@{ "Throughput"="400" }
}

New-AzResource -ResourceType "Microsoft.DocumentDb/databaseAccounts/apis/databases/containers" ` 
-ApiVersion "2015-04-08" -ResourceGroupName $resourceGroupName ` 
-Name $resourceName -PropertyObject $ContainerProperties

```

## Create an Azure Cosmos 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 = "database1"
$containerName = "container5"
$resourceName = $accountName + "/sql/" + $databaseName + "/" + $containerName

$ContainerProperties = @{
    "resource"=@{
        "id"=$containerName;
        "partitionKey"=@{
            "paths"=@("/myPartitionKey");
            "kind"="Hash"
        };
        "indexingPolicy"=@{
            "indexingMode"="Consistent";
            "includedPaths"= @(@{
                "path"="/";
                "indexes"= @()
            });
            "excludedPaths"= @()
        };
        "uniqueKeyPolicy"= @{
            "uniqueKeys"= @(@{
                "paths"= @(
                    "/myUniqueKey1";
                    "/myUniqueKey2"
                )
            })
        };
        "defaultTtl"= 86400;
    };
    "options"=@{ "Throughput"="400" }
}

New-AzResource -ResourceType "Microsoft.DocumentDb/databaseAccounts/apis/databases/containers" ` 
    -ApiVersion "2015-04-08" -ResourceGroupName $resourceGroupName ` 
    -Name $resourceName -PropertyObject $ContainerProperties

```

### Create an Azure Cosmos container with conflict resolution

To create a conflict resolution policy to use a stored procedure, set `"mode"="custom"` and set the resolution path as the name of the stored procedure, `"conflictResolutionPath"="myResolverStoredProcedure"`. To write all conflicts to the ConflictsFeed and handle separately, set `"mode"="custom"` and `"conflictResolutionPath""`

```

# Create container with last-writer-wins conflict resolution policy
$resourceGroupName = "myResourceGroup"
$accountName = "mycosmosaccount"
$databaseName = "database1"
$containerName = "container6"
$resourceName = $accountName + "/sql/" + $databaseName + "/" + $containerName

$ContainerProperties = @{
    "resource"=@{
        "id"=$containerName;
        "partitionKey"=@{
            "paths"=@("/myPartitionKey");
            "kind"="Hash"
        };
        "conflictResolutionPolicy"=@{
            "mode"="lastWriterWins";
            "conflictResolutionPath"="/myResolutionPath"
        }
    };
    "options"=@{ "Throughput"="400" }
}

New-AzResource -ResourceType "Microsoft.DocumentDb/databaseAccounts/apis/databases/containers" ` 
    -ApiVersion "2015-04-08" -ResourceGroupName $resourceGroupName ` 
    -Name $resourceName -PropertyObject $ContainerProperties

```

## List all Azure Cosmos containers in a database

```

# List all Azure Cosmos containers in a database
$resourceGroupName = "myResourceGroup"
$accountName = "mycosmosaccount"
$databaseName = "database1"
$resourceName = $accountName + "/sql/" + $databaseName

Get-AzResource -ResourceType "Microsoft.DocumentDb/databaseAccounts/apis/databases/containers" ` 
    -ApiVersion "2015-04-08" -ResourceGroupName $resourceGroupName ` 
    -Name $resourceName | Select-Object Properties

```

## Get a single Azure Cosmos container in a database

```

# Get a single Azure Cosmos container in a database
$resourceGroupName = "myResourceGroup"
$accountName = "mycosmosaccount"
$databaseName = "database1"
$containerName = "container5"
$resourceName = $accountName + "/sql/" + $databaseName + "/" + $containerName

Get-AzResource -ResourceType "Microsoft.DocumentDb/databaseAccounts/apis/databases/containers" ` 
    -ApiVersion "2015-04-08" -ResourceGroupName $resourceGroupName ` 
    -Name $resourceName | Select-Object Properties

```

## Delete an Azure Cosmos container

```
# Delete an Azure Cosmos container
$resourceGroupName = "myResourceGroup"
$accountName = "mycosmosaccount"
$databaseName = "database1"
$containerName = "container1"
$resourceName = $accountName + "/sql/" + $databaseName + "/" + $containerName

Remove-AzResource -ResourceType "Microsoft.DocumentDb/databaseAccounts/apis/databases/containers" ` 
    -ApiVersion "2015-04-08" -ResourceGroupName $resourceGroupName -Name $resourceName
```

## Next steps

- [All PowerShell Samples](#)
- [How to manage Azure Cosmos account](#)
- [Create an Azure Cosmos container](#)
- [Configure time-to-live in Azure Cosmos DB](#)

# Manage Azure Cosmos resources using Azure CLI

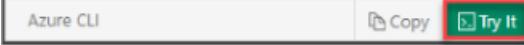
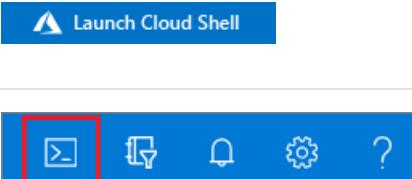
1/23/2020 • 7 minutes to read • [Edit Online](#)

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.

## 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 <b>Try It</b> in the upper-right corner of a code block. Selecting <b>Try It</b> doesn't automatically copy the code to Cloud Shell.	
Go to <a href="https://shell.azure.com">https://shell.azure.com</a> , or select the <b>Launch Cloud Shell</b> button to open Cloud Shell in your browser.	
Select the <b>Cloud Shell</b> button on the menu bar at the upper right in the <a href="#">Azure portal</a> .	

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.

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](#).

## 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 31 characters.

```
resourceGroupName='MyResourceGroup'
accountName='mycosmosaccount' #needs to be lower case and less than 31 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' # must be lower case and <31 characters

# 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-master for a Cosmos account

```
# Update an Azure Cosmos account from single to multi-master
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
```

## 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

```

## Change the throughput of a database

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

```

## 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 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 the throughput of a container

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
```

## 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 SQL (Core) API resources with Azure Resource Manager templates

2/24/2020 • 11 minutes to read • [Edit Online](#)

In this article, you learn how to use Azure Resource Manager templates to help automate management of your Azure Cosmos DB accounts, databases, and containers.

This article only shows Azure Resource Manager template examples for SQL API accounts. You can also find template examples for [Cassandra](#), [Gremlin](#), [MongoDB](#), and [Table](#) APIs.

## Create an Azure Cosmos account, database, and container

The following Azure Resource Manager template creates an Azure Cosmos account with:

- Two containers that share 400 Requested Units per second (RU/s) throughput at the database level.
- One container with dedicated 400 RU/s throughput.

To create the Azure Cosmos DB resources, copy the following example template and deploy it as described, either via [PowerShell](#) or [Azure CLI](#).

- Optionally, you can visit the [Azure Quickstart Gallery](#) and deploy the template from the Azure portal.
- You can also download the template to your local computer or create a new template and specify the local path with the `--template-file` parameter.

### IMPORTANT

- When you add or remove locations to an Azure Cosmos account, you can't simultaneously modify other properties. These operations must be done separately.
- Account names are limited to 44 characters, all lowercase.
- To change the throughput values, resubmit the template with updated RU/s.

```
{  
  "$schema": "https://schema.management.azure.com/schemas/2015-01-01/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."  
      }  
    }  
  },  
  "resources": [  
    {  
      "type": "Microsoft.DocumentDB/databaseAccounts/sql/  
      "name": "[parameters('accountName')]",  
      "apiVersion": "2017-02-28",  
      "location": "[parameters('location')]",  
      "properties": {  
        "throughput": 400,  
        "enableAutomaticDiagnosticsCollection": {  
          "diagnosticType": "Metrics",  
          "isStorage": true  
        },  
        "enableAutomaticTuning": true  
      }  
    },  
    {  
      "type": "Microsoft.DocumentDB/databaseAccounts/sql/databases/  
      "name": "[parameters('accountName')]/dbs/[guid()]",  
      "apiVersion": "2017-02-28",  
      "location": "[parameters('location')]",  
      "dependsOn": "[resourceId('Microsoft.DocumentDB/databaseAccounts/sql', parameters('accountName'))]",  
      "properties": {  
        "throughput": 400  
      }  
    },  
    {  
      "type": "Microsoft.DocumentDB/databaseAccounts/sql/databases/  
      "name": "[parameters('accountName')]/dbs/[guid()]/containers/  
      "name": "[guid()]",  
      "apiVersion": "2017-02-28",  
      "location": "[parameters('location')]",  
      "dependsOn": "[resourceId('Microsoft.DocumentDB/databaseAccounts/sql/databases', parameters('accountName'))]",  
      "properties": {  
        "throughput": 400  
      }  
    }  
  ]  
}
```

```
},
"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."
    }
},
"multipleWriteLocations": {
    "type": "bool",
    "defaultValue": false,
    "allowedValues": [ true, false ],
    "metadata": {
        "description": "Enable multi-master to make all regions writable."
    }
},
"automaticFailover": {
    "type": "bool",
    "defaultValue": false,
    "allowedValues": [ true, false ],
    "metadata": {
        "description": "Enable automatic failover for regions. Ignored when Multi-Master is enabled"
    }
},
"databaseName": {
    "type": "string",
    "metadata": {
        "description": "The name for the SQL database"
    }
},
"sharedThroughput": {
    "type": "int",
    "defaultValue": 400,
    "minValue": 400,
    "maxValue": 1000000,
    "metadata": {
        "description": "The throughput for the database to be shared"
    }
},
"sharedContainer1Name": {
    "type": "string",
    "defaultValue": "sharedContainer1",
    "metadata": {
        "description": "The name for the shared container"
    }
}
```

```
"metadata": {
    "description": "The name for the first container with shared throughput"
}
},
"sharedContainer2Name": {
    "type": "string",
    "defaultValue": "sharedContainer2",
    "metadata": {
        "description": "The name for the second container with shared throughput"
    }
},
"dedicatedContainer1Name": {
    "type": "string",
    "defaultValue": "dedicatedContainer1",
    "metadata": {
        "description": "The name for the container with dedicated throughput"
    }
},
"dedicatedThroughput": {
    "type": "int",
    "defaultValue": 400,
    "minValue": 400,
    "maxValue": 1000000,
    "metadata": {
        "description": "The throughput for the container with dedicated throughput"
    }
},
},
"variables": {
    "accountName": "[toLowerCase(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": "2019-08-01",
    "dependsOn": [
        "[resourceId('Microsoft.DocumentDB/databaseAccounts', variables('accountName'))]"
    ],
    "properties": {
        "name": "[variables('accountName')]",
        "location": "[parameters('primaryRegion')]",
        "OfferType": "Dedicated",
        "OfferThroughput": 400,
        "consistencyPolicy": {
            "defaultConsistencyLevel": "Eventual"
        },
        "partitionKeyPath": null
    }
}
]
```

```

"kind": "GlobalDocumentDB",
"location": "[parameters('location')]",
"properties": {
  "consistencyPolicy": "[variables('consistencyPolicy')[parameters('defaultConsistencyLevel')]]",
  "locations": "[variables('locations')]",
  "databaseAccountOfferType": "Standard",
  "enableAutomaticFailover": "[parameters('automaticFailover')]",
  "enableMultipleWriteLocations": "[parameters('multipleWriteLocations')]"
}
},
{
  "type": "Microsoft.DocumentDB/databaseAccounts/sqlDatabases",
  "name": "[concat(variables('accountName'), '/', parameters('databaseName'))]",
  "apiVersion": "2019-08-01",
  "dependsOn": [ "[resourceId('Microsoft.DocumentDB/databaseAccounts', variables('accountName'))]" ],
  "properties": {
    "resource": {
      "id": "[parameters('databaseName')]"
    },
    "options": { "throughput": "[parameters('sharedThroughput')]" }
  }
},
{
  "type": "Microsoft.DocumentDB/databaseAccounts/sqlDatabases/containers",
  "name": "[concat(variables('accountName'), '/', parameters('databaseName'), '/', parameters('sharedContainer1Name'))]",
  "apiVersion": "2019-08-01",
  "dependsOn": [ "[resourceId('Microsoft.DocumentDB/databaseAccounts/sqlDatabases', variables('accountName')), parameters('databaseName'))]" ],
  "properties": {
    "resource": {
      "id": "[parameters('sharedContainer1Name')]",
      "partitionKey": {
        "paths": [
          "/myPartitionKey"
        ],
        "kind": "Hash"
      },
      "indexingPolicy": {
        "indexingMode": "consistent",
        "includedPaths": [
          {
            "path": "/*"
          }
        ],
        "excludedPaths": [
          {
            "path": "/myPathToNotIndex/*"
          }
        ]
      }
    }
  }
},
{
  "type": "Microsoft.DocumentDB/databaseAccounts/sqlDatabases/containers",
  "name": "[concat(variables('accountName'), '/', parameters('databaseName'), '/', parameters('sharedContainer2Name'))]",
  "apiVersion": "2019-08-01",
  "dependsOn": [ "[resourceId('Microsoft.DocumentDB/databaseAccounts/sqlDatabases', variables('accountName')), parameters('databaseName'))]" ],
  "properties": {
    "resource": {
      "id": "[parameters('sharedContainer2Name')]",
      "partitionKey": {
        "paths": [
          "/myPartitionKey"
        ],
        "kind": "Hash"
      }
    }
  }
}

```

```
        },
        "indexingPolicy": {
            "indexingMode": "consistent",
            "includedPaths": [
                {
                    "path": "*"
                }
            ],
            "excludedPaths": [
                {
                    "path": "/myPathToNotIndex/*"
                }
            ]
        }
    }
},
{
    "type": "Microsoft.DocumentDB/databaseAccounts/sqlDatabases/containers",
    "name": "[concat(variables('accountName'), '/', parameters('databaseName'), '/', parameters('dedicatedContainer1Name'))]",
    "apiVersion": "2019-08-01",
    "dependsOn": [ "[resourceId('Microsoft.DocumentDB/databaseAccounts/sqlDatabases', variables('accountName'), parameters('databaseName'))]" ],
    "properties":
    {
        "resource":{
            "id": "[parameters('dedicatedContainer1Name')]",
            "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": {
            "uniqueKeyPolicy": [
                {
                    "path": "/name"
                }
            ]
        }
    }
}
```

```

        "uniquekeys": [
            {
                "paths": [
                    "/phoneNumber"
                ]
            }
        ],
        "options": { "throughput": "[parameters('dedicatedThroughput')]" }
    }
}
]
}

```

#### NOTE

To create a container with large partition key, modify the previous template to include the `"version":2` property within the `partitionKey` object.

## Deploy via PowerShell

To use PowerShell to deploy the Azure Resource Manager template:

1. **Copy** the script.
2. Select **Try it** to open Azure Cloud Shell.
3. Right-click in the Azure Cloud Shell window, and then select **Paste**.

```

$resourceGroupName = Read-Host -Prompt "Enter the Resource Group name"
$accountName = Read-Host -Prompt "Enter the account name"
.setLocation = Read-Host -Prompt "Enter the location (i.e. westus2)"
$primaryRegion = Read-Host -Prompt "Enter the primary region (i.e. westus2)"
$secondaryRegion = Read-Host -Prompt "Enter the secondary region (i.e. eastus2)"
$databaseName = Read-Host -Prompt "Enter the database name"
$sharedThroughput = Read-Host -Prompt "Enter the shared database throughput (i.e. 400)"
$sharedContainer1Name = Read-Host -Prompt "Enter the first shared container name"
$sharedContainer2Name = Read-Host -Prompt "Enter the second shared container name"
$dedicatedContainer1Name = Read-Host -Prompt "Enter the dedicated container name"
$dedicatedThroughput = Read-Host -Prompt "Enter the dedicated container throughput (i.e. 400)"

New-AzResourceGroup -Name $resourceGroupName -Location $location
New-AzResourceGroupDeployment ` 
    -ResourceGroupName $resourceGroupName ` 
    -TemplateUri "https://raw.githubusercontent.com/Azure/azure-quickstart-templates/master/101-cosmosdb-` 
    sql/azuredeploy.json" ` 
    -accountName $accountName ` 
    -location $location ` 
    -primaryRegion $primaryRegion ` 
    -secondaryRegion $secondaryRegion ` 
    -databaseName $databaseName ` 
    -sharedThroughput $ $sharedThroughput ` 
    -sharedContainer1Name $sharedContainer1Name ` 
    -sharedContainer2Name $sharedContainer2Name ` 
    -dedicatedContainer1Name $dedicatedContainer1Name ` 
    -dedicatedThroughput $dedicatedThroughput

(Get-AzResource --ResourceType "Microsoft.DocumentDb/databaseAccounts" --ApiVersion "2019-08-01" --` 
ResourceGroupName $resourceGroupName).name

```

You can choose to deploy the template with a locally installed version of PowerShell instead of Azure Cloud Shell. You'll need to [install the Azure PowerShell module](#). Run `Get-Module -ListAvailable Az` to find the required version.

## Deploy via Azure CLI

To use Azure CLI to deploy the Azure Resource Manager template:

1. **Copy** the script.
2. Select **Try it** to open Azure Cloud Shell.
3. Right-click in the Azure Cloud Shell window, and then select **Paste**.

```
read -p 'Enter the Resource Group name: ' resourceGroupName
read -p 'Enter the location (i.e. westus2): ' location
read -p 'Enter the account name: ' accountName
read -p 'Enter the primary region (i.e. westus2): ' primaryRegion
read -p 'Enter the secondary region (i.e. eastus2): ' secondaryRegion
read -p 'Enter the database name: ' databaseName
read -p 'Enter the shared database throughput: sharedThroughput
read -p 'Enter the first shared container name: ' sharedContainer1Name
read -p 'Enter the second shared container name: ' sharedContainer2Name
read -p 'Enter the dedicated container name: ' dedicatedContainer1Name
read -p 'Enter the dedicated container throughput: dedicatedThroughput

az group create --name $resourceGroupName --location $location
az group deployment create --resource-group $resourceGroupName \
    --template-uri https://raw.githubusercontent.com/azure/azure-quickstart-templates/master/101-cosmosdb-
sql/azuredploy.json \
    --parameters accountName=$accountName \
    primaryRegion=$primaryRegion \
    secondaryRegion=$secondaryRegion \
    databaseName=$databaseName \
    sharedThroughput=$sharedThroughput \
    sharedContainer1Name=$sharedContainer1Name \
    sharedContainer2Name=$sharedContainer2Name \
    dedicatedContainer1Name=$dedicatedContainer1Name \
    dedicatedThroughput=$dedicatedThroughput

az cosmosdb show --resource-group $resourceGroupName --name accountName --output tsv
```

The `az cosmosdb show` command shows the newly created Azure Cosmos account after it's provisioned. You can choose to deploy the template with a locally installed version of Azure CLI instead Azure Cloud Shell. For more information, see the [Azure Command-Line Interface \(CLI\)](#) article.

## Create an Azure Cosmos DB container with server-side functionality

You can use an Azure Resource Manager template to create an Azure Cosmos DB container with a stored procedure, trigger, and user-defined function.

Copy the following example template and deploy it as described, either with [PowerShell](#) or [Azure CLI](#).

- Optionally, you can visit [Azure Quickstart Gallery](#) and deploy the template from the Azure portal.
- You can also download the template to your local computer or create a new template and specify the local path with the `--template-file` parameter.

```
{
    "$schema": "https://schema.management.azure.com/schemas/2015-01-01/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 86400. Multi Region: 300 to 86400."
            }
        },
        "multipleWriteLocations": {
            "type": "bool",
            "defaultValue": false,
            "allowedValues": [ true, false ],
            "metadata": {
                "description": "Enable multi-master to make all regions writable."
            }
        },
        "automaticFailover": {
            "type": "bool",
            "defaultValue": false,
            "allowedValues": [ true, false ],
            "metadata": {
                "description": "Enable automatic failover for regions. Ignored when Multi-Master is enabled"
            }
        },
        "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": "[toLowerCase(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": "2019-08-01",
            "location": "[parameters('location')]",
            "kind": "GlobalDocumentDB",
            "properties": {
                "consistencyPolicy": "[variables('consistencyPolicy')[parameters('defaultConsistencyLevel')]]",
                "locations": "[variables('locations')]",
                "databaseAccountOfferType": "Standard",
                "enableAutomaticFailover": "[parameters('automaticFailover')]",
                "enableMultipleWriteLocations": "[parameters('multipleWriteLocations')]"
            }
        },
    ],
    "outputs": [
        {
            "name": "connectionString",
            "type": "string",
            "value": "[listKeys(resourceId('Microsoft.DocumentDB/databaseAccounts', variables('accountName')), '2019-08-01').connectionStrings.primaryConnectionString]"
        }
    ]
}
```

```
{
    "type": "Microsoft.DocumentDB/databaseAccounts/sqlDatabases",
    "name": "[concat(variables('accountName'), '/', parameters('databaseName'))]",
    "apiVersion": "2019-08-01",
    "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": "2019-08-01",
    "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": "2019-08-01",
            "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": "2019-08-01",
            "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": "2019-08-01",
          "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;}"
            }
          }
        }
      ]
    }
  ]
}

```

## Deploy with PowerShell

To use PowerShell to deploy the Azure Resource Manager template:

1. **Copy** the script.
2. Select **Try it** to open Azure Cloud Shell.
3. Right-click the Azure Cloud Shell window, and then select **Paste**.

```

$resourceGroupName = Read-Host -Prompt "Enter the Resource Group name"
$accountName = Read-Host -Prompt "Enter the account name"
$location = Read-Host -Prompt "Enter the location (i.e. westus2)"
$primaryRegion = Read-Host -Prompt "Enter the primary region (i.e. westus2)"
$secondaryRegion = Read-Host -Prompt "Enter the secondary region (i.e. eastus2)"
$databaseName = Read-Host -Prompt "Enter the database name"
$containerName = Read-Host -Prompt "Enter the container name"

New-AzResourceGroup -Name $resourceGroupName -Location $location
New-AzResourceGroupDeployment ` 
  -ResourceGroupName $resourceGroupName ` 
  -TemplateUri "https://raw.githubusercontent.com/Azure/azure-quickstart-templates/master/101-cosmosdb-sql-container-sprocs/azuredeploy.json" ` 
  -accountName $accountName ` 
  -location $location ` 
  -primaryRegion $primaryRegion ` 
  -secondaryRegion $secondaryRegion ` 
  -databaseName $databaseName ` 
  -containerName $containerName

(Get-AzResource --ResourceType "Microsoft.DocumentDb/databaseAccounts" --ApiVersion "2019-08-01" -- 
ResourceGroupName $resourceGroupName).name

```

You can choose to deploy the template with a locally installed version of PowerShell instead of Azure Cloud Shell. You'll need to [install the Azure PowerShell module](#). Run `Get-Module -ListAvailable Az` to find the required version.

## Deploy with Azure CLI

To use Azure CLI to deploy the Azure Resource Manager template:

1. **Copy** the script.
2. Select **Try it** to open Azure Cloud Shell.
3. Right-click in the Azure Cloud Shell window, and then select **Paste**.

```
read -p 'Enter the Resource Group name: ' resourceGroupName
read -p 'Enter the location (i.e. westus2): ' location
read -p 'Enter the account name: ' accountName
read -p 'Enter the primary region (i.e. westus2): ' primaryRegion
read -p 'Enter the secondary region (i.e. eastus2): ' secondaryRegion
read -p 'Enter the database name: ' databaseName
read -p 'Enter the container name: ' containerName

az group create --name $resourceGroupName --location $location
az group deployment create --resource-group $resourceGroupName \
    --template-uri https://raw.githubusercontent.com/azure/azure-quickstart-templates/master/101-cosmosdb-sql-
    container-sprocs/azuredeploy.json \
    --parameters accountName=$accountName primaryRegion=$primaryRegion secondaryRegion=$secondaryRegion
    databaseName=$databaseName \
    containerName=$containerName

az cosmosdb show --resource-group $resourceGroupName --name accountName --output tsv
```

## 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](#)

# Configure multi-master in your applications that use Azure Cosmos DB

12/5/2019 • 2 minutes to read • [Edit Online](#)

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-master 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 (i.e. multi-master) with zero down time. To learn more see, [Configure multiple-write regions](#)

## .NET SDK v2

To enable multi-master 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-master 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 Async SDK

To enable multi-master 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-master 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-master 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

12/13/2019 • 5 minutes to read • [Edit Online](#)

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. Clients can always override it.

### CLI

```
# create with a default consistency
az cosmosdb create --name <name of Cosmos DB Account> --resource-group <resource group name> --default-
consistency-level Session

# update an existing account's default consistency
az cosmosdb update --name <name of Cosmos DB Account> --resource-group <resource group name> --default-
consistency-level Eventual
```

### PowerShell

This example creates a new Azure Cosmos account with multiple write regions enabled, in East US and West US regions. The default consistency level is set to *Session* consistency.

```
$locations = @(@{"locationName"="East US"; "failoverPriority"=0},
              @{"locationName"="West US"; "failoverPriority"=1})

$iprangepickerfilter = ""

$consistencyPolicy = @{"defaultConsistencyLevel"="Session"}

$cosmosDBProperties = @{"databaseAccountOfferType"="Standard";
                        "locations"=$locations;
                        "consistencyPolicy"=$consistencyPolicy;
                        "ipRangeFilter"=$iprangepickerfilter;
                        "enableMultipleWriteLocations"="true"}

New-AzResource -ResourceType "Microsoft.DocumentDb/databaseAccounts" ` 
    -ApiVersion "2015-04-08" ` 
    -ResourceGroupName "myResourceGroup" ` 
    -Location "East US" ` 
    -Name "myCosmosDbAccount" ` 
    -Properties $cosmosDBProperties
```

### Azure portal

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.

The screenshot shows the Azure portal's 'Default consistency' settings for a database named 'sqlcdb'. The 'SESSION' tab is selected. A note states: 'Session consistency is most widely used consistency level both for single region as well as, globally distributed applications.' To the right, there's a 'Understand Session consistency' section with a world map and a musical staff diagram. The staff has four tracks labeled from top to bottom: 'East US Write in Session A', 'East US Reads in Session A', 'North Europe Reads in Session A', and 'Australia Southeast Reads in Session B'. Each track contains a series of colored musical notes (red, blue, green, orange) representing operations across regions.

## 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.

### .NET SDK V2

```
// 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);
```

### .NET SDK V3

```
// Override consistency at the request level via request options
ItemRequestOptions requestOptions = new ItemRequestOptions { ConsistencyLevel = ConsistencyLevel.Strong };

var response = await client.GetContainer(databaseName, containerName)
    .CreateItemAsync(
        item,
        new PartitionKey(itemPartitionKey),
        requestOptions);
```

### Java Async SDK

```
// 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();
```

## Java Sync SDK

```
// Override consistency at the client level
ConnectionPolicy connectionPolicy = new ConnectionPolicy();
DocumentClient client = new DocumentClient(accountEndpoint, accountKey, connectionPolicy,
ConsistencyLevel.Eventual);
```

## 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 V2

```
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);
```

### .NET SDK V3

```

Container container = client.GetContainer(databaseName, collectionName);
ItemResponse<SalesOrder> response = await container.CreateItemAsync<SalesOrder>(salesOrder);
string sessionToken = response.Headers.Session;

ItemRequestOptions options = new ItemRequestOptions();
options.SessionToken = sessionToken;
ItemResponse<SalesOrder> response = await container.ReadItemAsync<SalesOrder>(salesOrder.Id, new
PartitionKey(salesOrder.PartitionKey), options);

```

## Java Async SDK

```

// 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);

```

## Java Sync SDK

```

// Get session token from response
ResourceResponse<Document> response = client.readDocument(documentLink, null);
String sessionToken = response.getSessionToken();

// Resume the session by setting the session token on the RequestOptions
RequestOptions options = new RequestOptions();
options.setSessionToken(sessionToken);
ResourceResponse<Document> response = client.readDocument(documentLink, 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

```

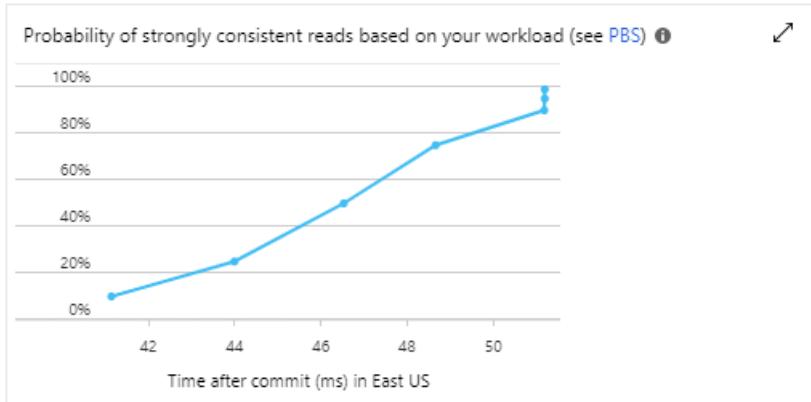
// Get the session token from the last response headers
item = client.ReadItem(item_link)
session_token = client.last_response_headers["x-ms-session-token"]

// 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](#)
- [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

12/13/2019 • 6 minutes to read • [Edit Online](#)

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 V2

```
DocumentCollection lwwCollection = await createClient.CreateDocumentCollectionIfNotExistsAsync(
    UriFactory.CreateDatabaseUri(this.databaseName), new DocumentCollection
    {
        Id = this.lwwCollectionName,
        ConflictResolutionPolicy = new ConflictResolutionPolicy
        {
            Mode = ConflictResolutionMode.LastWriterWins,
            ConflictResolutionPath = "/myCustomId",
        },
    });
});
```

### .NET SDK V3

```
Container container = await createClient.GetDatabase(this.databaseName)
    .CreateContainerIfNotExistsAsync(new ContainerProperties(this.lwwCollectionName, "/partitionKey")
    {
        ConflictResolutionPolicy = new ConflictResolutionPolicy()
        {
            Mode = ConflictResolutionMode.LastWriterWins,
            ResolutionPath = "/myCustomId",
        }
    });
});
```

### Java Async SDK

```
DocumentCollection collection = new DocumentCollection();
collection.setId(id);
ConflictResolutionPolicy policy = ConflictResolutionPolicy.createLastWriterWinsPolicy("/myCustomId");
collection.setConflictResolutionPolicy(policy);
DocumentCollection createdCollection = client.createCollection(databaseUri, collection,
    null).toBlocking().value();
```

### Java Sync SDK

```

DocumentCollection lwwCollection = new DocumentCollection();
lwwCollection.setId(this.lwwCollectionName);
ConflictResolutionPolicy lwwPolicy = ConflictResolutionPolicy.createLastWriterWinsPolicy("/myCustomId");
lwwCollection.setConflictResolutionPolicy(lwwPolicy);
DocumentCollection createdCollection = this.tryCreateDocumentCollection(createClient, database,
lwwCollection);

```

## 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;
            });
    }
}
```

```

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")
});

```

## .NET SDK V3

```

Container container = await createClient.GetDatabase(this.databaseName)
    .CreateContainerIfNotExistsAsync(new ContainerProperties(this.udpCollectionName, "/partitionKey")
{
    ConflictResolutionPolicy = new ConflictResolutionPolicy()
    {
        Mode = ConflictResolutionMode.Custom,
        ResolutionProcedure = string.Format("dbs/{0}/colls/{1}/sprocs/{2}", this.databaseName,
this.udpCollectionName, "resolver")
    }
});

await container.Scripts.CreateStoredProcedureAsync(
    new StoredProcedureProperties("resolver", File.ReadAllText(@"resolver.js")))
);

```

## Java Async SDK

```

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.

## Java Sync SDK

```

DocumentCollection udpCollection = new DocumentCollection();
udpCollection.setId(this.udpCollectionName);
ConflictResolutionPolicy udpPolicy = ConflictResolutionPolicy.createCustomPolicy(
    String.format("dbs/%s/colls/%s/sprocs/%s", this.databaseName, this.udpCollectionName, "resolver"));
udpCollection.setConflictResolutionPolicy(udpPolicy);
DocumentCollection createdCollection = this.tryCreateDocumentCollection(createClient, database,
    udpCollection);

```

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 V2

```
DocumentCollection manualCollection = await createClient.CreateDocumentCollectionIfNotExistsAsync(
  UriFactory.CreateDatabaseUri(this.databaseName), new DocumentCollection
  {
    Id = this.manualCollectionName,
    ConflictResolutionPolicy = new ConflictResolutionPolicy
    {
      Mode = ConflictResolutionMode.Custom,
    },
  });
});
```

### .NET SDK V3

```
Container container = await createClient.GetDatabase(this.databaseName)
  .CreateContainerIfNotExistsAsync(new ContainerProperties(this.manualCollectionName, "/partitionKey")
  {
    ConflictResolutionPolicy = new ConflictResolutionPolicy()
    {
      Mode = ConflictResolutionMode.Custom
    }
  });
});
```

## Java Async SDK

```
DocumentCollection collection = new DocumentCollection();
collection.setId(id);
ConflictResolutionPolicy policy = ConflictResolutionPolicy.createCustomPolicy();
collection.setConflictResolutionPolicy(policy);
DocumentCollection createdCollection = client.createCollection(databaseUri, collection,
null).toBlocking().value();
```

## Java Sync SDK

```
DocumentCollection manualCollection = new DocumentCollection();
manualCollection.setId(this.manualCollectionName);
ConflictResolutionPolicy customPolicy = ConflictResolutionPolicy.createCustomPolicy(null);
manualCollection.setConflictResolutionPolicy(customPolicy);
DocumentCollection createdCollection = client.createCollection(database.getSelfLink(), collection,
null).getResource();
```

## 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 V2

```
FeedResponse<Conflict> conflicts = await delClient.ReadConflictFeedAsync(this.collectionUri);
```

### .NET SDK V3

```

FeedIterator<ConflictProperties> conflictFeed = container.Conflicts.GetConflictQueryIterator();
while (conflictFeed.HasMoreResults)
{
    FeedResponse<ConflictProperties> conflicts = await conflictFeed.ReadNextAsync();
    foreach (ConflictProperties conflict in conflicts)
    {
        // Read the conflicted content
        MyClass intendedChanges = container.Conflicts.ReadConflictContent<MyClass>(conflict);
        MyClass currentState = await container.Conflicts.ReadCurrentAsync<MyClass>(conflict, new PartitionKey(intendedChanges.MyPartitionKey));

        // Do manual merge among documents
        await container.ReplaceItemAsync<MyClass>(intendedChanges, intendedChanges.Id, new PartitionKey(intendedChanges.MyPartitionKey));

        // Delete the conflict
        await container.Conflicts.DeleteAsync(conflict, new PartitionKey(intendedChanges.MyPartitionKey));
    }
}

```

## Java Async SDK

```

FeedResponse<Conflict> response = client.readConflicts(this.manualCollectionUri, null)
    .first().toBlocking().single();
for (Conflict conflict : response.getResults()) {
    /* Do something with conflict */
}

```

## Java Sync SDK

```

Iterator<Conflict> conflictsIterator = client.readConflicts(this.collectionLink, null).getQueryIterator();
while (conflictsIterator.hasNext()) {
    Conflict conflict = conflictsIterator.next();
    /* 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-master in your applications](#)

- [Configure clients for multihoming](#)
- [Add or remove regions from your Azure Cosmos DB account](#)
- [How to configure multi-master in your applications.](#)
- [Partitioning and data distribution](#)
- [Indexing in Azure Cosmos DB](#)

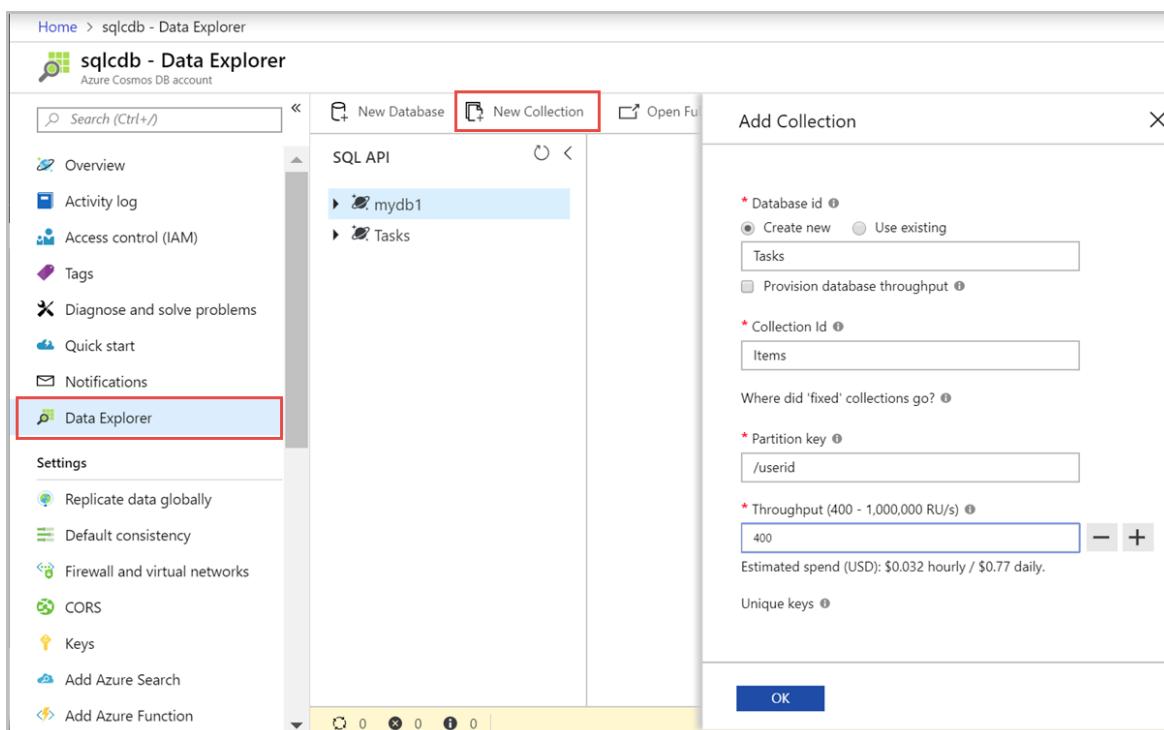
# Provision throughput on an Azure Cosmos container

2/24/2020 • 2 minutes to read • [Edit Online](#)

This article explains how to provision throughput on a container (collection, graph, or table) in Azure Cosmos DB. 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.

## 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.
  - Enter a Container (or Table or Graph) ID.
  - Enter a partition key value (for example, `/userid`).
  - 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](#)

#### **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. 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

#### **NOTE**

Use the Cosmos SDKs for SQL API to provision throughput for all Cosmos DB APIs, except Cassandra API.

### **SQL, MongoDB, Gremlin, and Table APIs**

#### **.Net V2 SDK**

```
// 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 });
```

#### **.Net V3 SDK**

```
// Create a container with a partition key and provision throughput of 1000 RU/s
string containerName = "myContainerName";
string partitionKeyPath = "/myPartitionKey";

await this.cosmosClient.GetDatabase("myDatabase").CreateContainerAsync(
    id: containerName,
    partitionKeyPath: partitionKeyPath,
    throughput: 1000);
```

## 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);

```

## Cassandra API

Similar commands can be issued through any CQL-compliant driver.

```

// 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");

```

## Alter or change throughput for 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");

```

## Next steps

See the following articles to learn about throughput provisioning in Azure Cosmos DB:

- [How to provision throughput on a database](#)
- [Request units and throughput in Azure Cosmos DB](#)

# Provision throughput on a database in Azure Cosmos DB

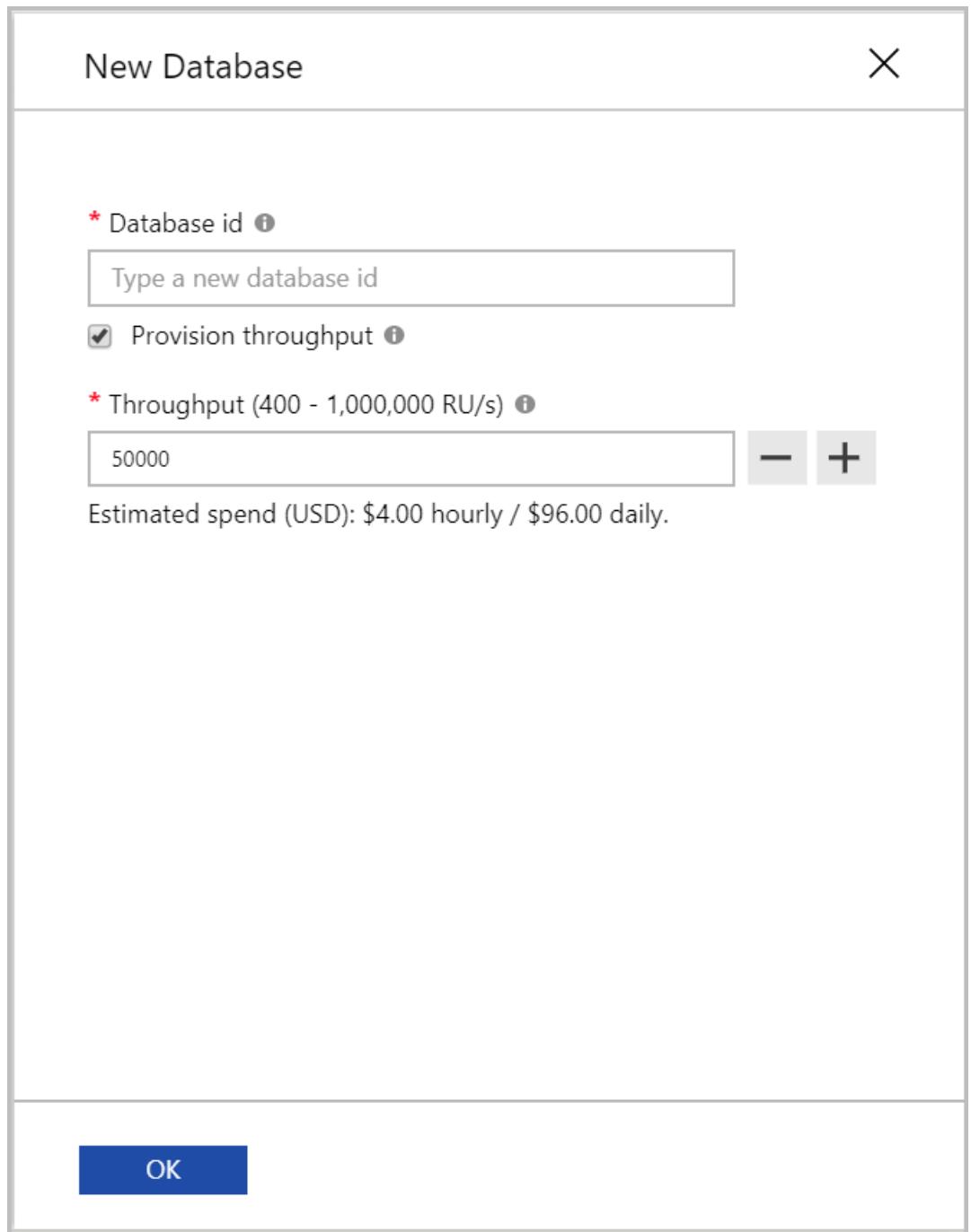
2/24/2020 • 2 minutes to read • [Edit Online](#)

This article explains how to provision throughput on a database in Azure Cosmos DB. 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.

## Provision throughput using Azure portal

### SQL (Core) API

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 **Provision throughput**.
  - 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 Cosmos SDKs for SQL API to provision throughput for all APIs. You can optionally use the following example for Cassandra API as well.

[All APIs](#)

[.Net V2 SDK](#)

```
//set the throughput for the database
RequestOptions options = new RequestOptions
{
    OfferThroughput = 500
};

//create the database
await client.CreateDatabaseIfNotExistsAsync(
    new Database {Id = databaseName},
    options);
```

## .Net V3 SDK

```
//create the database with throughput
string databaseName = "MyDatabaseName";
await this.cosmosClient.CreateDatabaseIfNotExistsAsync(
    id: databaseName,
    throughput: 1000);
```

## Cassandra API

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");
```

## 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 throughput for a container](#)
- [Request units and throughput in Azure Cosmos DB](#)

# Estimate RU/s using the Azure Cosmos DB capacity planner

8/1/2019 • 6 minutes to read • [Edit Online](#)

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	<p>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.</p> <p>Use basic mode for a quick, high-level estimate when you are evaluating a potential workload to run on Azure Cosmos DB.</p>
Advanced	<p>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.</p> <p>Use advanced mode when you are estimating RU/s for a new project or want a more detailed estimate.</p>

### 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 <a href="#">global distribution</a> in Azure Cosmos DB for more details.

INPUT	DESCRIPTION
Multi-region writes	If you enable <a href="#">multi-region writes</a> , 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 <a href="#">how RUs are different for single and multiple-write regions</a> 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.

Category	Description	Value
Storage	Cost per GB/month	0.250 USD
Storage	Total Data stored per region	x 10 GB
Storage	EST. STORAGE COST PER MONTH	2.50 USD
Workload	Cost per 100 RU/s per hour	0.008 USD
Workload	EST. THROUGHPUT REQUIRED <a href="#">Show Details</a>	x 595 RU/s
Workload	EST. WORKLOAD COST/MONTH	34.75 USD
Total Cost	Number of regions	x 1
Total Cost	EST. TOTAL COST/MONTH	37.25 USD

**SAVE UP TO 65% WITH RESERVED CAPACITY**  
[See here for more details](#)

**YOU WILL SAVE UP TO 70% TCO WITH COSMOS**  
[Learn more about Cosmos TCO](#)

## 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 <a href="#">global distribution</a> in Azure Cosmos DB for more details.
Multi-region writes	<p>If you enable <a href="#">multi-region writes</a>, 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.</p> <p>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.</p> <p>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 <a href="#">how RUs are different for single and multiple-write regions</a> article.</p>
Default consistency	<p>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 <a href="#">consistency levels</a> article.</p> <p>By default, Azure Cosmos DB uses session consistency, which guarantees the ability to read your own writes in a session.</p> <p>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.</p>
Indexing policy	<p>By default, Azure Cosmos DB <a href="#">indexes all properties</a> in all items for flexible and efficient queries (maps to the <b>Automatic</b> indexing policy).</p> <p>If you choose <b>off</b>, none of the properties are indexed. This results in the lowest RU charge for writes. Select <b>off</b> policy if you expect to only do <a href="#">point reads</a> (key value lookups) and/or writes, and no queries.</p> <p>Custom indexing policy allows you to include or exclude specific properties from the index for lower write throughput and storage. To learn more, see <a href="#">indexing policy</a> and <a href="#">sample indexing policies</a> articles.</p>
Total data stored (per region)	Total estimated data stored in GB in a single region.

INPUT	DESCRIPTION
Workload mode	<p>Select <b>Steady</b> if your workload volume is constant.</p> <p>Select <b>Variable</b> if your workload volume changes over time. For example, during a specific day or a month.</p> <p>The following settings are available if you choose the variable workload option:</p> <ul style="list-style-type: none"> <li>Percentage of time at peak: Percentage of time in a month where your workload requires peak (highest) throughput.</li> </ul> <p>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%.</p> <ul style="list-style-type: none"> <li>Reads/sec per region at peak - Number of reads expected per second at peak.</li> <li>Writes/sec per region at peak – Number of writes expected per second at peak.</li> <li>Reads/sec per region off peak – Number of reads expected per second during off peak.</li> <li>Writes/sec per region off peak – Number of writes expected per second during off peak.</li> </ul> <p>With peak and off-peak intervals, you can optimize your cost by <a href="#">programmatically scaling your provisioned throughput</a> up and down accordingly.</p>
Item size	<p>The size of the data item (e.g. document), ranging from 1 KB to 2 MB.</p> <p>You can also <b>Upload sample (JSON)</b> document for a more accurate estimate.</p> <p>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 <b>Add new item</b> button to add multiple sample JSON documents.</p>

You can also use the **Save Estimate** button to download a CSV file containing the current estimate.

**Cosmos Account Settings**

- API: SQL (Core)
- Number of regions: 1
- Multi-region writes: Disabled
- Default consistency: Session (Default)
- Indexing policy: Automatic

**Workload per region**

- Total data stored: 10 GB
- Workload mode: Variable (highlighted)
- Percentage of time at peak: 10% at Peak, 90% off Peak

**Sample item 1**

Item size: Specify size (1 KB)

Reads/sec per region at peak	1000
Writes/sec per region at peak	1000
Reads/sec per region off peak	500
Writes/sec per region off peak	500

+ Add new item

**Calculate**

**Cost Estimate**

**Storage**

Cost per GB/month	0.250 USD
Total Data stored per region	x 10 GB
EST. STORAGE COST PER MONTH	2.50 USD

**Workload**

Cost per 100 RU/s per hour	0.008 USD
EST. THROUGHPUT AT PEAK Show Details	x 6520 RU/s
EST. THROUGHPUT OFF PEAK Show Details	x 3260 RU/s
EST. WORKLOAD COST/MONTH Show Details	209.45 USD

\* Saving 45% (USD 171.36) by programmatically lowering provisioned throughput during off peak hours. [Learn more](#).

**Number of regions**: x 1

**EST. TOTAL COST/MONTH**: 211.95 USD

**Save estimate**

**SAVE UP TO 65% WITH RESERVED CAPACITY**  
[See here for more details](#)

**YOU WILL SAVE UP TO 70% TCO WITH COSMOS**  
[Learn more about Cosmos TCO](#)

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  $100 \text{ query / sec} * 10 \text{ RU / query} = 1000 \text{ RU/s}$  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](#).

# Find the request unit charge in Azure Cosmos DB

2/24/2020 • 7 minutes to read • [Edit Online](#)

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. 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.

## SQL (Core) API

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

Currently, you can find the request charge in the Azure portal only for a SQL query.

1. Sign in to the [Azure portal](#).
2. [Create a new Azure Cosmos account](#) and feed it with data, or select an 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.

The screenshot shows the Azure portal's Data Explorer interface. On the left, there's a sidebar with 'flights' selected under 'departuredelays'. The main area has a 'New SQL Query' button at the top, which is highlighted with a red box. Below it is a 'Execute Query' button, also highlighted with a red box. A query is entered: 'SELECT \* FROM d WHERE d.origin = 'ABE''. At the bottom, the 'Results' tab is active, but the 'Query Stats' tab is highlighted with a red box. A table below shows the results:

METRIC	VALUE
Request Charge	46.18000000000001 RUs
Showing Results	1 - 100
Round Trips	1

### Use the .NET SDK

#### .Net V2 SDK

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;
}

```

## .Net V3 SDK

Objects that are returned from the .NET SDK v3 expose a `RequestCharge` property:

```

Container container = this.cosmosClient.GetContainer("database", "container");
string itemId = "myItem";
string partitionKey = "partitionKey";
string storedProcedureId = "storedProcedureId";
string queryText = "SELECT * FROM c";

ItemResponse<dynamic> itemResponse = await container.CreateItemAsync<dynamic>(
    item: new { id = itemId, pk = partitionKey },
    partitionKey: new PartitionKey(partitionKey));
double requestCharge = itemResponse.RequestCharge;

Scripts scripts = container.Scripts;
StoredProcedureExecuteResponse<object> sprocResponse = await scripts.ExecuteStoredProcedureAsync<object>(
    storedProcedureId: storedProcedureId,
    partitionKey: new PartitionKey(partitionKey),
    parameters: new dynamic[] { new object() });

requestCharge = sprocResponse.RequestCharge;

FeedIterator<dynamic> feedIterator = container.GetItemQueryIterator<dynamic>(
    queryText: queryText,
    requestOptions: new QueryRequestOptions() { PartitionKey = new PartitionKey(partitionKey) });
while (feedIterator.HasMoreResults)
{
    FeedResponse<dynamic> feedResponse = await feedIterator.ReadNextAsync();
    requestCharge = feedResponse.RequestCharge;
}

```

For more information, see [Quickstart: Build a .NET web app by using a SQL API account in Azure Cosmos DB.](#)

## 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 [Nodejs 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 Nodejs 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](#).

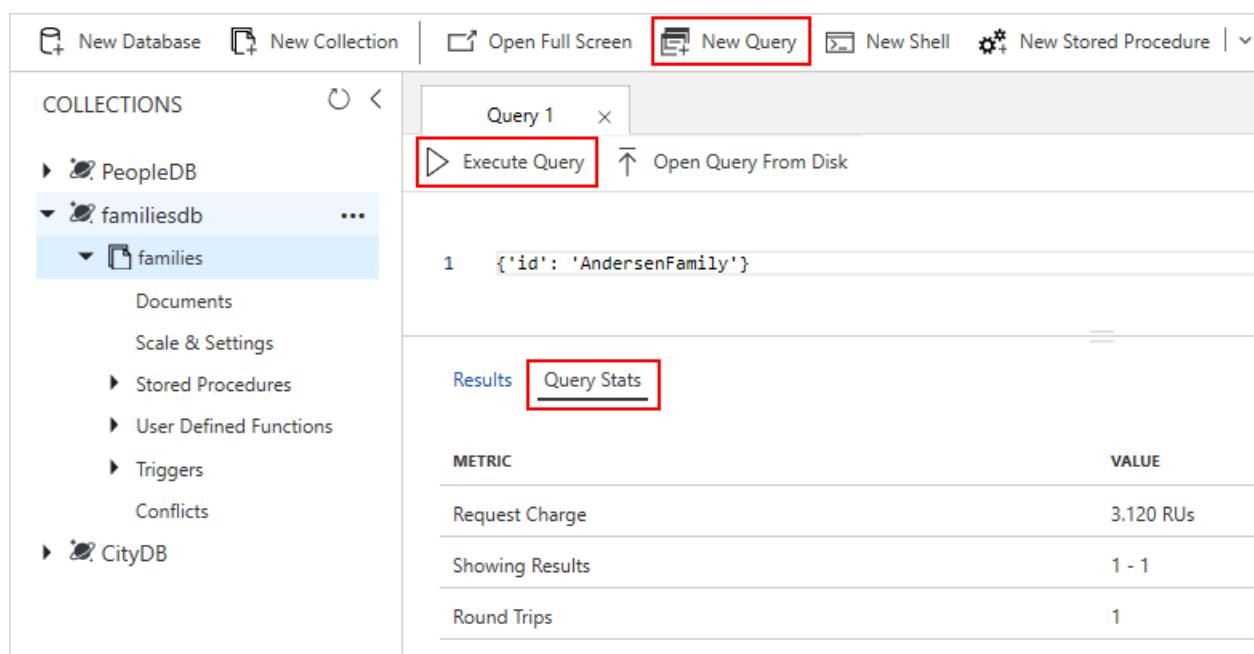
## Azure Cosmos DB API for MongoDB

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

Currently, you can find the request charge in the Azure portal only for a query.

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.



The screenshot shows the Azure portal's Data Explorer pane. On the left, the 'COLLECTIONS' sidebar lists databases like 'PeopleDB', 'familiesdb' (selected), and 'CityDB'. Under 'familiesdb', a 'families' collection is selected. The main area shows a query editor with 'Query 1' containing the command `1 { 'id': 'AndersenFamily' }`. Below the query, there are two tabs: 'Results' (selected) and 'Query Stats'. The 'Query Stats' tab displays metrics for the last query execution. The table shows:

METRIC	VALUE
Request Charge	3.120 RUs
Showing Results	1 - 1
Round Trips	1

## 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 Nodejs 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 Nodejs web app to Azure Cosmos DB](#).

## Cassandra API

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](#).

## Gremlin API

When you use the Gremlin API, you have multiple options for finding the RU consumption for an operation against an Azure Cosmos container.

### Use drivers and SDK

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.

### Use the .NET SDK

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

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](#).

## Table API

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
- Globally scale provisioned throughput
- Provision throughput on containers and databases
- Provision throughput for a container
- Monitor and debug with metrics in Azure Cosmos DB

# Scale Azure Cosmos DB throughput by using Azure Functions Timer trigger

1/13/2020 • 2 minutes to read • [Edit Online](#)

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 [Role-based Access Control](#) (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](#).

# Work with data using Azure Storage Explorer

10/24/2019 • 9 minutes to read • [Edit Online](#)

Using Azure Cosmos DB in Azure Storage Explorer enables users to manage Azure Cosmos DB entities, manipulate data, update stored procedures and triggers along with other Azure entities like Storage blobs and queues. Now you can use the same tool to manage your different Azure entities in one place. At this time, Azure Storage Explorer supports Cosmos accounts configured for SQL, MongoDB, Graph, and Table APIs.

## Prerequisites

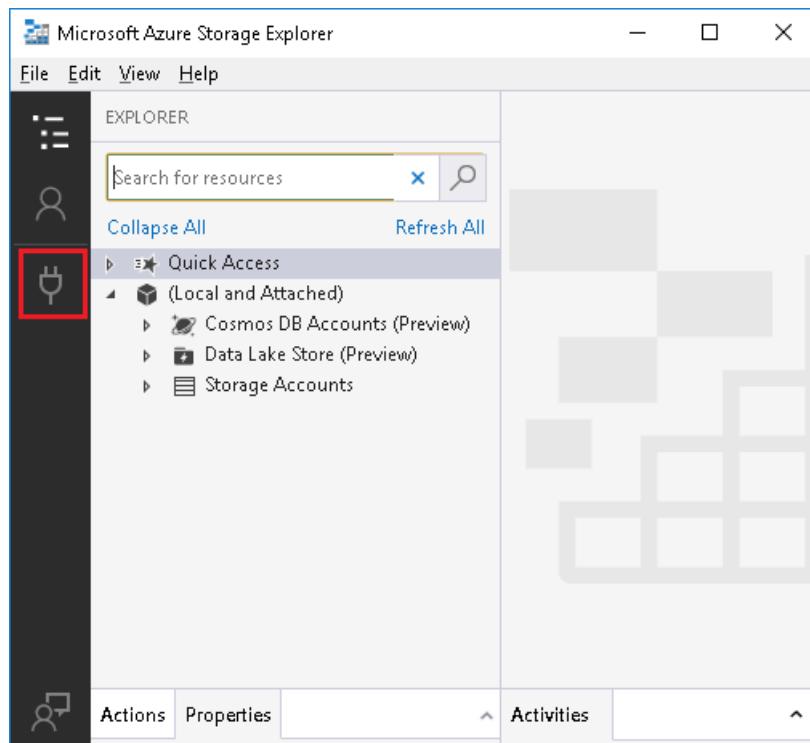
A Cosmos account with SQL API or Azure Cosmos DB's API for MongoDB. If you don't have an account, you can create one in the Azure portal, as described in [Azure Cosmos DB: Build a SQL API web app with .NET and the Azure portal](#).

## Installation

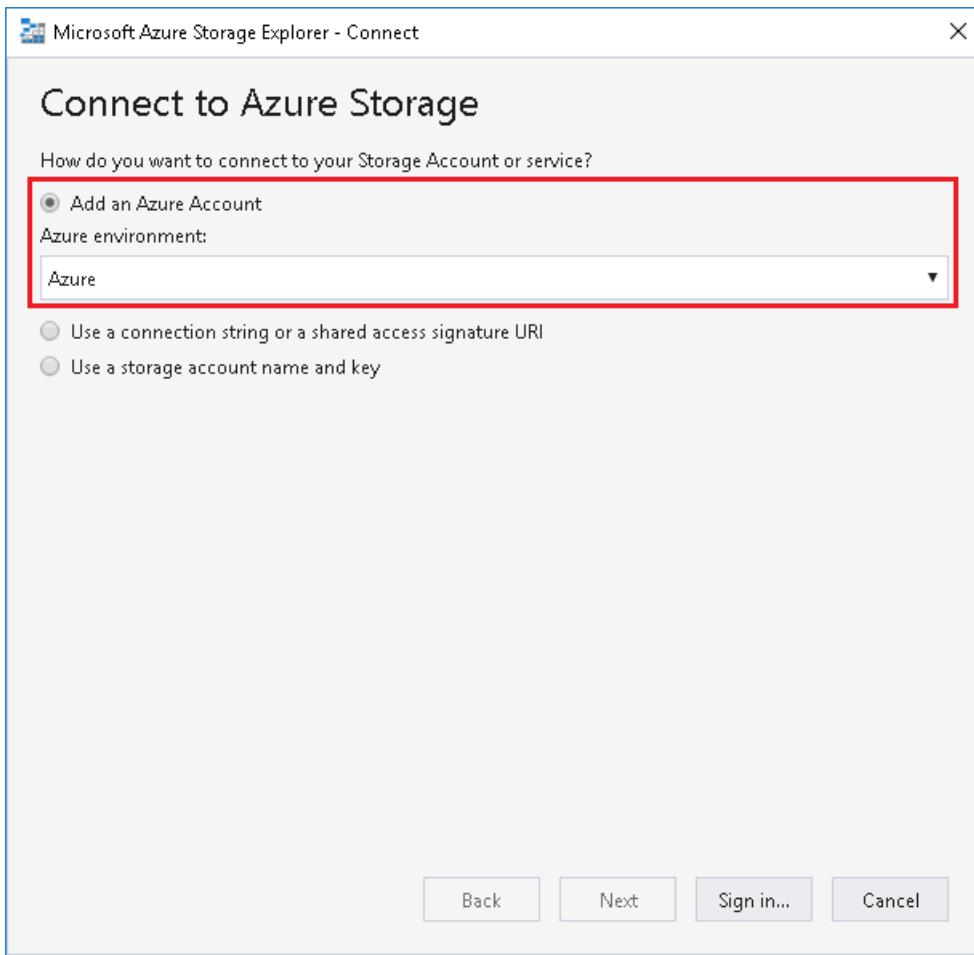
Install the newest Azure Storage Explorer bits here: [Azure Storage Explorer](#), now we support Windows, Linux, and MAC version.

## Connect to an Azure subscription

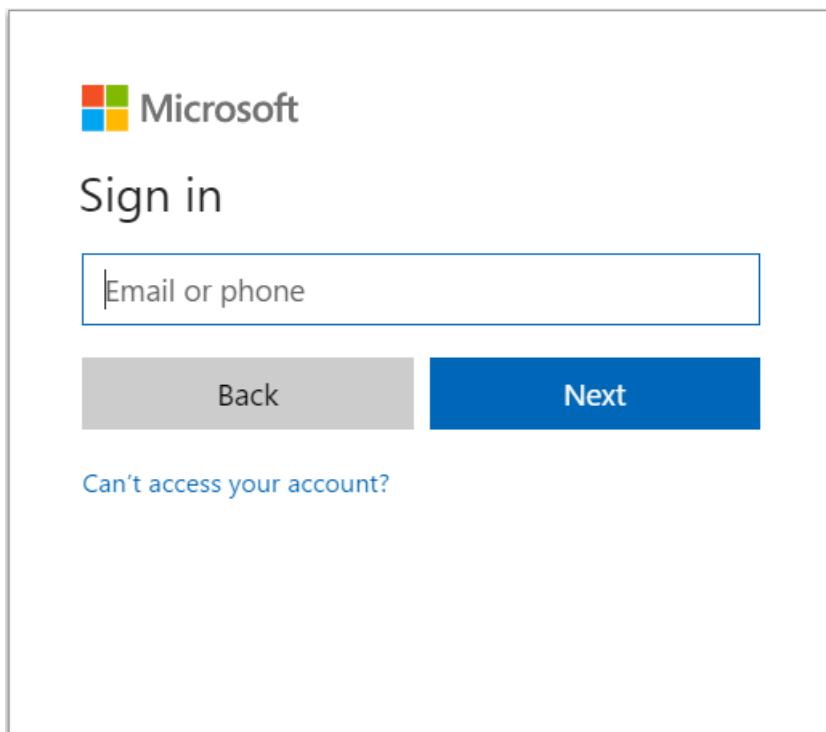
1. After installing the **Azure Storage Explorer**, click the **plug-in** icon on the left as shown in the following image:



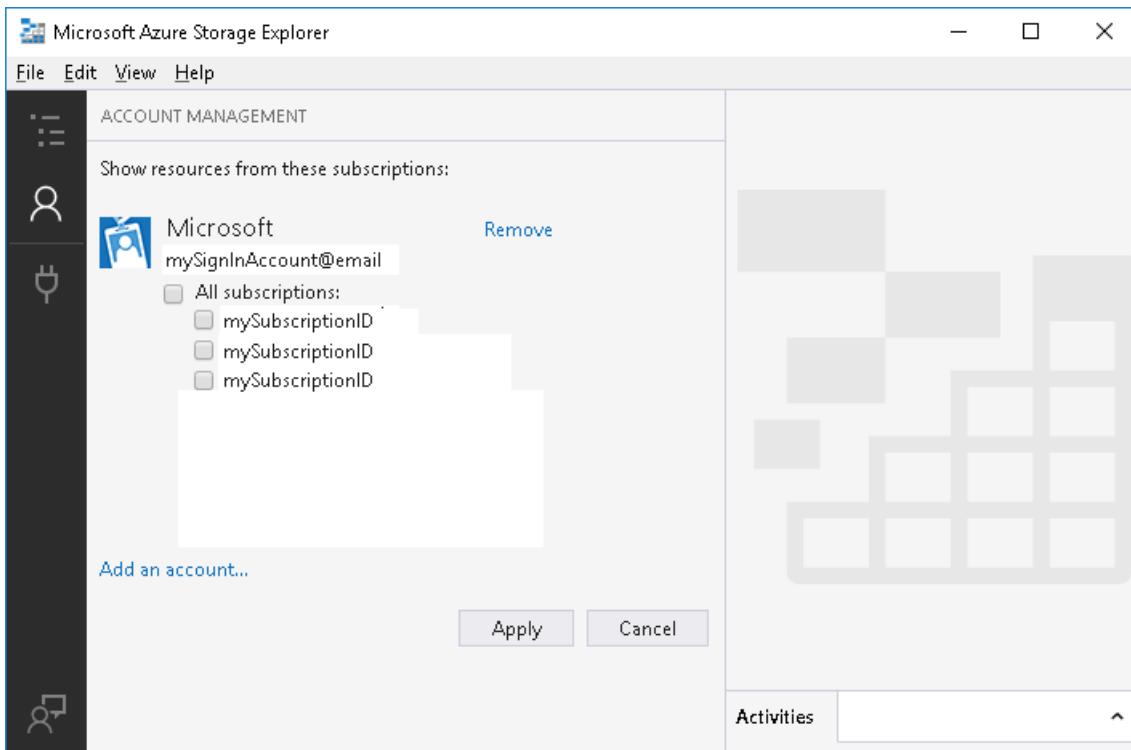
2. Select **Add an Azure Account**, and then click **Sign-in**.



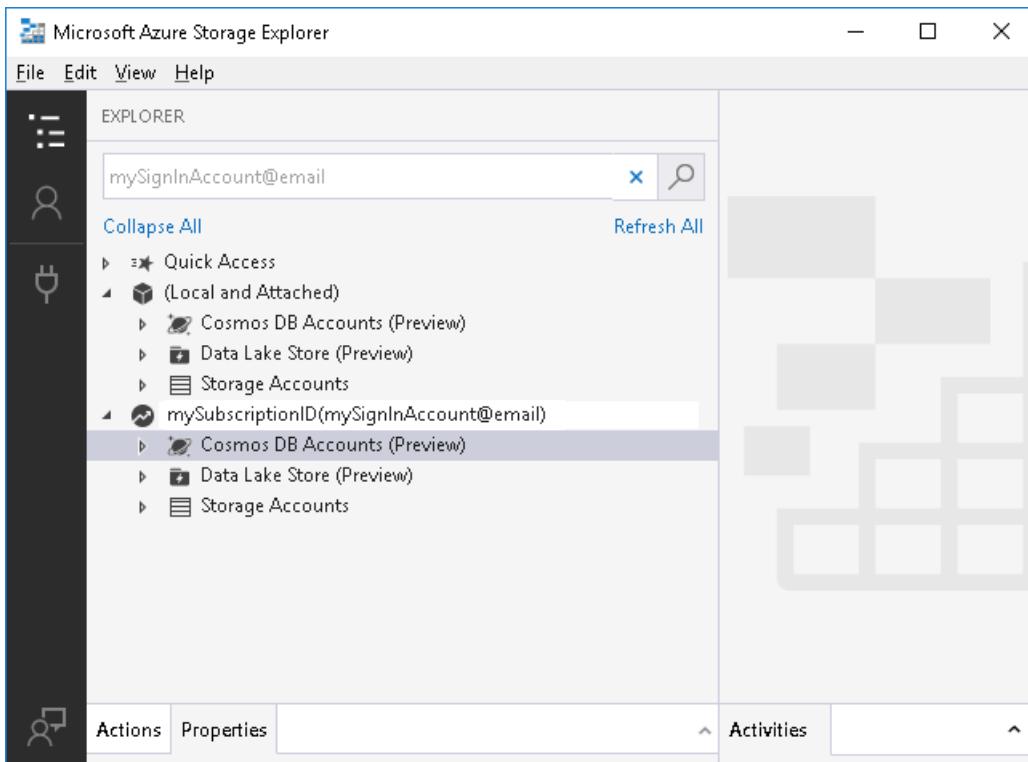
3. In the **Azure Sign in** dialog box, select **Sign in**, and then enter your Azure credentials.



4. Select your subscription from the list and then click **Apply**.



The Explorer pane updates and displays the accounts in the selected subscription.

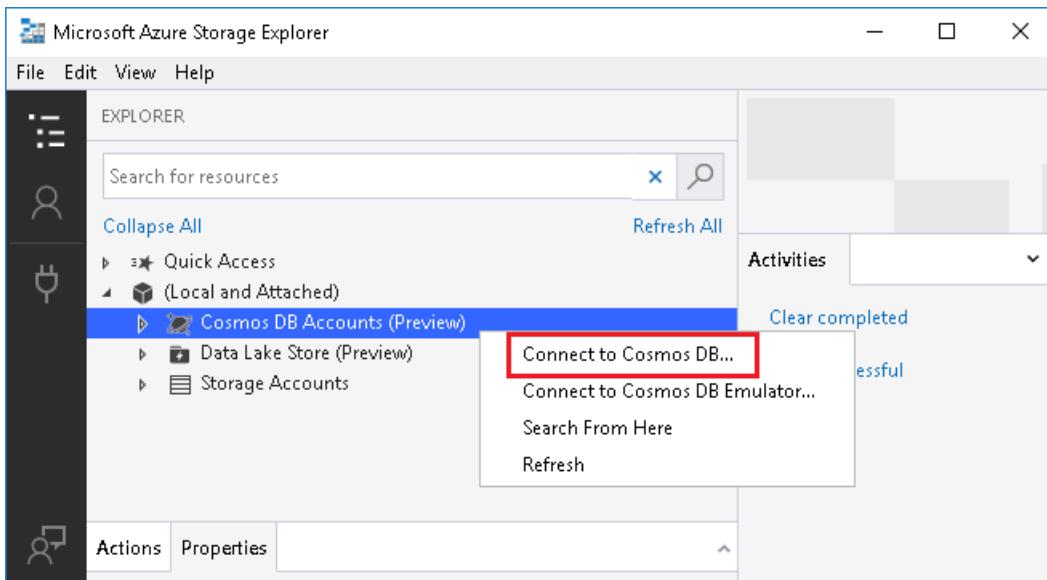


You have successfully connected to your **Cosmos DB account** to your Azure subscription.

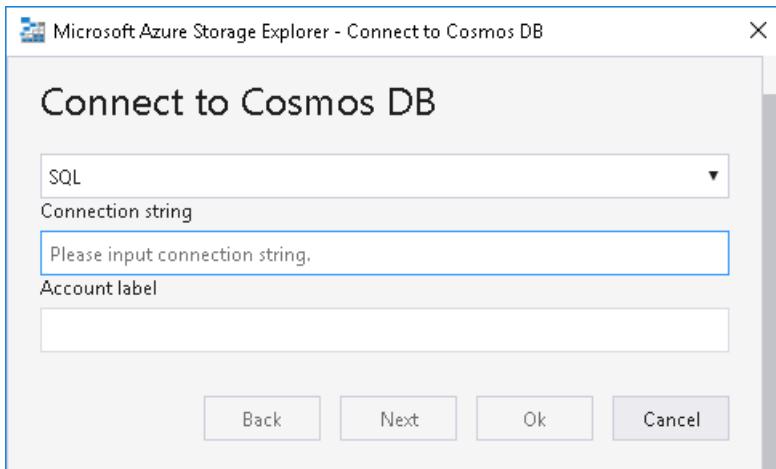
## Connect to Azure Cosmos DB by using a connection string

An alternative way of connecting to an Azure Cosmos DB is to use a connection string. Use the following steps to connect using a connection string.

1. Find **Local and Attached** in the left tree, right-click **Cosmos DB Accounts**, choose **Connect to Cosmos DB...**.



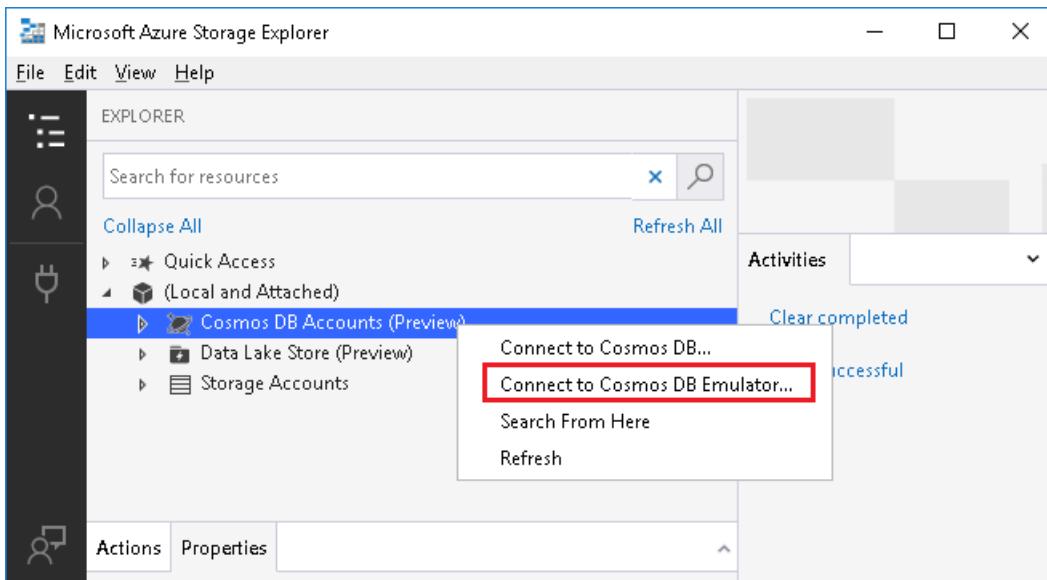
2. Only support SQL and Table API currently. Choose API, paste **Connection String**, input **Account label**, click **Next** to check the summary, and then click **Connect** to connect Azure Cosmos DB account. For information on retrieving the primary connection string, see [Get the connection string](#).



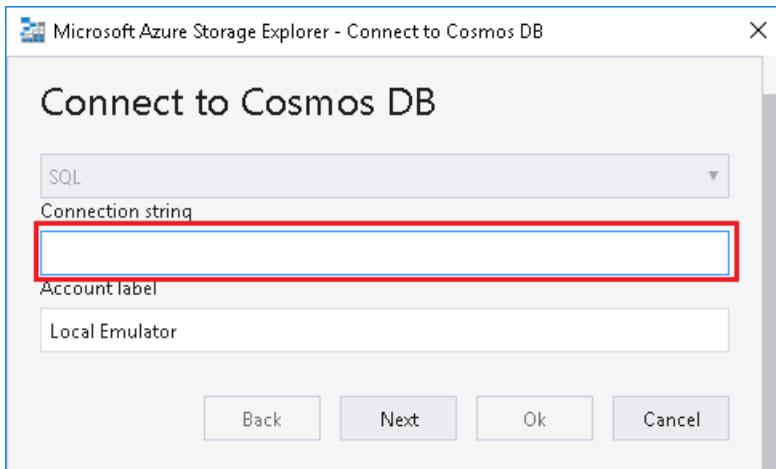
## Connect to Azure Cosmos DB by using local emulator

Use the following steps to connect to an Azure Cosmos DB by Emulator, only support SQL account currently.

1. Install Emulator and launch. For how to install Emulator, see [Cosmos DB Emulator](#)
2. Find **Local and Attached** in the left tree, right-click **Cosmos DB Accounts**, choose **Connect to Cosmos DB Emulator...**



3. Only support SQL API currently. Paste **Connection String**, input **Account label**, click **Next** to check the summary, and then click **Connect** to connect Azure Cosmos DB account. For information on retrieving the primary connection string, see [Get the connection string](#).



## Azure Cosmos DB resource management

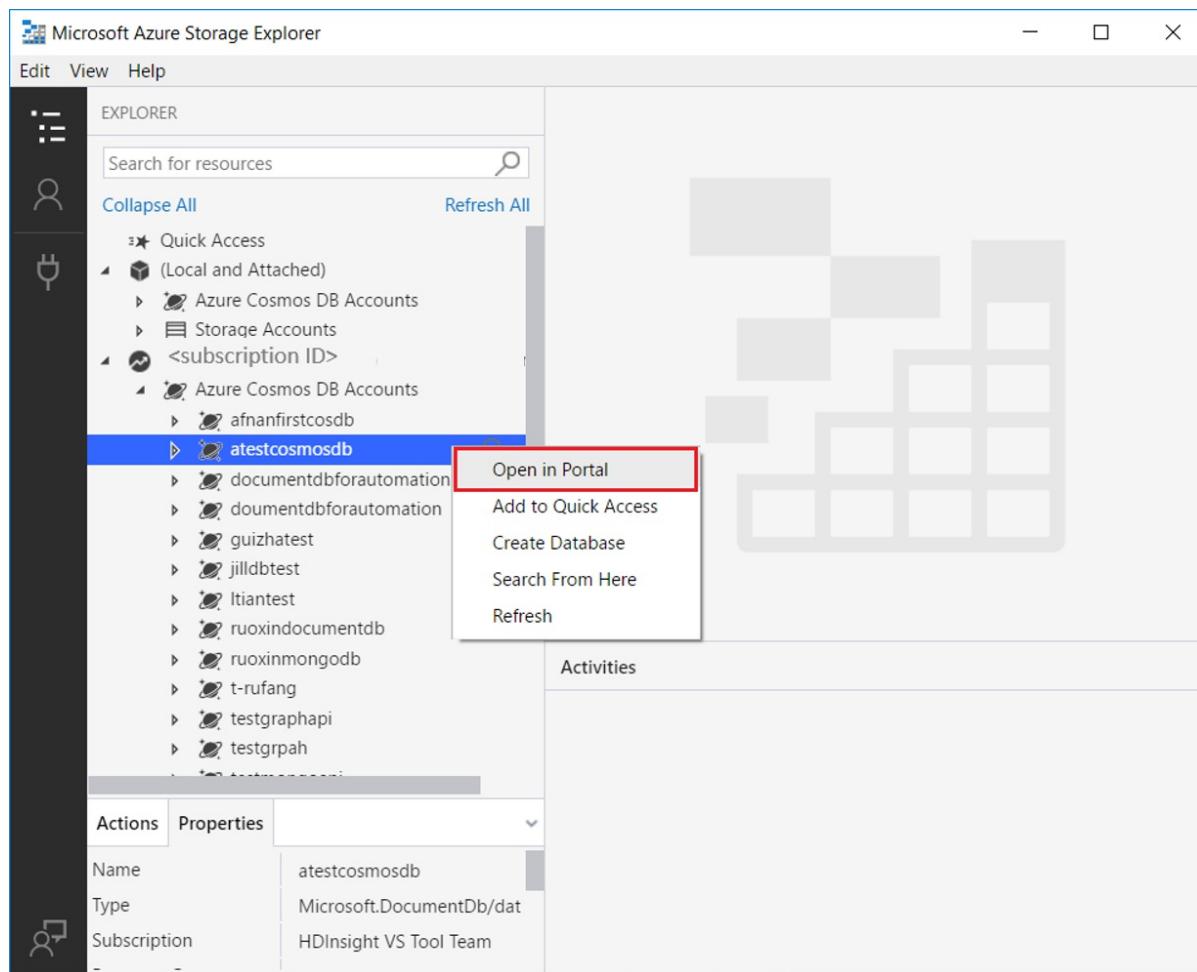
You can manage an Azure Cosmos DB account by doing following operations:

- 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

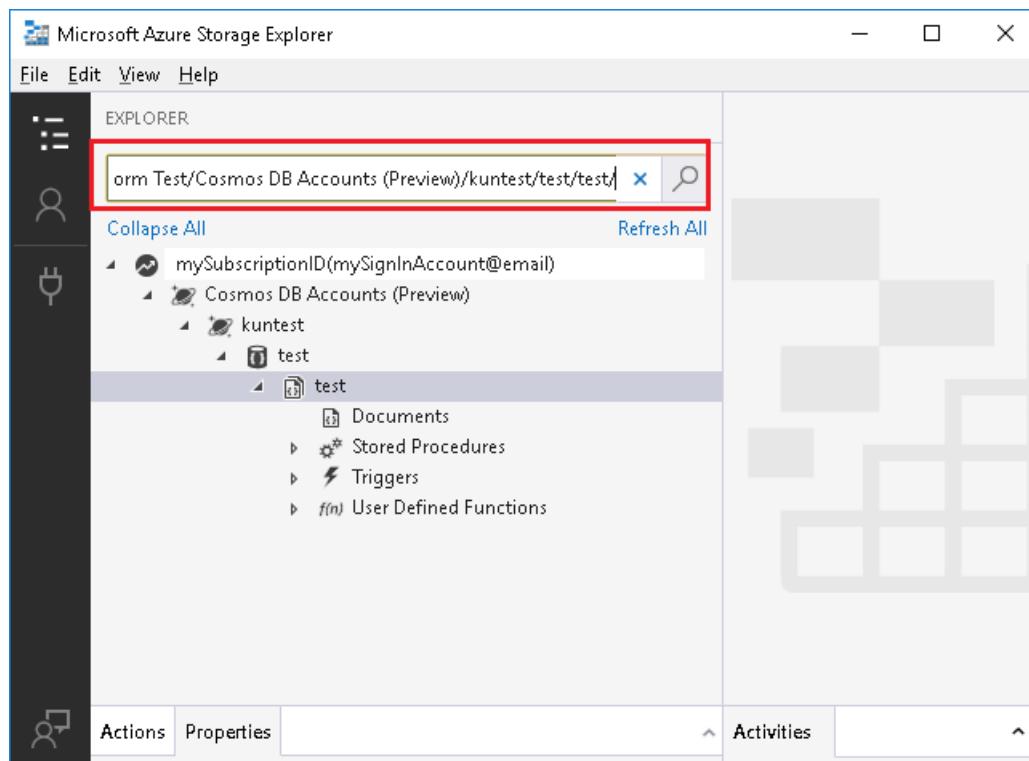
### Quick access tasks

By right-clicking on a subscription in the Explorer pane, you can perform many quick action tasks:

- Right-click an Azure Cosmos DB account or a database, you can choose **Open in Portal** and manage the resource in the browser on the Azure portal.



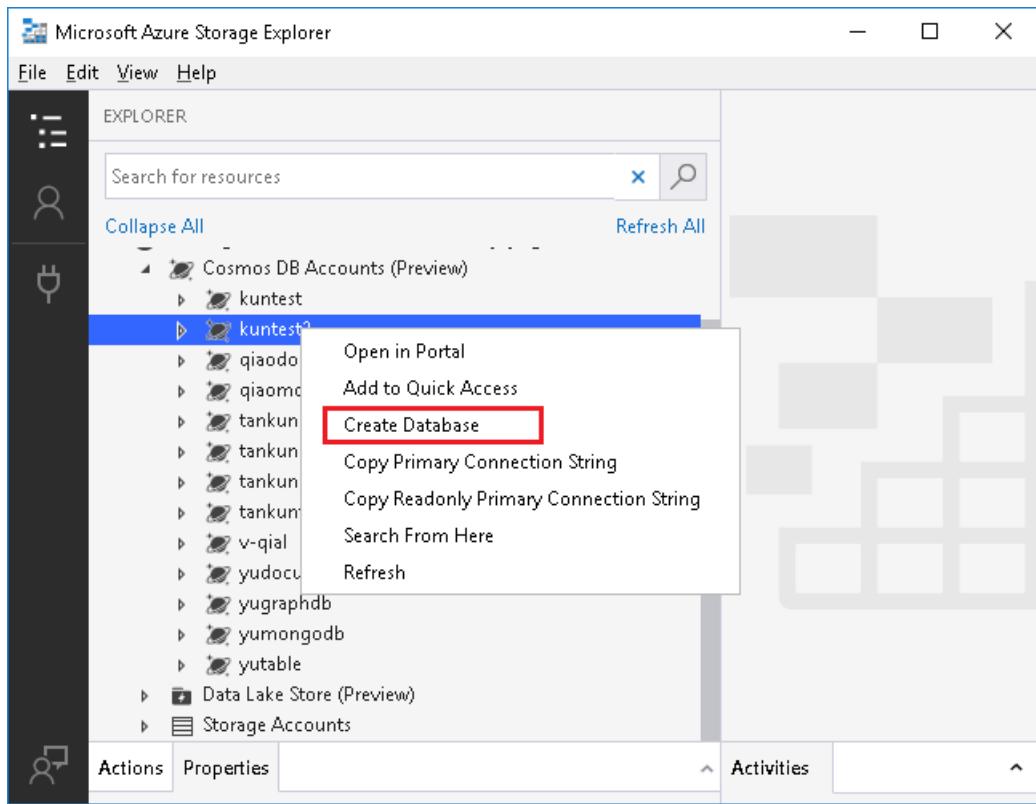
- You can also add Azure Cosmos DB account, database, collection to **Quick Access**.
- **Search from Here** enables keyword search under the selected path.



## Database and collection management

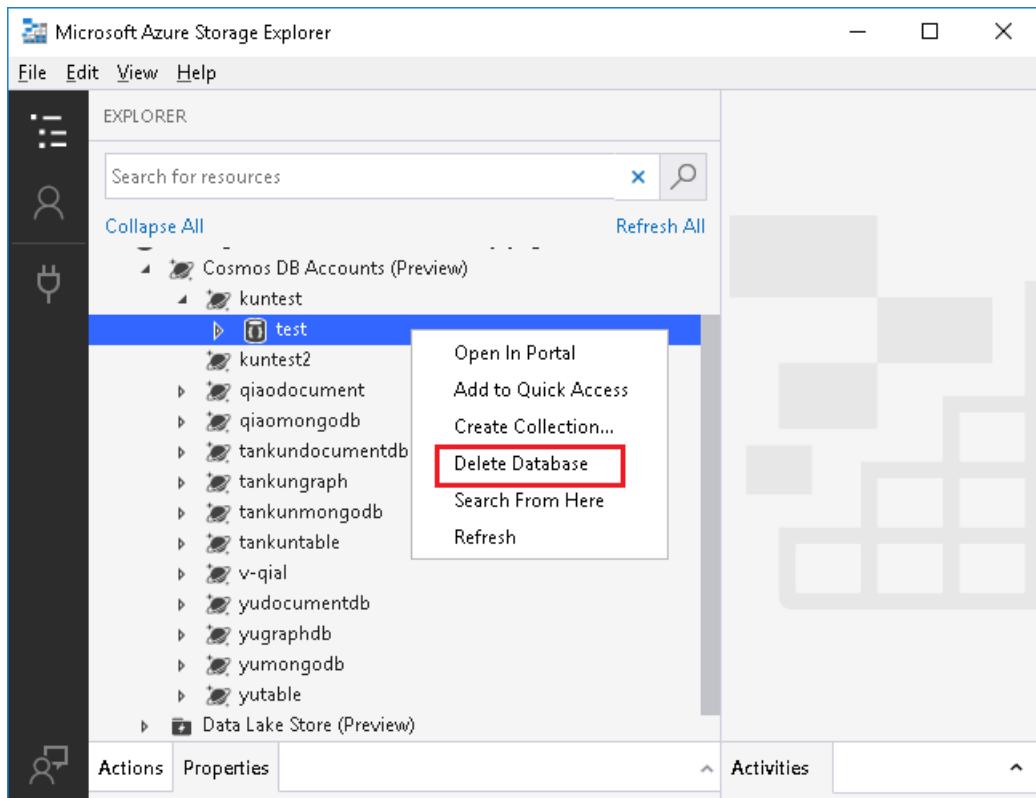
### Create a database

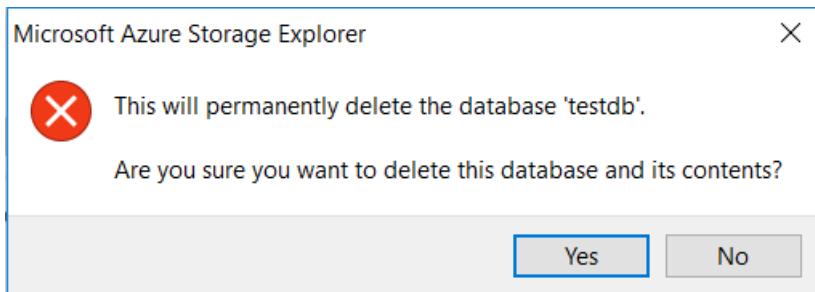
- Right-click the Azure Cosmos DB account, choose **Create Database**, input the database name, and press **Enter** to complete.



#### Delete a database

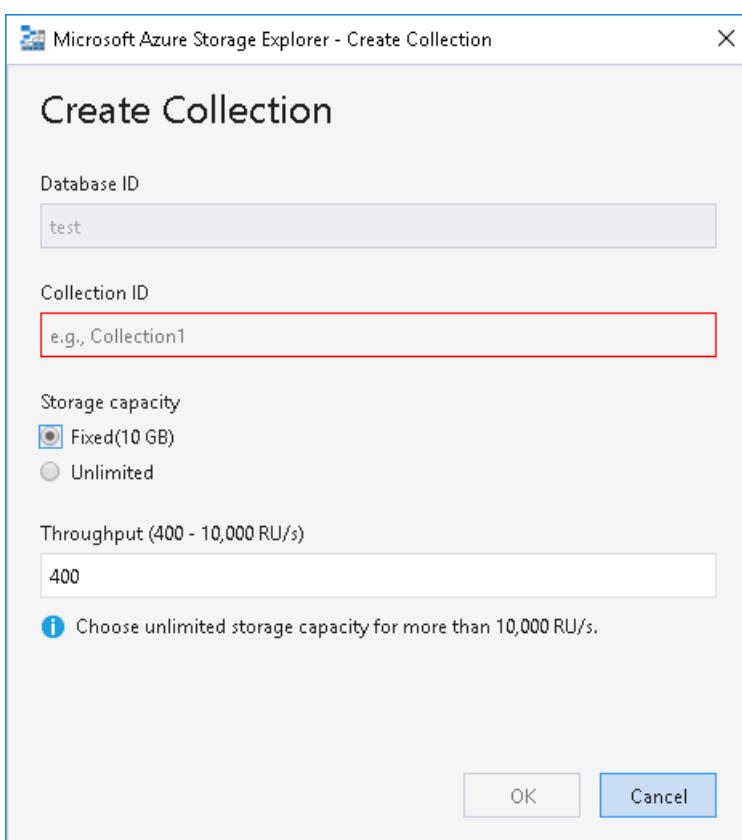
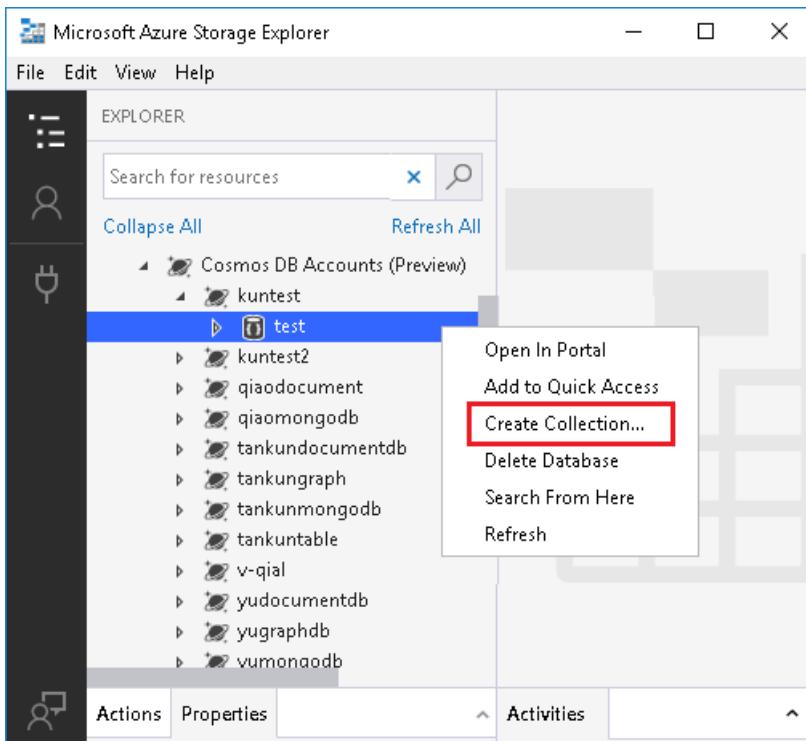
- Right-click the database, click **Delete Database**, and click **Yes** in the pop-up window. The database node is deleted, and the Azure Cosmos DB account refreshes automatically.





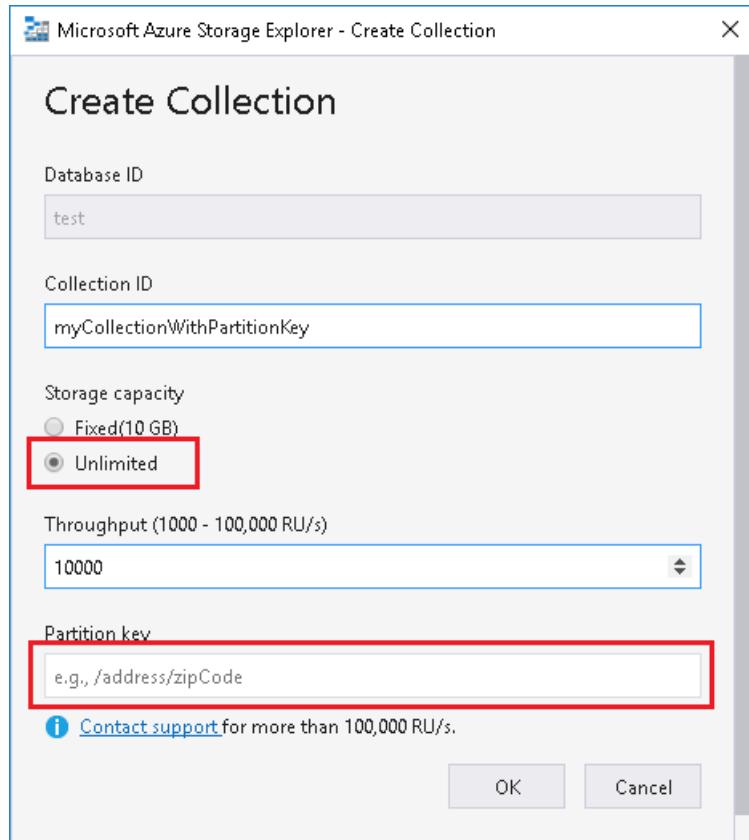
#### Create a collection

1. Right-click your database, choose **Create Collection**, and then provide the following information like **Collection ID, Storage capacity**, etc. Click **OK** to finish.



2. Select **Unlimited** to be able to specify partition key, then click **OK** to finish.

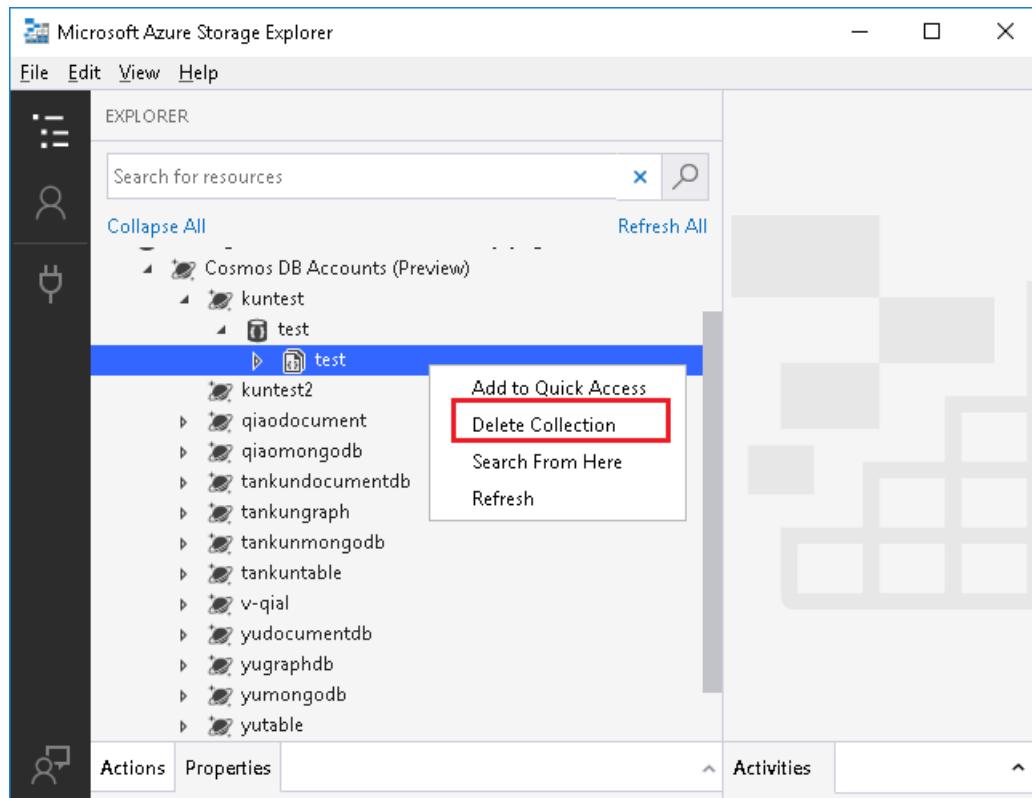
If a partition key is used when creating a collection, once creation is completed, the partition key value can't be changed on the collection.



#### Delete a collection

- Right-click the collection, click **Delete Collection**, and then click **Yes** in the pop-up window.

The collection node is deleted, and the database refreshes automatically.



#### Document management

## Create and modify documents

- To create a new document, open **Documents** in the left window, click **New Document**, edit the contents in the right pane, then click **Save**. You can also update an existing document, and then click **Save**. Changes can be discarded by clicking **Discard**.

The screenshot shows the Azure portal's 'collection1' blade. On the left, the 'EXPLORER' sidebar lists various Azure services like Quick Access, Local and Attached accounts, and Azure Cosmos DB Accounts (Preview). Under 'collection1', there are 'Documents', 'Stored Procedures', 'Triggers', and 'User Defined Functions'. A specific document with ID 66 is selected in the main pane. The top navigation bar includes 'New Document', 'Save', 'Discard', 'Delete', and 'Refresh' buttons.

## Delete a document

- Click the **Delete** button to delete the selected document.

## Query for documents

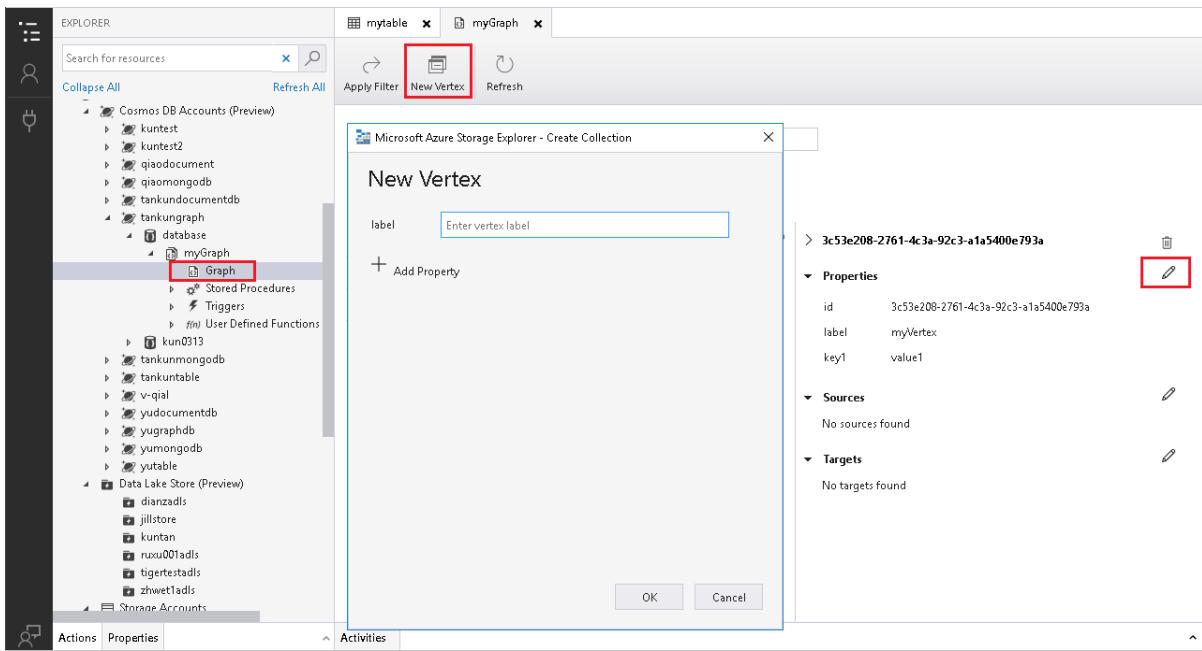
- Edit the document filter by entering a [SQL query](#) and then click **Apply**.

The screenshot shows the 'Collection1' blade. The 'Filter' and 'Apply' buttons are highlighted. In the query editor, the SQL query 'SELECT \* FROM c where c.id = "22"' is entered. The results table shows a single document with ID 22 selected. The top navigation bar includes 'New Document', 'Save', 'Discard', 'Delete', and 'Refresh' buttons.

## Graph management

### Create and modify vertex

- To create a new vertex, open **Graph** from the left window, click **New Vertex**, edit the contents, then click **OK**.
- To modify an existing vertex, click the pen icon in the right pane.

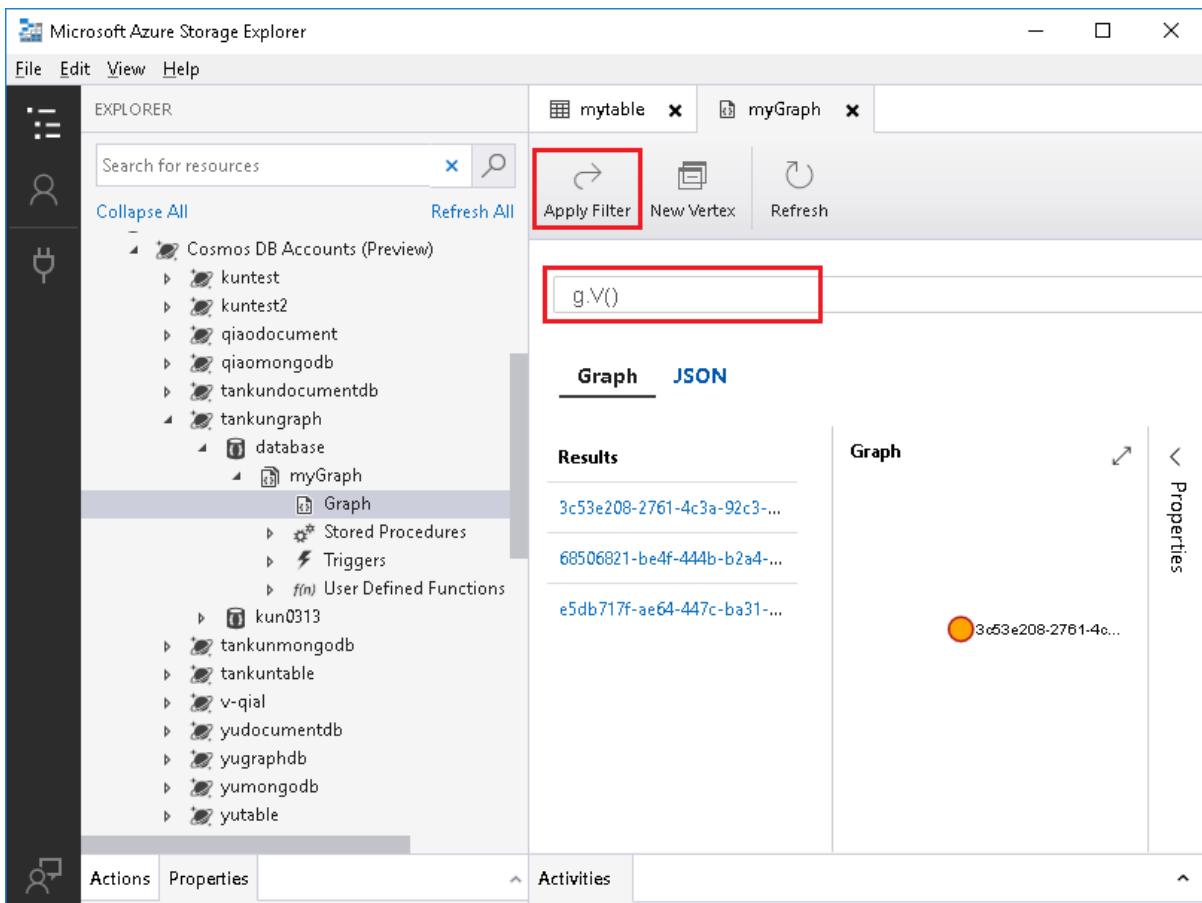


### Delete a graph

- To delete a vertex, click the recycle bin icon beside the vertex name.

### Filter for graph

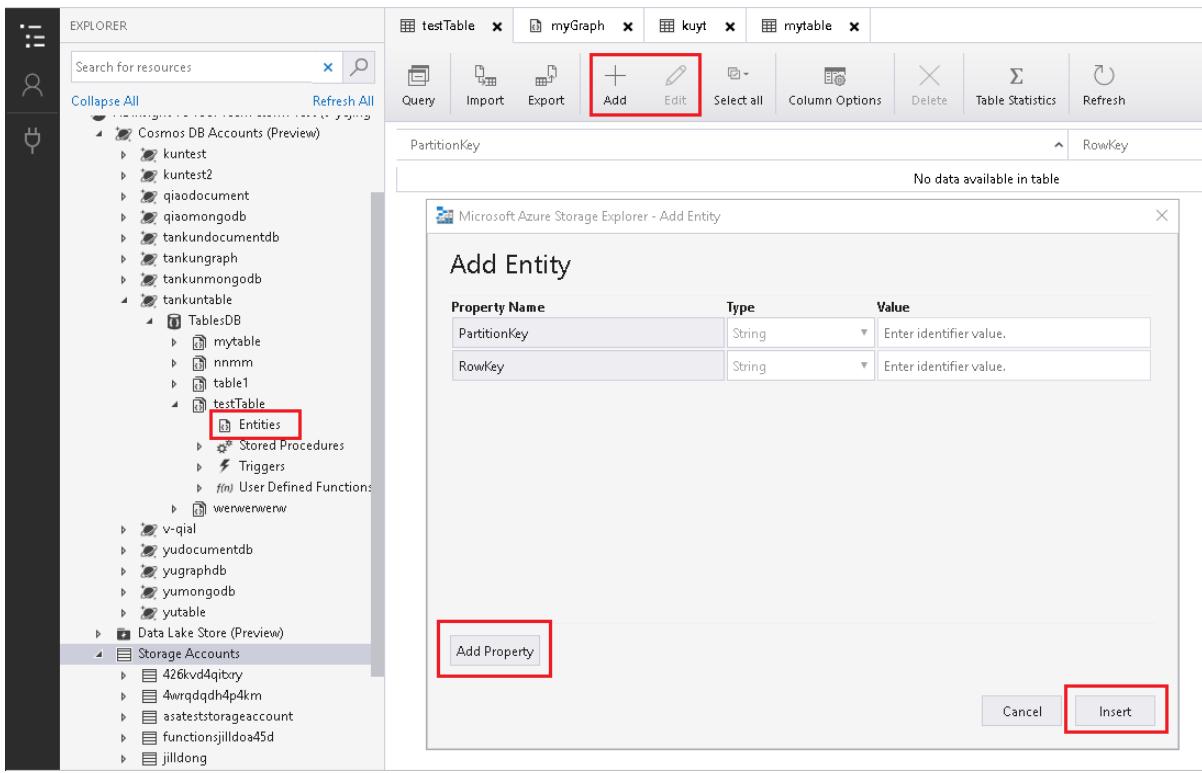
- Edit the graph filter by entering a [gremlin query](#) and then click **Apply Filter**.



## Table management

### Create and modify table

- To create a new table, open **Entities** from the left window, click **Add**, edit the content in **Add Entity** dialog, add property by clicking button **Add Property**, then click **Insert**.
- To modify a table, click **Edit**, modify the content, then click **Update**.



#### Import and export table

1. To import, click **Import** button and choose an existing table.
2. To export, click **Export** button and choose a destination.

PartitionKey	RowKey	Capacity	ContainerCount	ObjectCount	Timestamp	xcsd
20180106T0000	data	4641645650	5	5789	2018-03-15T08:44:25.239Z	vfs
20180107T0000	data	4641645650	5	5789	2018-03-15T09:34:45.015Z	
20180109T0000	data	4641645650	5	5789	2018-03-15T03:36:30.873Z	
20180110T0000	data	4641645650	5	5789	2018-03-15T03:34:45.015Z	
20180111T0000		4662618517	5	5796	2018-03-15T03:34:45.034Z	

#### Delete entities

- Select the entities and click button **Delete**.

The screenshot shows the Microsoft Azure Storage Explorer interface. On the left, the 'EXPLORER' pane displays a tree view of Cosmos DB accounts and tables. In the center, the 'mytable' table is selected, showing a list of entities with columns: PartitionKey, RowKey, Capacity, ContainerCount, ObjectCount, Timestamp, and xcsds. A specific entity is selected, and the 'Delete' button in the toolbar is highlighted with a red box. A confirmation dialog box titled 'Microsoft Azure Storage Explorer' asks 'Are you sure you want to delete selected entities?' with 'Yes' and 'No' buttons. The status bar at the bottom indicates 'Showing 1 to 5 of 5 cached items'.

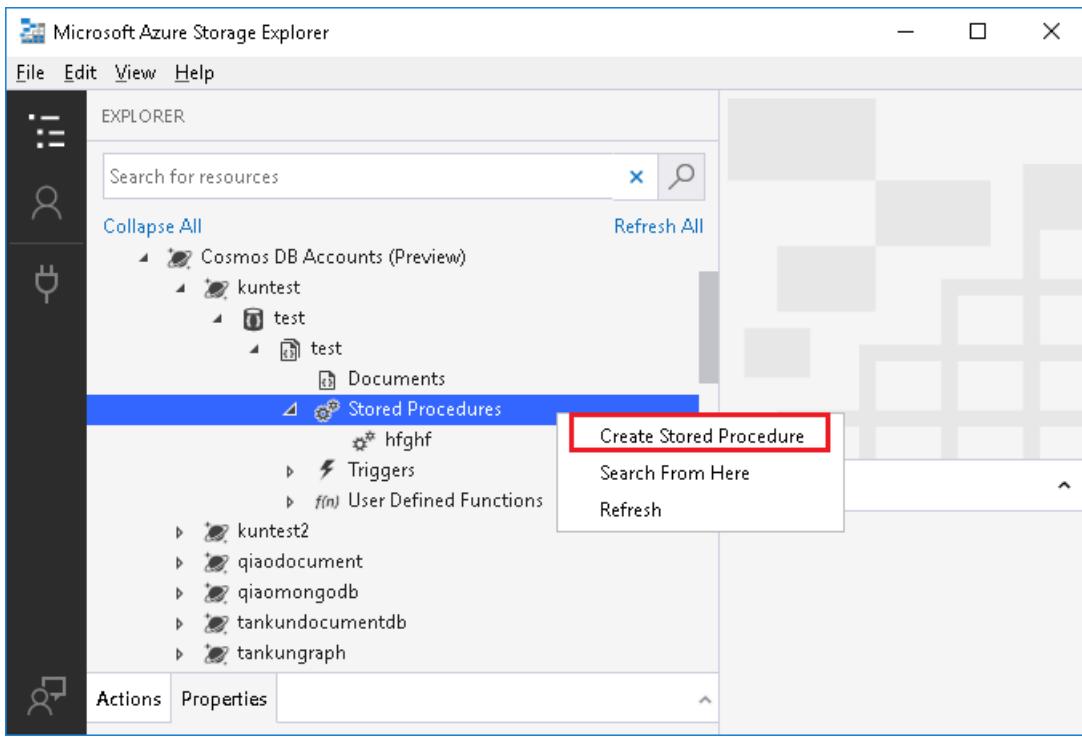
### Query table

- Click **Query** button, input query condition, then click **Execute Query** button. Close Query pane by clicking **Close Query** button.

The screenshot shows the Microsoft Azure Storage Explorer interface with the 'testTable' table selected. The 'Query' pane is open, showing a WHERE clause with conditions: PartitionKey = '20180107T0000' AND RowKey = 'data'. The 'Execute' button in the Query pane is highlighted with a yellow box. Below the pane, the table data is displayed with one row: PartitionKey '20180107T0000', RowKey 'data', Capacity '4641645650', ContainerCount '5', ObjectCount '5789', and Timestamp '2018-03-15T03:36:30.034Z'.

### Manage stored procedures, triggers, and UDFs

- To create a stored procedure, in the left tree, right-click **Stored Procedure**, choose **Create Stored Procedure**, enter a name in the left, type the stored procedure scripts in the right window, and then click **Create**.
- You can also edit existing stored procedures by double-clicking, making the update, and then clicking **Update** to save, or click **Discard** to cancel the change.



- The operations for **Triggers** and **UDF** are similar with **Stored Procedures**.

## Troubleshooting

Azure Cosmos DB in Azure Storage Explorer is a standalone app that allows you to connect to Azure Cosmos DB accounts hosted on Azure and Sovereign Clouds from Windows, macOS, or Linux. It enables you to manage Azure Cosmos DB entities, manipulate data, update stored procedures and triggers along with other Azure entities like Storage blobs and queues.

These are solutions for common issues seen for Azure Cosmos DB in Storage Explorer.

### Sign in issues

Before proceeding further, try restarting your application and see if the problems can be fixed.

#### Self-signed certificate in certificate chain

There are a few reasons you may be seeing this error, the two most common ones are:

- You're behind a *transparent proxy*, which means someone (such as your IT department) is intercepting HTTPS traffic, decrypting it, and then encrypting it using a self-signed certificate.
- You're running software, such as anti-virus software, which is injecting a self-signed SSL certificates into the HTTPS messages you receive.

When Storage Explorer encounters one of these "self-signed certificates", it can no longer know if the HTTPS message it's receiving has been tampered with. If you have a copy of the self-signed certificate though, then you can tell Storage Explorer to trust it. If you're unsure of who is injecting the certificate, then you can try to find it yourself by doing the following steps:

1. Install Open SSL
  - [Windows](#) (any of the light versions is ok)
  - Mac and Linux: Should be included with your operating system
2. Run Open SSL
  - Windows: Go to the install directory, then **/bin/**, then double-click on **openssl.exe**.
  - Mac and Linux: execute **openssl** from a terminal
3. Execute `s_client -showcerts -connect microsoft.com:443`

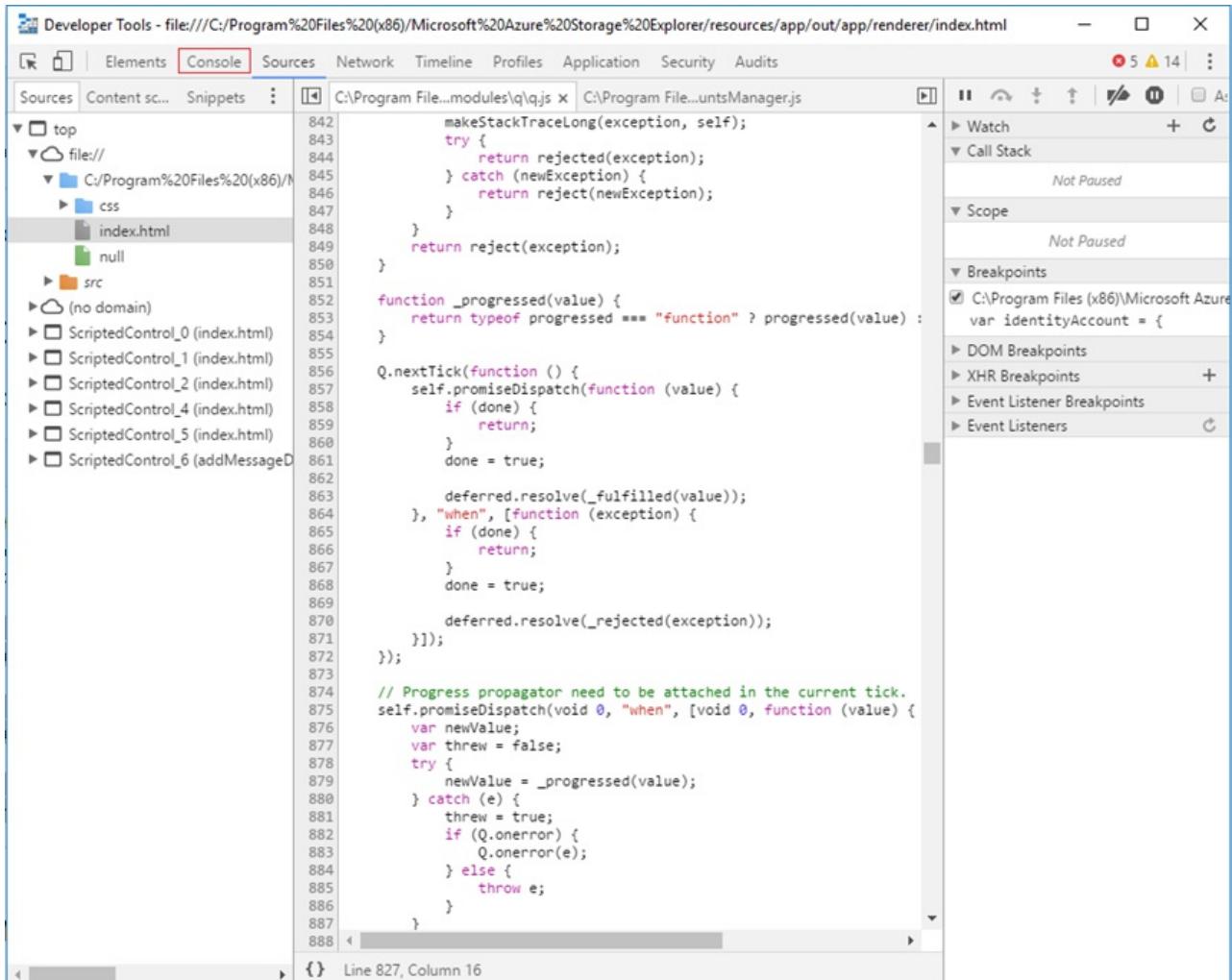
4. Look for self-signed certificates. If you're unsure, which are self-signed, then look for anywhere the subject ("s:") and issuer ("i:") are the same.
  5. Once you have found any self-signed certificates, copy and paste everything from and including -----**BEGIN CERTIFICATE**----- to -----**END CERTIFICATE**----- to a new .cer file for each one.
  6. Open Storage Explorer and then go to **Edit > SSL Certificates > Import Certificates**. Using the file picker, find, select, and open the .cer files you created.

If you're unable to find any self-signed certificates using the above steps, could send feedback for more help.

## Unable to retrieve subscriptions

If you're unable to retrieve your subscriptions after you successfully signed in:

- Verify your account has access to the subscriptions by signing into the [Azure Portal](#)
  - Make sure you have signed in using the correct environment ([Azure](#), [Azure China](#), [Azure Germany](#), [Azure US Government](#), or Custom Environment/Azure Stack)
  - If you're behind a proxy, make sure that you have configured the Storage Explorer proxy properly
  - Try removing and readding the account
  - Try deleting the following files from your home directory (such as: C:\Users\ContosoUser), and then readding the account:
    - .adalcache
    - .devaccounts
    - .extaccounts
  - Watch the developer tools console (f12) while signing in for any error messages



## Unable to see the authentication page

If you're unable to see the authentication page:

- Depending on the speed of your connection, it may take a while for the sign-in page to load, wait at least one minute before closing the authentication dialog
- If you're behind a proxy, make sure that you have configured the Storage Explorer proxy properly
- Bring up the developer console by pressing F12 key. Watch the responses from developer console and see if you can find any clue for why authentication is not working

#### Cannot remove account

If you're unable to remove an account, or if the reauthenticate link does not do anything

- Try deleting the following files from your home directory, and then readding the account:
  - .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 Mac
  - ~/config/StorageExplorer for Linux
  - **You will have to reenter all your credentials** if you delete these files

#### Http/Https proxy issue

You cannot list Azure Cosmos DB nodes in left tree when configuring http/https proxy in ASE. It's a known issue, and will be fixed in next release. You could use Azure Cosmos DB data explorer in Azure portal as a work-around at this moment.

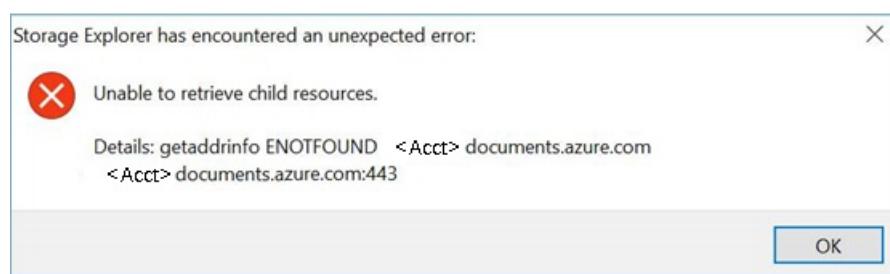
#### "Development" node under "Local and Attached" node issue

There is no response after clicking the "Development" node under "Local and Attached" node in left tree. The behavior is expected. Azure Cosmos DB local emulator will be supported in next release.



#### Attaching Azure Cosmos DB account in "Local and Attached" node error

If you see below error after attaching Azure Cosmos DB account in "Local and Attached" node, then check if you're using the right connection string.



#### Expand Azure Cosmos DB node error

You may see below error while trying to expand the tree nodes in left.



Try the following suggestions:

- Check if the Azure Cosmos DB account is in provision progress and try again when the account is being created successfully.
- If the account is under "Quick Access" node or "Local and Attached" nodes, then check if the account has been deleted. If so, you need to remove the node manually.

## Contact us

If none of the solutions work for you, send email to Azure Cosmos DB Dev Tooling Team ([cosmosdbtooling@microsoft.com](mailto:cosmosdbtooling@microsoft.com)) with details about the issue, for fixing the issues.

## Next steps

- Watch the following 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 explorer

5/24/2019 • 2 minutes to read • [Edit Online](#)

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.

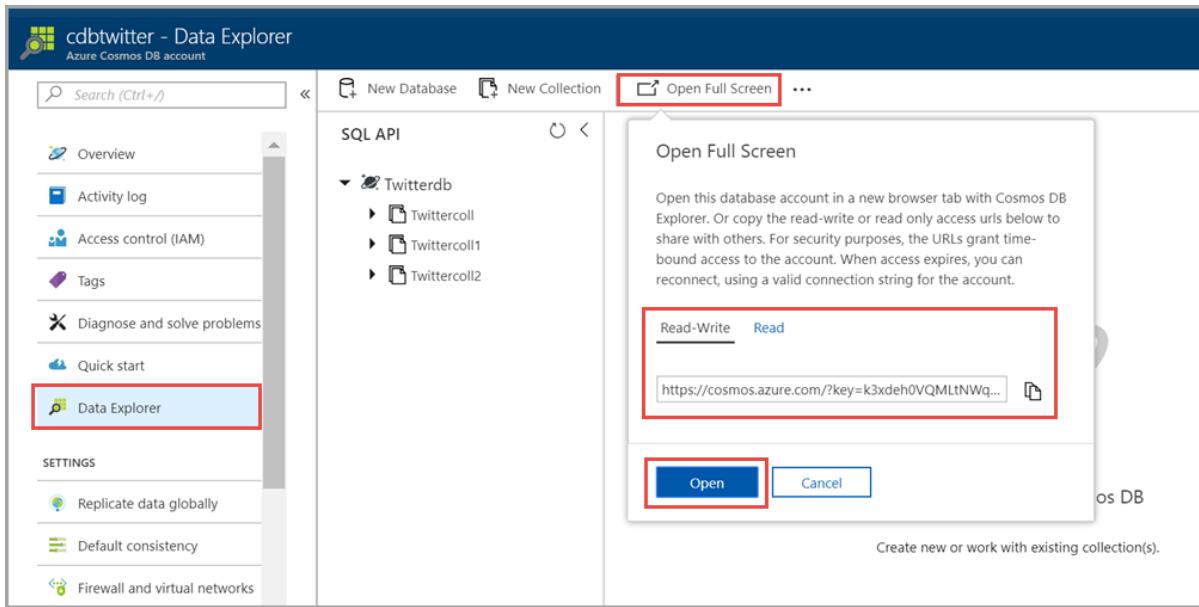
If you want to provide other users temporary access to your Azure Cosmos DB account, you can do so by using the read-write and read access URLs.

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.

## 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.

# Create an Azure Cosmos container

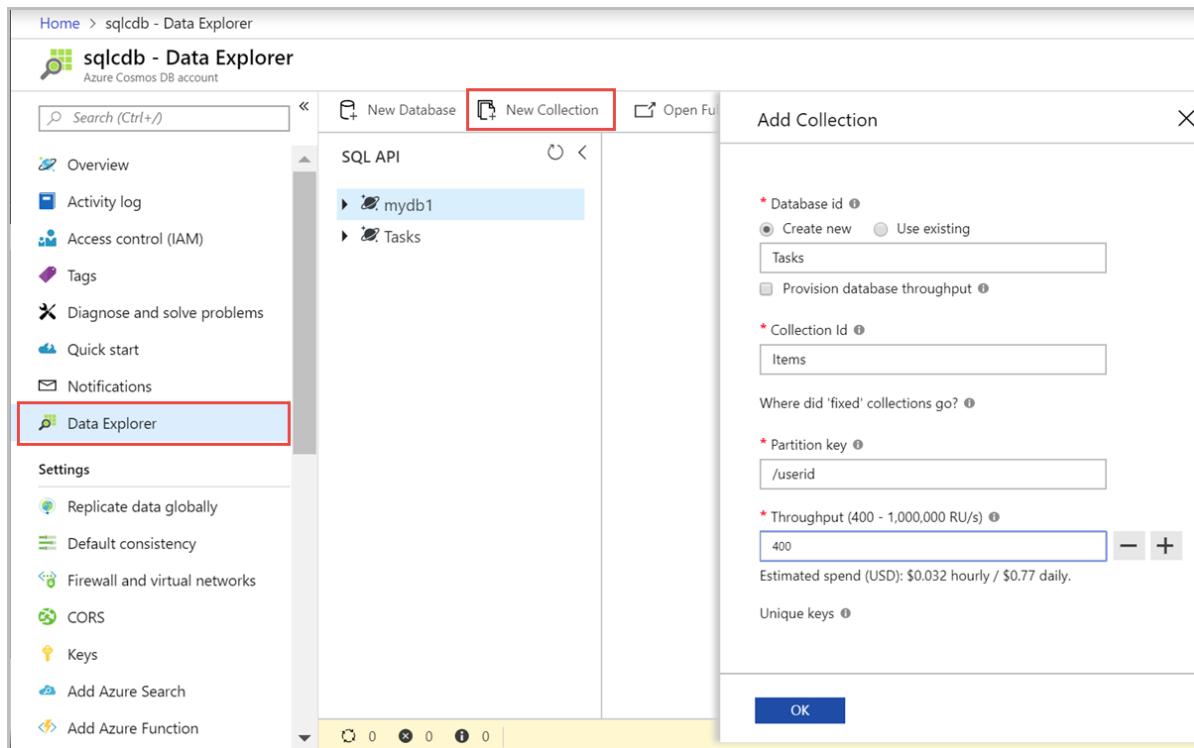
12/5/2019 • 3 minutes to read • [Edit Online](#)

This article explains the different ways to create an Azure Cosmos container (collection, table, or graph). You can use Azure portal, Azure CLI, or supported SDKs for this. This article demonstrates how to create a container, specify the partition key, and provision throughput.

## Create a container using Azure portal

### SQL API

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**.



### Azure Cosmos DB API for MongoDB

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 shard key.
- Enter a throughput to be provisioned (for example, 1000 RUs).
- Select **OK**.

Add Collection X

---

\* Database id i  
 Create new  Use existing

Provision database throughput i

\* Collection Id i

\* Storage capacity i  
 Fixed (10 GB)  Unlimited

\* Shard key i

\* Throughput (1,000 - 1,000,000 RU/s) i  
 - +

Unique keys i  
 [Delete]

+ Add unique key

## Cassandra API

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 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**.

Add Table X

\* Keyspace name i

\* Enter CQL command to create the table. [Learn More](#)  
CREATE TABLE MyKeySpace.   
(userid int, name text, email text, PRIMARY KEY  
(userid))

\* Throughput (1,000 - 1,000,000 RU/s) i  
 - +

Contact support for more than 1,000,000 RU/s.

**NOTE**

For Cassandra API, the primary key is used as the partition key.

**Gremlin API**

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 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**.

Add Graph

\* Database id ⓘ  
 Create new  Use existing  
MyDatabase

Provision database throughput ⓘ

\* Graph Id ⓘ  
MyGraph

\* Storage capacity ⓘ  
Fixed (10 GB) **Unlimited**

\* Partition key ⓘ  
/MyPartitionKey

\* Throughput (1,000 - 1,000,000 RU/s) ⓘ  
10000 **-** **+**

...

## Table API

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 Table**. Next, provide the following details:
  - Enter a Table ID.
  - Enter a throughput to be provisioned (for example, 1000 RUs).
  - Select **OK**.

Add Table

\* Table Id ⓘ  
MyTable

\* Storage capacity ⓘ  
Fixed (10 GB) **Unlimited**

\* Throughput (1,000 - 1,000,000 RU/s) ⓘ  
10000 **-** **+**

...

### NOTE

For Table API, the partition key is specified each time you add a new row.

## Create a container using Azure CLI

The links below show how to create container resources for Azure Cosmos DB using Azure CLI.

For a listing of all Azure CLI samples across all Azure Cosmos DB APIs see, [SQL API](#), [Cassandra API](#), [MongoDB](#)

[API](#), [Gremlin API](#), and [Table API](#)

- [Create a container with Azure CLI](#)
- [Create a collection for Azure Cosmos DB for MongoDB API with Azure CLI](#)
- [Create a Cassandra table with Azure CLI](#)
- [Create a Gremlin graph with Azure CLI](#)
- [Create a Table API table with Azure CLI](#)

## Create a container using PowerShell

The links below show how to create container resources for Azure Cosmos DB using PowerShell.

For a listing of all Azure CLI samples across all Azure Cosmos DB APIs see, [SQL API](#), [Cassandra API](#), [MongoDB API](#), [Gremlin API](#), and [Table API](#)

- [Create a container with Powershell](#)
- [Create a collection for Azure Cosmos DB for MongoDB API with Powershell](#)
- [Create a Cassandra table with Powershell](#)
- [Create a Gremlin graph with Powershell](#)
- [Create a Table API table with Powershell](#)

## Create a container using .NET SDK

### **SQL API and Gremlin API**

```
// 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 });
```

### **Azure Cosmos DB API for MongoDB**

```
// 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.

### **Cassandra API**

```
// 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);
```

## 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

1/10/2020 • 2 minutes to read • [Edit Online](#)

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.

New Container

**Start at \$24/mo per database, multiple containers included** [Learn more](#)

\* Database id [i](#)

Create new  Use existing

Type a new database id

Provision database throughput [i](#)

\* Container id [i](#)

e.g., Container1

\* Partition key [i](#)

e.g., /address/zipCode

My partition key is larger than 100 bytes

Old SDKs do not work with containers that support large partition keys, ensure you are using the right [SDK version](#). [Learn more](#)

\* Throughput (400-10,000 RU/s)

1000 [+](#) [-](#)

Estimated spend (USD): \$0.080 hourly / \$1.92 daily.

Choose unlimited storage capacity for more than 10,000 RU/s.

Unique keys [i](#)

+ Add unique key

## 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`.

### v3 .NET SDK

```
await database.CreateContainerAsync(
    new ContainerProperties(collectionName, "/longpartitionkey")
{
    PartitionKeyDefinitionVersion = PartitionKeyDefinitionVersion.V2,
})
```

### v2 .NET SDK

```

DocumentCollection collection = await newClient.CreateDocumentCollectionAsync(
    database,
    new DocumentCollection
    {
        Id = Guid.NewGuid().ToString(),
        PartitionKey = new PartitionKeyDefinition
        {
            Paths = new Collection<string> {"/longpartitionkey" },
            Version = PartitionKeyDefinitionVersion.V2
        }
    },
    new RequestOptions { OfferThroughput = 400 });

```

## 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 <code>2017-05-03</code> by using the <code>x-ms-version</code> request header.
Resource Manager template	version 2 by using the <code>"version":2</code> property within the <code>partitionKey</code> 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

12/5/2019 • 2 minutes to read • [Edit Online](#)

This article explains how to query a container (collection, graph, or table) 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 handles the query automatically. It routes the query to the partitions corresponding to the partition key values specified in the filter. For example, the following query is routed to the `DeviceId` partition, which holds all the documents corresponding to partition key value `XMS-0001`.

```
// Query using partition key into a class called, DeviceReading
IQueryable<DeviceReading> query = client.CreateDocumentQuery<DeviceReading>(
    UriFactory.CreateDocumentCollectionUri("myDatabaseName", "myCollectionName"))
    .Where(m => m.MetricType == "Temperature" && m.DeviceId == "XMS-0001");
```

## Cross-partition query

The following query doesn't have a filter on the partition key (`DeviceId`), and is fanned out to all partitions where it is run against the partition's index. To run a query across partitions, set `EnableCrossPartitionQuery` to true (or `x-ms-documentdb-query-enablecrosspartition` in the REST API).

The `EnableCrossPartitionQuery` property accepts a boolean value. When set to true and if your query doesn't have a partition key, Azure Cosmos DB fans out the query across partitions. The fan out is done by issuing individual queries to all the partitions. To read the query results, the client applications should consume the results from the `FeedResponse` and check for the `ContinuationToken` property. To read all the results, keep iterating on the data until the `ContinuationToken` is null.

```
// Query across partition keys into a class called, DeviceReading
IQueryable<DeviceReading> crossPartitionQuery = client.CreateDocumentQuery<DeviceReading>(
    UriFactory.CreateDocumentCollectionUri("myDatabaseName", "myCollectionName"),
    new FeedOptions { EnableCrossPartitionQuery = true })
    .Where(m => m.MetricType == "Temperature" && m.MetricValue > 100);
```

Azure Cosmos DB supports aggregate functions `COUNT`, `MIN`, `MAX`, and `AVG` over containers by using SQL. The aggregate functions over containers starting from the SDK version 1.12.0 and later. Queries must include a single aggregate operator, and must include a single value in the projection.

## 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. For example, the following query is configured to run in parallel across partitions.

```
// Cross-partition Order By Query with parallel execution
IQueryable<DeviceReading> crossPartitionQuery = client.CreateDocumentQuery<DeviceReading>(
    UriFactory.CreateDocumentCollectionUri("myDatabaseName", "myCollectionName"),
    new FeedOptions { EnableCrossPartitionQuery = true, MaxDegreeOfParallelism = 10, MaxBufferedItemCount = 100 })
    .Where(m => m.MetricType == "Temperature" && m.MetricValue > 100)
    .OrderBy(m => m.MetricValue);
```

You can manage parallel query execution by tuning the following parameters:

- **MaxDegreeOfParallelism:** 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 `MaxDegreeOfParallelism` is not specified or set to 0, which is the default value, 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.

With the same state of the collection, a parallel query returns results in the same order as a serial execution. When performing a cross-partition query that includes sorting operators (ORDER BY, TOP), the Azure Cosmos DB SDK issues the query in parallel across partitions. It merges partially sorted results in the client side to produce globally ordered results.

## 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](#)

# How to model and partition data on Azure Cosmos DB using a real-world example

12/13/2019 • 17 minutes to read • [Edit Online](#)

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.

## 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)

As 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:

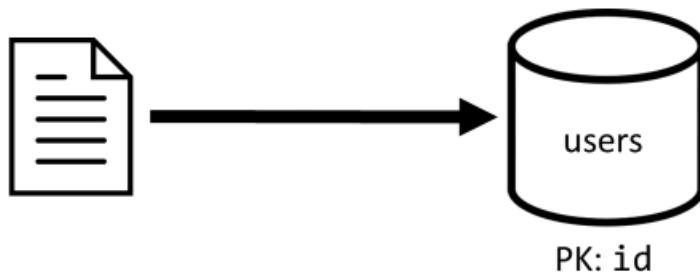
- there's no upper limit to how many posts a user can create,
- posts can be arbitrarily long,
- there's no upper limit to how many comments and likes a post can have,
- we want to be able to add a comment or a like to a post without having to update the post itself.

## 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

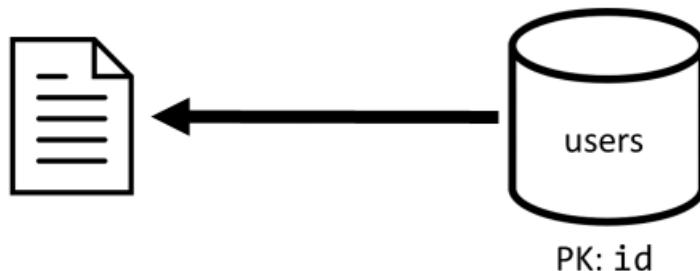
This request is straightforward to implement as we just create or update an item in the `users` container. The requests will nicely spread across all partitions thanks to the `id` partition key.



LATENCY	RU CHARGE	PERFORMANCE
7 ms	5.71 RU	□

### [Q1] Retrieve a user

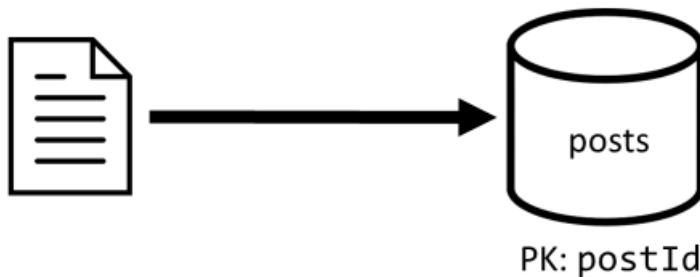
Retrieving a user is done by reading the corresponding item from the `users` container.



LATENCY	RU CHARGE	PERFORMANCE
2 ms	1 RU	□

## [C2] Create/edit a post

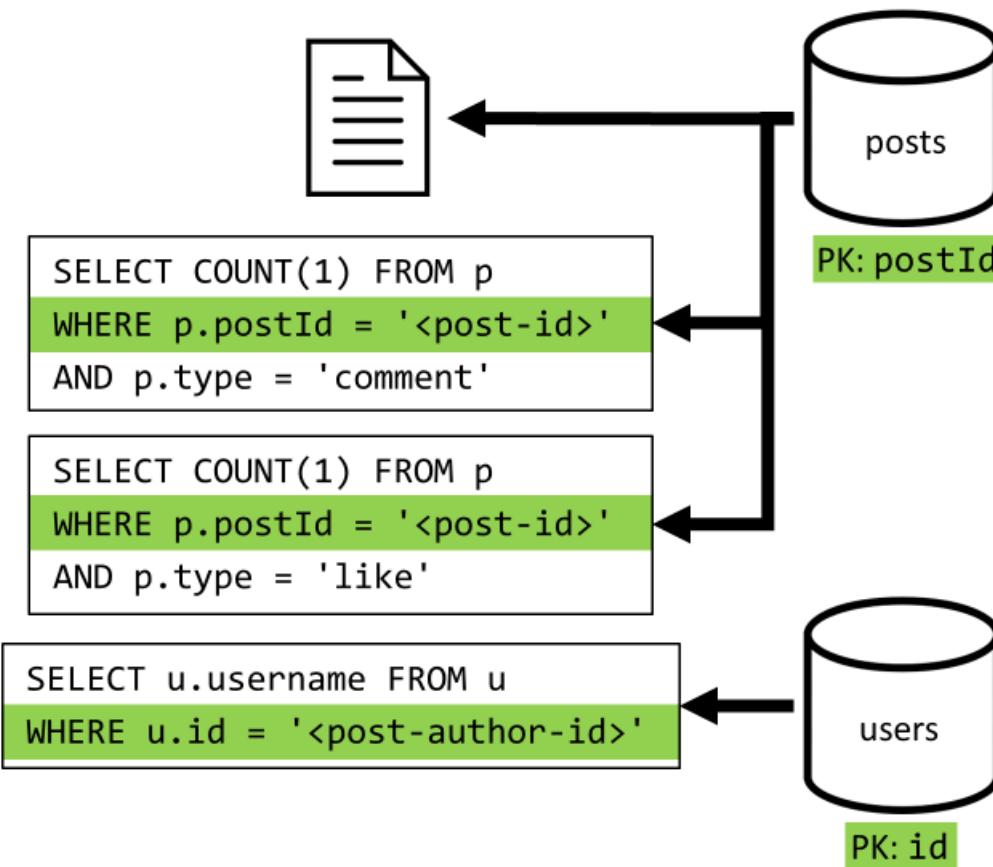
Similarly to [C1], we just have to write to the `posts` container.



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.

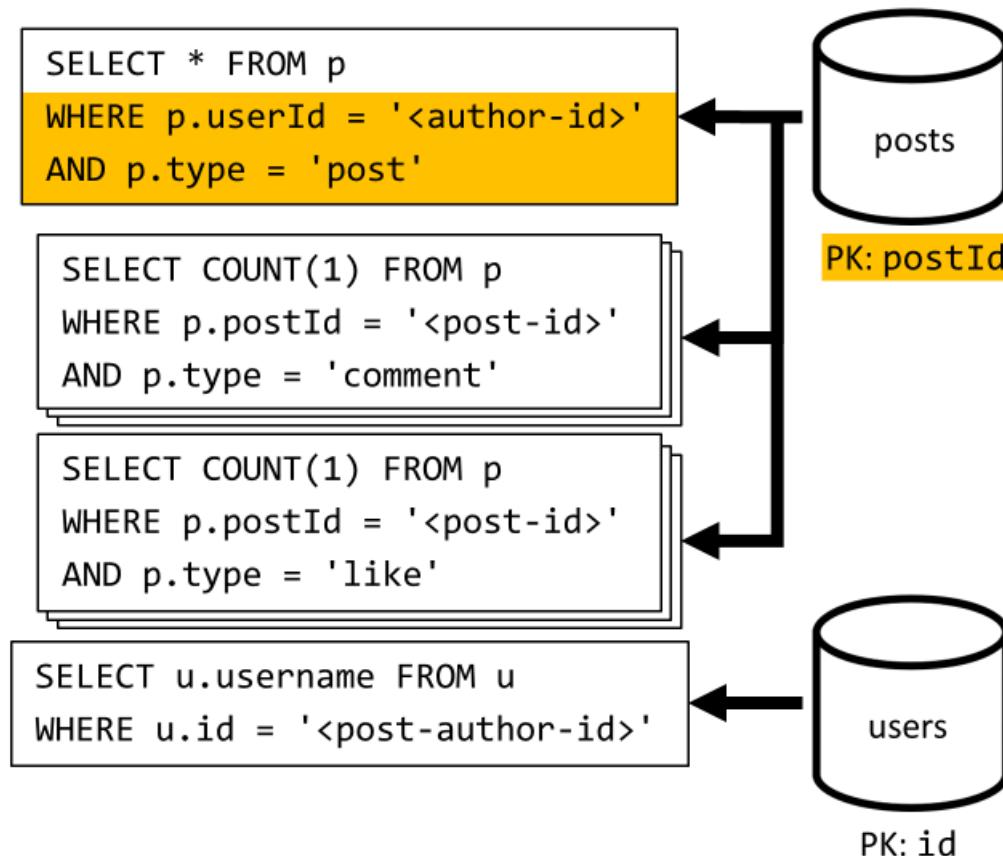


Each of the additional queries filters on the partition key of its respective container, which is exactly what we want to maximize performance and scalability. But we eventually have to perform four operations to return a single post, so we'll improve that in a next iteration.

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.



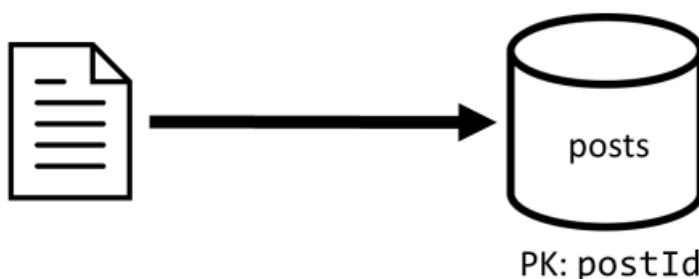
This implementation presents many drawbacks:

- 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

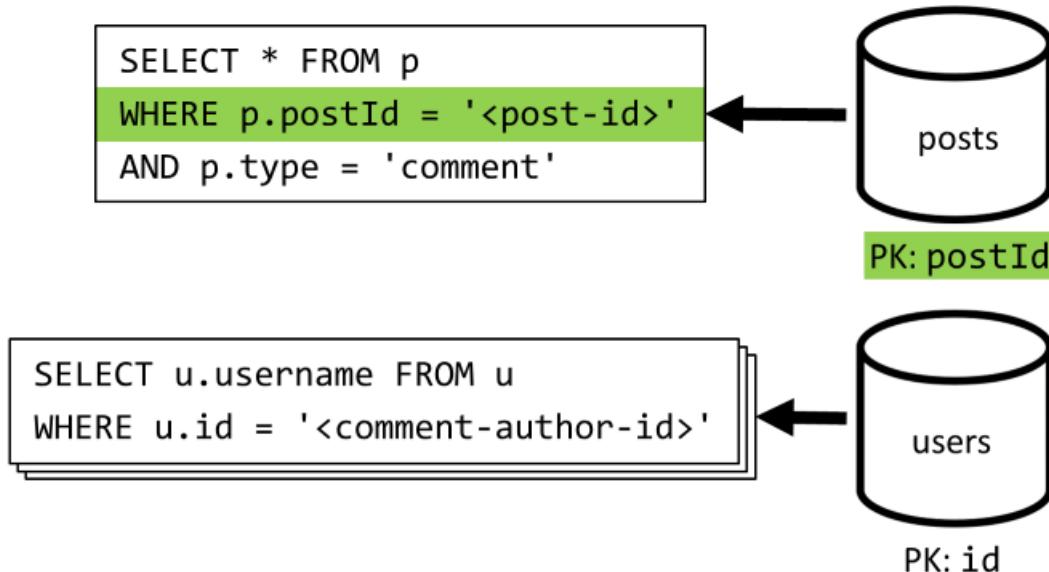
A comment is created by writing the corresponding item in the `posts` container.



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.

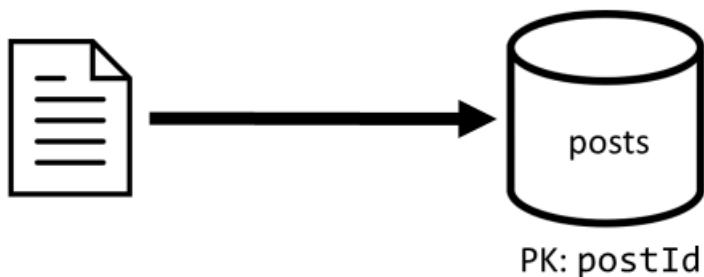


Although the main query does filter on the container's partition key, aggregating the usernames separately penalizes the overall performance. We'll improve that later on.

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.

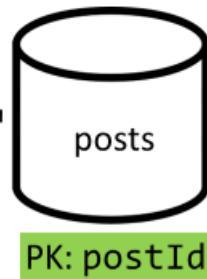


LATENCY	RU CHARGE	PERFORMANCE
6 ms	7.05 RU	□

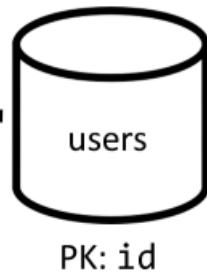
#### [Q5] List a post's likes

Just like [Q4], we query the likes for that post, then aggregate their usernames.

```
SELECT * FROM p
WHERE p.postId = '<post-id>'
AND p.type = 'like'
```



```
SELECT u.username FROM u
WHERE u.id = '<like-author-id>'
```

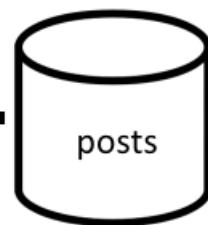


LATENCY	RU CHARGE	PERFORMANCE
59 ms	58.92 RU	□

#### [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.

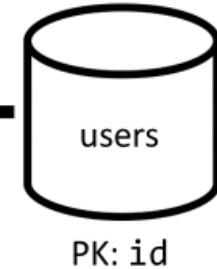
```
SELECT TOP 100 * FROM p
WHERE p.type = 'post'
ORDER BY p.creationDate DESC
```



```
SELECT COUNT(1) FROM p
WHERE p.postId = '<post-id>'
AND p.type = 'comment'
```

```
SELECT COUNT(1) FROM p
WHERE p.postId = '<post-id>'
AND p.type = 'like'
```

```
SELECT u.username FROM u
WHERE u.id = '<post-author-id>'
```



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 (**[IC3]**), 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

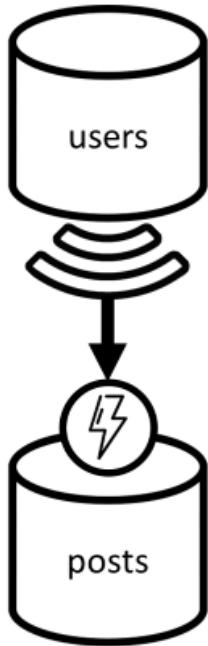
As stored procedures are executed as atomic transactions, it is guaranteed that the value of `commentCount` and the actual number of comments will always stay in sync.

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

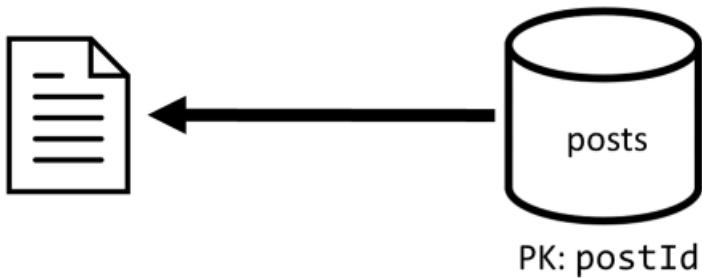
#### IMPORTANT

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:

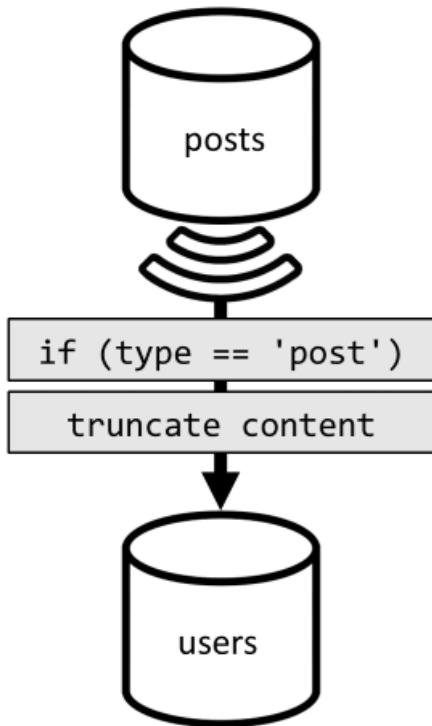
```
{
  "id": "<user-id>",
  "type": "user",
  "userId": "<user-id>",
  "username": "<username>"
}

{
  "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.



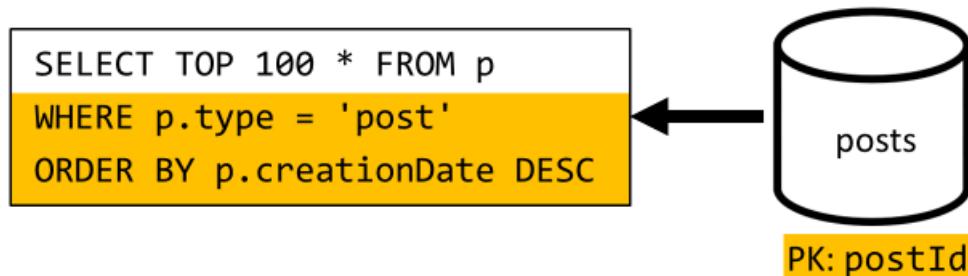
We can now route our query to the `users` container, filtering on the container's partition key.



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.

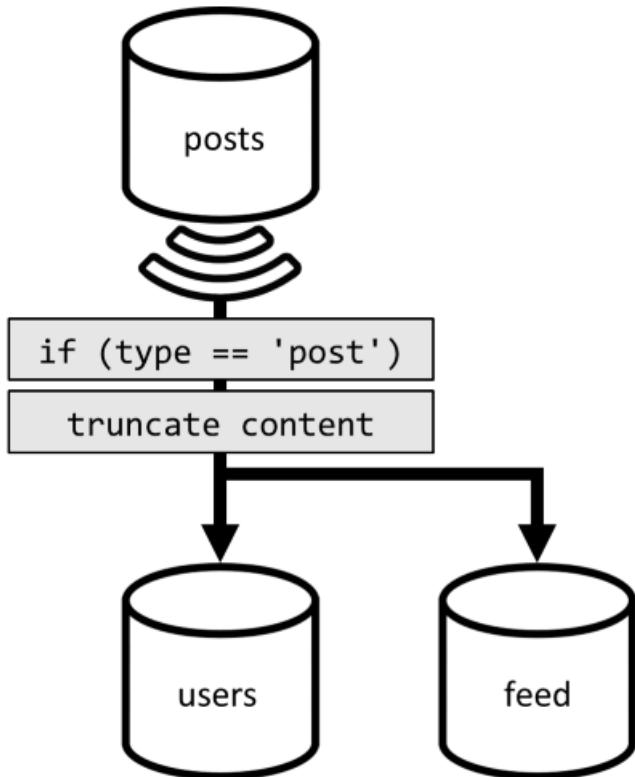
So to optimize this last request, we introduce a third container to our design, entirely dedicated to serving this

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;

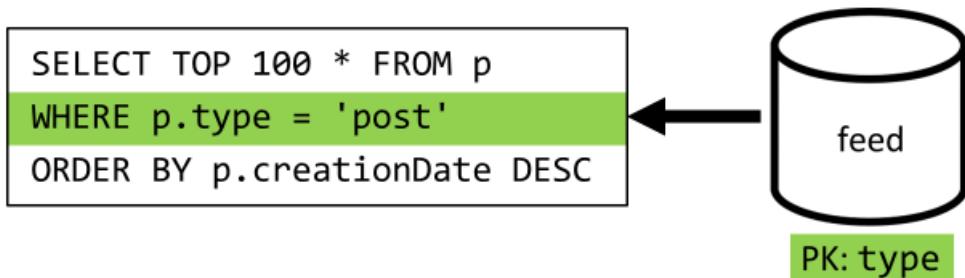
        processDocsToRemove(results, 0);
      });
  }
}

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](#)

# Migrate non-partitioned containers to partitioned containers

2/26/2020 • 4 minutes to read • [Edit Online](#)

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

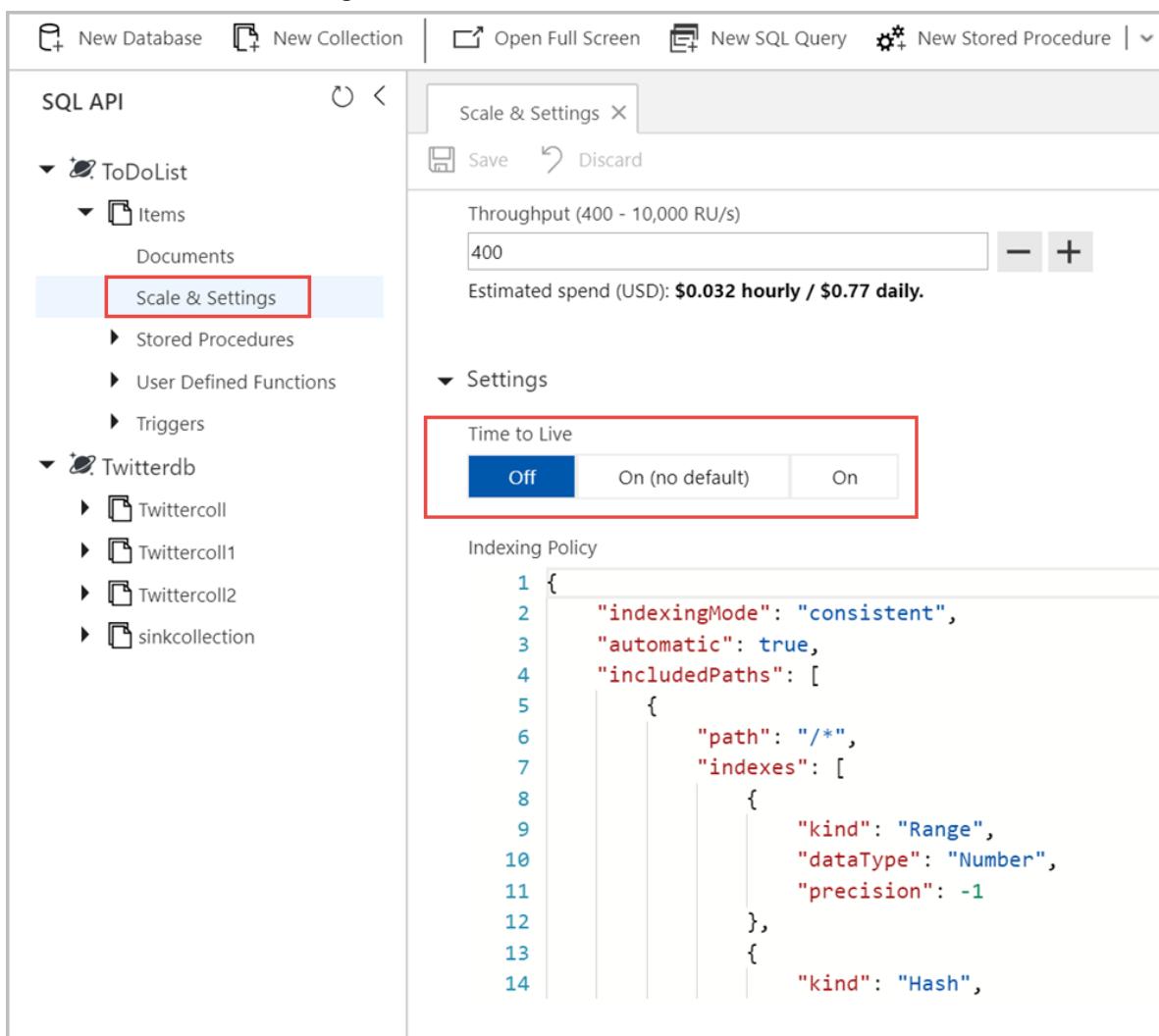
12/13/2019 • 6 minutes to read • [Edit Online](#)

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.

## 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 select an 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 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);
```

### .NET SDK V3 ([Microsoft.Azure.Cosmos](#))

```
// Create a new container with TTL enabled and without any expiration value
await client.GetDatabase("database").CreateContainerAsync(new ContainerProperties
{
    Id = "container",
    PartitionKeyPath = "/myPartitionKey",
    DefaultTimeToLive = -1 // (never expire by default)
});
```

## 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.

### .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);
```

### .NET SDK V3 ([Microsoft.Azure.Cosmos](#))

```
// Create a new container with TTL enabled and a 90 day expiration
await client.GetDatabase("database").CreateContainerAsync(new ContainerProperties
{
    Id = "container",
    PartitionKeyPath = "/myPartitionKey",
    DefaultTimeToLive = 90 * 60 * 60 * 24; // expire all documents after 90 days
});
```

## 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 select an 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": "Jic9ANWd0-EFAAAAAAAA==",
  "_self": " dbs/Jic9AA==/colls/Jic9ANWd0-E=/docs/Jic9ANWd0-EFAAAAAAAA==/",
  "_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 = "S005",
    CustomerId = "C018009186470",
    ttl = 60 * 60 * 24 * 30; // Expire sales orders in 30 days
};
```

## NodeJS SDK

```
const itemDefinition = {
  id: "doc",
  name: "sample Item",
  key: "value",
  ttl: 2
};
```

## 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 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/S005",
    new RequestOptions { PartitionKey = new PartitionKey("C018009186470") });

Document readDocument = response.Resource;
readDocument.ttl = 60 * 30 * 30; // update time to live
response = await client.ReplaceDocumentAsync(readDocument);
```

## .NET SDK V3 (Microsoft.Azure.Cosmos)

```
// This examples leverages the Sales Order class above.  
// Read a document, update its TTL, save it.  
ItemResponse<SalesOrder> itemResponse = await client.GetContainer("database",  
"container").ReadItemAsync<SalesOrder>("S005", new PartitionKey("C018009186470"));  
  
itemResponse.Resource.ttl = 60 * 30 * 30; // update time to live  
await client.GetContainer("database", "container").ReplaceItemAsync(itemResponse.Resource, "S005");
```

## 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 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/S005"),  
    new RequestOptions { PartitionKey = new PartitionKey("C018009186470") });  
  
Document readDocument = response.Resource;  
readDocument.ttl = null; // inherit the default TTL of the container  
  
response = await client.ReplaceDocumentAsync(readDocument);
```

## .NET SDK V3 (Microsoft.Azure.Cosmos)

```
// This examples leverages the Sales Order class above.  
// Read a document, turn off its override TTL, save it.  
ItemResponse<SalesOrder> itemResponse = await client.GetContainer("database",  
"container").ReadItemAsync<SalesOrder>("S005", new PartitionKey("C018009186470"));  
  
itemResponse.Resource.ttl = null; // inherit the default TTL of the container  
await client.GetContainer("database", "container").ReplaceItemAsync(itemResponse.Resource, "S005");
```

## 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 they have explicitly overridden the previous default TTL value.

## .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);
```

## .NET SDK V3 (Microsoft.Azure.Cosmos)

```
// Get the container, update DefaultTimeToLive to null
ContainerResponse containerResponse = await client.GetContainer("database", "container").ReadContainerAsync();
// Disable TTL
containerResponse.Resource.DefaultTimeToLive = null;
await client.GetContainer("database", "container").ReplaceContainerAsync(containerResponse.Resource);
```

## Next steps

Learn more about time to live in the following article:

- [Time to live](#)

# Manage indexing policies in Azure Cosmos DB

2/20/2020 • 9 minutes to read • [Edit Online](#)

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.

## 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 can 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 can 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 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 filter and have an ORDER BY clause on different properties.

### 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 secondary index is required (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.

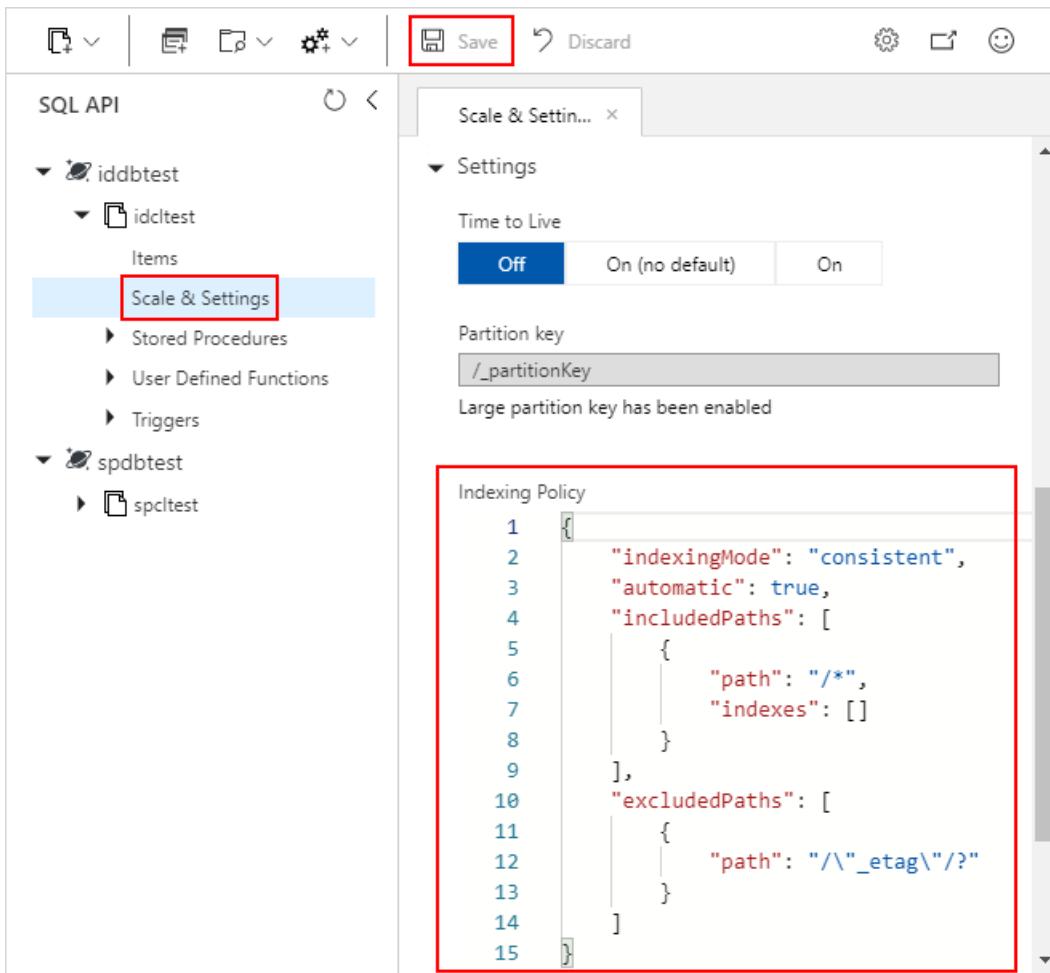
### NOTE

When updating indexing policy, writes to Azure Cosmos DB will be uninterrupted. During re-indexing, queries may return partial results as the index is being updated.

## 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 select an 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 V2

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 .NET SDK V3

The `ContainerProperties` object from the [.NET SDK v3](#) (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
ContainerResponse containerResponse = await client.GetContainer("database",
"container").ReadContainerAsync();
// 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
SpatialPath spatialPath = new SpatialPath
{
    Path = "/locations/*"
};
spatialPath.SpatialTypes.Add(SpatialType.Point);
containerResponse.Resource.IndexingPolicy.SpatialIndexes.Add(spatialPath);
// 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.GetContainer("database", "container").ReplaceContainerAsync(containerResponse.Resource);

```

To track the index transformation progress, 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
ContainerResponse containerResponse = await client.GetContainer("database",
    "container").ReadContainerAsync(new ContainerRequestOptions { PopulateQuotaInfo = true });
// retrieve the index transformation progress from the result
long indexTransformationProgress = long.Parse(containerResponse.Headers["x-ms-documentdb-collection-index-
transformation-progress"]);
```

When defining a custom indexing policy while creating a new container, the SDK V3's fluent API lets you write this definition in a concise and efficient way:

```
await client.GetDatabase("database").DefineContainer(name: "container", partitionKeyPath: "/myPartitionKey")
    .WithIndexingPolicy()
        .WithIncludedPaths()
            .Path("/")
        .Attach()
    .WithExcludedPaths()
        .Path("/name/*")
    .Attach()
    .WithSpatialIndex()
        .Path("/locations/*", SpatialType.Point)
    .Attach()
    .WithCompositeIndex()
        .Path("/name", CompositePathSortOrder.Ascending)
        .Path("/age", CompositePathSortOrder.Descending)
    .Attach()
.Attach()
.CreateIfNotExistsAsync();
```

## 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 V3

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": "/name/*"}]
}
```

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)
```

## Use the Python SDK V4

When using the [Python SDK V4](#), 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

```
database_client = cosmos_client.get_database_client('database')
container_client = database_client.get_container_client('container')
container = container_client.read()
```

Set the indexing mode to consistent

```
indexingPolicy = {
    'indexingMode': 'consistent'
}
```

Define an indexing policy with an included path and a spatial index

```
indexingPolicy = {
    "indexingMode": "consistent",
    "spatialIndexes": [
        {"path": "/location/*", "types": ["Point"]}
    ],
    "includedPaths": [{"path": "/age/*", "indexes": []}],
    "excludedPaths": [{"path": "/name/*"}]
}
```

Define an indexing policy with an excluded path

```
indexingPolicy = {
    "indexingMode": "consistent",
    "includedPaths": [{"path": "/*", "indexes": []}],
    "excludedPaths": [{"path": "/name/*"}]
}
```

Add a composite index

```
indexingPolicy['compositeIndexes'] = [
    [
        {
            "path": "/name",
            "order": "ascending"
        },
        {
            "path": "/age",
            "order": "descending"
        }
    ]
]
```

Update the container with changes

```
response = database_client.replace_container(container_client, container['partitionKey'], indexingPolicy)
```

## Next steps

Read more about the indexing in the following articles:

- [Indexing overview](#)
- [Indexing policy](#)

# Define unique keys for an Azure Cosmos container

12/5/2019 • 2 minutes to read • [Edit Online](#)

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**

The screenshot shows the 'Add Container' dialog box. At the top, it says 'Add Container' and has a close button 'X'. Below that, there are fields for 'Database id': 'my-database' (radio button selected) and 'Container id': 'my-container'. There is also a checkbox for 'Provision database throughput'. Under 'Partition key', the value '/id' is entered. Below that, under 'Throughput (400 - 100,000 RU/s)', the value '400' is selected from a dropdown. A note below says 'Estimated spend (USD): \$0.032 hourly / \$0.77 daily (1 region x 400RU/s x \$0.00008/RU)'. The 'Unique keys' section is highlighted with a red box, containing the paths '/firstName, /lastName, /emailAddress' and '/address/zipCode'. Below this, a red box surrounds the '+ Add unique key' button. At the bottom, there is an 'OK' button.

## Use Powershell

To create a container with unique keys see, [Create an Azure Cosmos container with unique key and TTL](#)

## Use the .NET SDK V2

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 .NET SDK V3

When creating a new container using the [.NET SDK v3](#), use the SDK's fluent API to declare unique keys in a concise and readable way.

```
await client.GetDatabase("database").DefineContainer(name: "container", partitionKeyPath: "/myPartitionKey")
    .WithUniqueKey()
    .Path("/firstName")
    .Path("/lastName")
    .Path("/emailAddress")
    .Attach()
    .WithUniqueKey()
    .Path("/address/zipCode")
    .Attach()
    .CreateIfNotExistsAsync();
```

## 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](#)

# Get SQL query execution metrics and analyze query performance using .NET SDK

9/15/2019 • 5 minutes to read • [Edit Online](#)

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 however 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

8/19/2019 • 12 minutes to read • [Edit Online](#)

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
<code>EnableCrossPartitionQuery</code>	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
<code>EnableScanInQuery</code>	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.
<code>MaxItemCount</code>	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.
<code>MaxBufferedItemCount</code>	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.
<code>MaxDegreeOfParallelism</code>	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.
<code>PopulateQueryMetrics</code>	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.
<code>RequestContinuation</code>	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.
<code>ResponseContinuationTokenLimitInKb</code>	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%3drp1Hi83Y8aVV5V6LzZ6xhtQVXRAMz0WNMnUvriUv%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;queryPhysicalPlanBuildTimeInMs=0.10;queryOptimizationTimeInMs=0.00;VMExecutionTimeInMs=32.56;indexLookupTimeInMs=0.36;documentLoadTimeInMs=9.58;systemFunctionExecuteTimeInMs=0.00;userFunctionExecuteTimeInMs=0.00;retrievedDocumentCount=2000;retrievedDocumentSize=1125600;outputDocumentCount=2000;writeOutputTimeInMs=18.10;indexUtilizationRatio=1.00
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
<code>x-ms-item-count</code>	The number of items returned in the response. This is dependent on the supplied <code>x-ms-max-item-count</code> , the number of items that can be fit within the maximum response payload size, the provisioned throughput, and query execution time.
<code>x-ms-continuation:</code>	The continuation token to resume execution of the query, if additional results are available.
<code>x-ms-documentdb-query-metrics</code>	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 <code>x-ms-documentdb-populatequerymetrics</code> is set to <code>True</code> .
<code>x-ms-request-charge</code>	The number of <a href="#">request units</a> 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  $(1000 \text{ RU/s}) / (5 \text{ RU/query}) = 200 \text{ query/s}$  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
<code>totalExecutionTimeInMs</code>	milliseconds	Query execution time
<code>queryCompileTimeInMs</code>	milliseconds	Query compile time
<code>queryLogicalPlanBuildTimeInMs</code>	milliseconds	Time to build logical query plan

METRIC	UNIT	DESCRIPTION
<code>queryPhysicalPlanBuildTimeInMs</code>	milliseconds	Time to build physical query plan
<code>queryOptimizationTimeInMs</code>	milliseconds	Time spent in optimizing query
<code>VMExecutionTimeInMs</code>	milliseconds	Time spent in query runtime
<code>indexLookupTimeInMs</code>	milliseconds	Time spent in physical index layer
<code>documentLoadTimeInMs</code>	milliseconds	Time spent in loading documents
<code>systemFunctionExecuteTimeInMs</code>	milliseconds	Total time spent executing system (built-in) functions in milliseconds
<code>userFunctionExecuteTimeInMs</code>	milliseconds	Total time spent executing user-defined functions in milliseconds
<code>retrievedDocumentCount</code>	count	Total number of retrieved documents
<code>retrievedDocumentSize</code>	bytes	Total size of retrieved documents in bytes
<code>outputDocumentCount</code>	count	Number of output documents
<code>writeOutputTimeInMs</code>	milliseconds	Query execution time in milliseconds
<code>indexUtilizationRatio</code>	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
<code>SELECT TOP 100 * FROM c</code>	<code>"RetrievedDocumentCount": 101</code>	The number of documents retrieved is 100+1 to match the TOP clause. Query time is mostly spent in <code>WriteOutputTime</code> and <code>DocumentLoadTime</code> since it is a scan.
<code>SELECT TOP 500 * FROM c</code>	<code>"RetrievedDocumentCount": 501</code>	RetrievedDocumentCount is now higher (500+1 to match the TOP clause).
<code>SELECT * FROM c WHERE c.N = 55</code>	<code>"IndexLookupTime": "00:00:00.0009500"</code>	About 0.9 ms is spent in IndexLookupTime for a key lookup, because it's an index lookup on <code>/N/?</code> .

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 <code>/N/?</code> .
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 <code>c.N</code> .
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 <code>/Name/?</code> . 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](#)

# Performance tips for Azure Cosmos DB and .NET

2/26/2020 • 16 minutes to read • [Edit Online](#)

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 the [SQL .NET SDK](#).

So if you're asking "How can I improve my database performance?" consider the following options:

## Hosting recommendations

### 1. For query-intensive workloads, use Windows 64-bit instead of Linux or Windows 32 host processing

Windows 64-bit host processing is recommended for improved performance. The SQL SDK includes a native ServiceInterop.dll to parse and optimize queries locally, and is only supported on the Windows x64 platform. For Linux and other unsupported platforms where the ServiceInterop.dll is not available it will do an additional network call to the gateway to get the optimized query. The following types of applications have 32-bit host process as the default, so in order to change that to 64-bit, follow these steps based on the type of your application:

- For Executable applications, this can be done by setting the [Platform target](#) to **x64** in the **Project Properties** window, on the **Build** tab.
- For VSTest based test projects, this can be done by selecting **Test->Test Settings->Default Processor Architecture as X64**, from the **Visual Studio Test** menu option.
- For locally deployed ASP.NET Web applications, this can be done by checking the **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, this can be done by choosing the **Platform as 64-bit** in the **Application Settings** on the Azure portal.

#### NOTE

Visual studio defaults new projects to Any CPU. It's recommend to set the project to x64 to avoid it switching to x86. Any CPU project can easily switch to x86 if any dependency is added that is x86 only.

The ServiceInterop.dll needs to be in the same folder as the SDK dll is being executed from. This should only need to be done for users manually coping dlls or have custom build/deployment systems.

### 2. Turn on server-side Garbage Collection(GC)

Reducing the frequency of garbage collection may help in some cases. In .NET, set [gcServer](#) to true.

### 3. 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.

**NOTE**

High CPU usage can cause increased latency and request timeout exceptions.

## Networking

### 1. Connection policy: Use direct connection mode

How a client connects to Azure Cosmos DB has important implications on performance, especially in terms of observed client-side latency. There are two key configuration settings available for configuring client Connection Policy – the connection *mode* and the connection *protocol*. The two available modes are:

- Gateway mode

Gateway mode is supported on all SDK platforms and is the configured default for [Microsoft.Azure.DocumentDB SDK](#). If your application runs 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. Gateway connection mode is also recommended when you run applications in environments with limited number of socket connections.

When using the SDK in Azure Functions, particularly in [consumption plan](#), be mindful of the current [limits in connections](#). In that case, gateway mode might be recommended if you are also working with other HTTP based clients within your Azure Functions application.

- Direct mode

Direct mode supports connectivity through TCP protocol and is the default connectivity mode if you are using [Microsoft.Azure.Cosmos/.Net V3 SDK](#).

When using gateway mode, Cosmos DB uses port 443 and ports 10250, 10255 and 10256 when using Azure Cosmos DB's API for MongoDB. The 10250 port maps to a default MongoDB instance without geo-replication and 10255/10256 ports map to the MongoDB instance with geo-replication functionality. When using TCP in 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. If these ports are not open and you attempt to use TCP, you receive a 503 Service Unavailable error. The following table shows connectivity modes available for different APIs and the service ports user for each API:

CONNECTION MODE	SUPPORTED PROTOCOL	SUPPORTED SDKS	API/SERVICE PORT
Gateway	HTTPS	All SDKs	SQL(443), Mongo(10250, 10255, 10256), Table(443), Cassandra(10350), Graph(443)
Direct	TCP	.NET SDK	Ports within 10,000- 20,000 range

Azure Cosmos DB offers a simple and open RESTful programming model over HTTPS.

Additionally, it offers an efficient TCP protocol, which is also RESTful in its communication model

and is available through the .NET client SDK. TCP protocol uses SSL for initial authentication and encrypting traffic. For best performance, use the TCP protocol when possible.

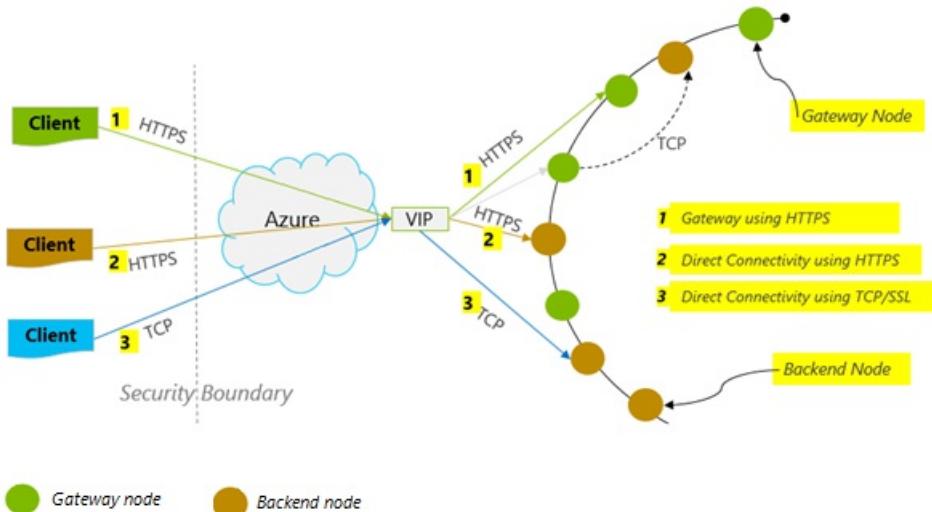
For SDK V3, the connectivity mode is configured while creating the CosmosClient instance, as part of the CosmosClientOptions, remember that Direct mode is the default.

```
var serviceEndpoint = new Uri("https://contoso.documents.net");
var authKey = "your authKey from the Azure portal";
CosmosClient client = new CosmosClient(serviceEndpoint, authKey,
new CosmosClientOptions
{
    ConnectionMode = ConnectionMode.Gateway // ConnectionMode.Direct is the default
});
```

For the Microsoft.Azure.DocumentDB SDK, the connectivity mode is configured during the construction of the DocumentClient instance with the ConnectionPolicy parameter. If Direct Mode is used, the Protocol can also be set within the ConnectionPolicy parameter.

```
var serviceEndpoint = new Uri("https://contoso.documents.net");
var 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
});
```

Because TCP is only supported in Direct Mode, if gateway mode is used, then the HTTPS protocol is always used to communicate with the Gateway and the Protocol value in the ConnectionPolicy is ignored.



## 2. Call `OpenAsync` to avoid startup latency on first request

By default, the first request has a higher latency because it has to fetch the address routing table. When using the [SDK V2](#), to avoid this startup latency on the first request, you should call `OpenAsync()` once during initialization as follows.

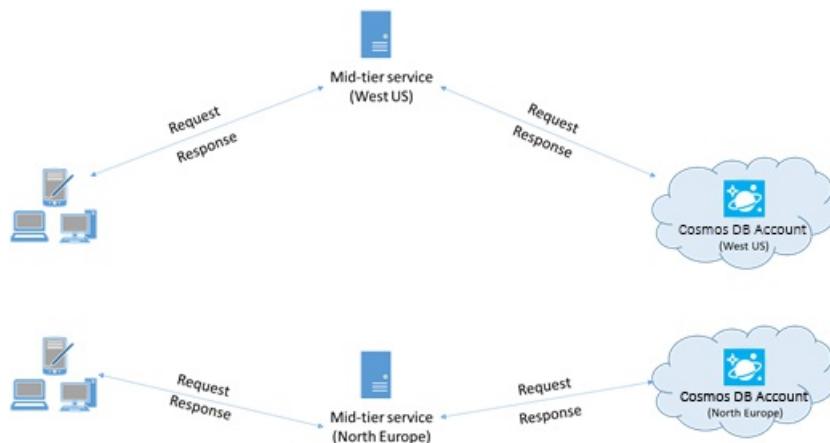
```
await client.OpenAsync();
```

#### **NOTE**

OpenAsync method will generate requests to obtain the address routing table for all the containers in the account. For accounts that have many containers but their application accesses a subset of them, it would generate an unnecessary amount of traffic that makes the initialization slow. So using OpenAsync method might not be useful in this scenario as it slows down application startup.

### **3. 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](#).



### **4. Increase number of threads/tasks**

Since calls to Azure Cosmos DB are made over the network, you may need to vary the degree of parallelism of your requests so that the client application spends very little time waiting between requests. For example, if you're using .NET's [Task Parallel Library](#), create in the order of 100s of Tasks reading or writing to Azure Cosmos DB.

### **5. Enable accelerated networking**

In order to reduce latency and CPU jitter, we recommend that the client virtual machines are accelerated networking enabled. See the [Create a Windows virtual machine with Accelerated Networking](#) or [Create a Linux virtual machine with Accelerated Networking](#) articles to enable accelerated networking.

## **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 Stream APIs**

The [.NET SDK V3](#) contains stream APIs that can receive and return data without serializing.

The middle-tier applications that don't consume the responses from the SDK directly but relay them to other application tiers can benefit from the stream APIs. See the [Item management](#) samples for examples on stream handling.

### 3. Use a singleton Azure Cosmos DB client for the lifetime of your application

Each DocumentClient and CosmosClient 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 the SDK client, it is recommended to use a single instance per AppDomain for the lifetime of the application.

### 4. Increase System.Net MaxConnections 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 MaxConnections to a higher value (100-1000) so that the client library can utilize multiple simultaneous connections to Azure Cosmos DB. In the .NET SDK 1.8.0 and above, the default value for [ServicePointManager.DefaultConnectionLimit](#) is 50 and to change the value, you can set the [Documents.Client.ConnectionPolicy.MaxConnectionLimit](#) to a higher value.

### 5. Tuning parallel queries for partitioned collections

SQL .NET 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. Parallel queries provide two parameters that users can tune to custom-fit their requirements, (a) MaxDegreeOfParallelism: to control the maximum number of partitions then can be queried in parallel, and (b) MaxBufferedItemCount: to control the number of pre-fetched results.

(a) **Tuning degree of parallelism:** Parallel query works by querying multiple partitions in parallel. However, data from an individual partition is fetched serially with respect to the query. Setting the `MaxDegreeOfParallelism` in [SDK V2](#) or `MaxConcurrency` in [SDK V3](#) to the number of partitions 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 set the degree of parallelism to a high number, and the system chooses the minimum (number of partitions, user provided input) as the 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 MaxBufferedItemCount:** 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. MaxBufferedItemCount is the parameter to limit the number of pre-fetched results. Setting MaxBufferedItemCount to the expected number of results returned (or a higher number) allows the query to receive maximum benefit from pre-fetching.

Pre-fetching works the same way irrespective of the degree of parallelism, and there is a single buffer for the data from all partitions.

### 6. Implement backoff at RetryAfter 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 SQL .NET and Java, version 1.9.0 and above of the Node.js and Python, and all supported versions of the .NET Core SDKs. For more information, [RetryAfter](#).

With version 1.19 and later of the .NET SDK, there is a mechanism to log additional diagnostic information and troubleshoot 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 the number of times you observed 429s for a given request.

```
ResourceResponse<Document> readDocument = await  
this.readClient.ReadDocumentAsync(oldDocuments[i].SelfLink);  
readDocument.RequestDiagnosticsString
```

## 7. Cache document URIs for lower read latency

Cache document URIs whenever possible for the best read performance. You have to define logic to cache the resource ID when you create the resource. Resource ID based lookups are faster than name based lookups, so caching these values improves the performance.

## 8. Tune the page size for queries/read feeds for better performance

When performing a bulk read of documents using read feed functionality (for example, `ReadDocumentFeedAsync`) 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 `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.

### NOTE

The `maxItemCount` property shouldn't be used just for pagination purpose. It's main usage 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 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 most 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 value for `maxItemCount`, the number of trips required to send the data within the TCP packet are high, which impacts the performance. So if you are not sure what value to set for `maxItemCount` property, it's best to set it to -1 and let the SDK choose the default value.

## 9. Increase number of threads/tasks

See [Increase number of threads/tasks](#) in the Networking section.

# Indexing Policy

## 1. Exclude unused paths from indexing for faster writes

Cosmos DB's indexing policy also allows you to specify which document paths to include or exclude from indexing by leveraging Indexing Paths (`IndexingPolicy.IncludedPaths` and `IndexingPolicy.ExcludedPaths`). 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 (a subtree) from indexing 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"), excluded);
```

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> in the .NET SDK) to measure the number of request units consumed by these 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 (i.e., 2000 RUs / second). For example, 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 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 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 (i.e., 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

For a sample application used to evaluate Azure Cosmos DB for high-performance scenarios on a few client machines, see [Performance and scale testing with Azure Cosmos DB](#).

Also, 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 Java

8/19/2019 • 10 minutes to read • [Edit Online](#)

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 the [SQL Java SDK](#).

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:

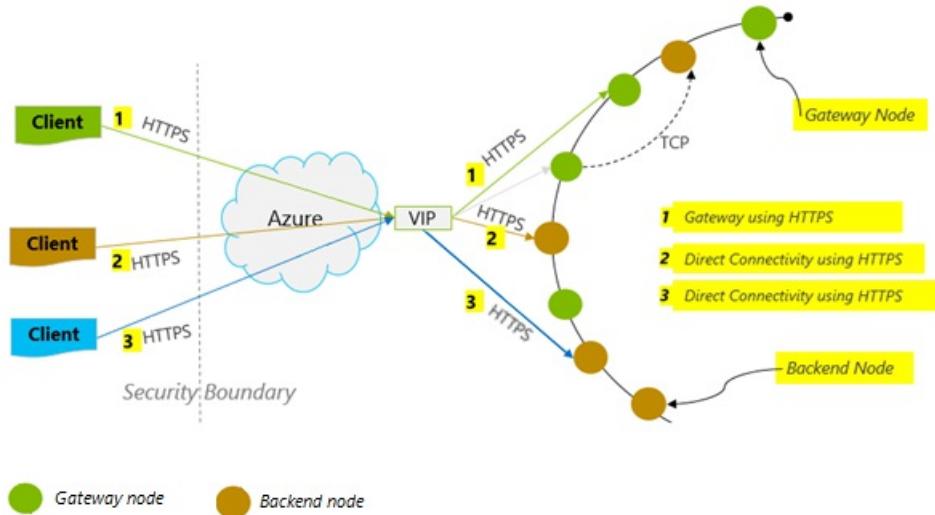
- a. [Gateway \(default\)](#)
- b. [DirectHttps](#)

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 Java SDK uses HTTPS as a transport protocol. HTTPS uses SSL for initial authentication and encrypting traffic. When using the Java SDK, only HTTPS port 443 needs to be open.

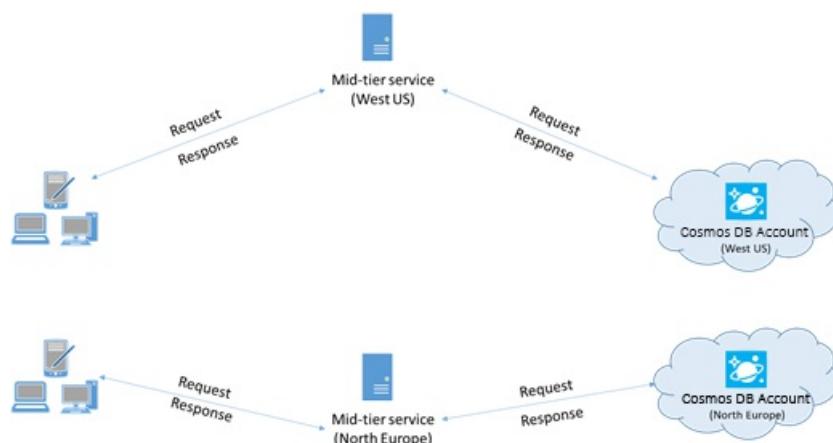
The `ConnectionMode` is configured during the construction of the `DocumentClient` instance with the `ConnectionPolicy` parameter.

```
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 Java SDK, the default value for [ConnectionPolicy.getMaxPoolSize](#) is 100. Use [setMaxPoolSize](#) to change the value.

#### 4. Tuning parallel queries for partitioned collections

Azure Cosmos DB SQL 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 [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 of the documents (a.k.a. a subtree) from indexing using the "\*" wildcard.

```
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.

```
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 tips for Azure Cosmos DB and Async Java

2/1/2020 • 14 minutes to read • [Edit Online](#)

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 [SQL Async Java SDK](#).

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 Async Java SDK, the two available ConnectionModes are:

- [Gateway \(default\)](#)
- [Direct](#)

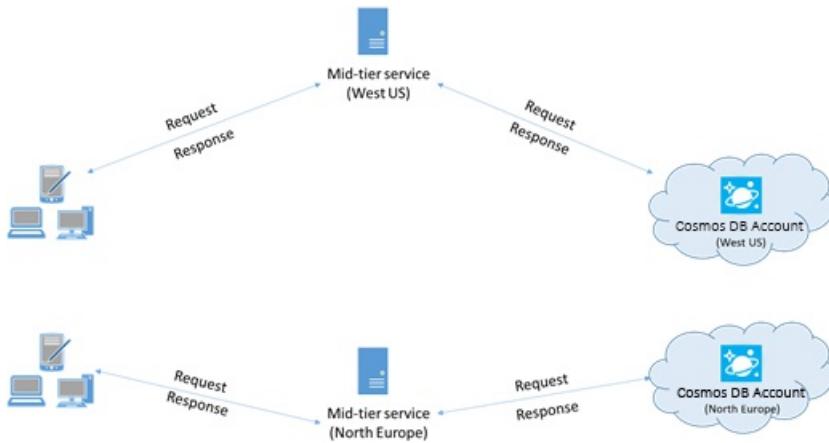
Gateway mode is supported on all SDK platforms and it is the configured option by default. If your applications run within a corporate network with strict firewall restrictions, Gateway mode is the best choice since it uses the standard HTTPS port and a single endpoint. The performance tradeoff, however, 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.

```
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 SDK](#) pages to determine the most recent SDK and review improvements.

- **Use a singleton Azure Cosmos DB client for the lifetime of your application**

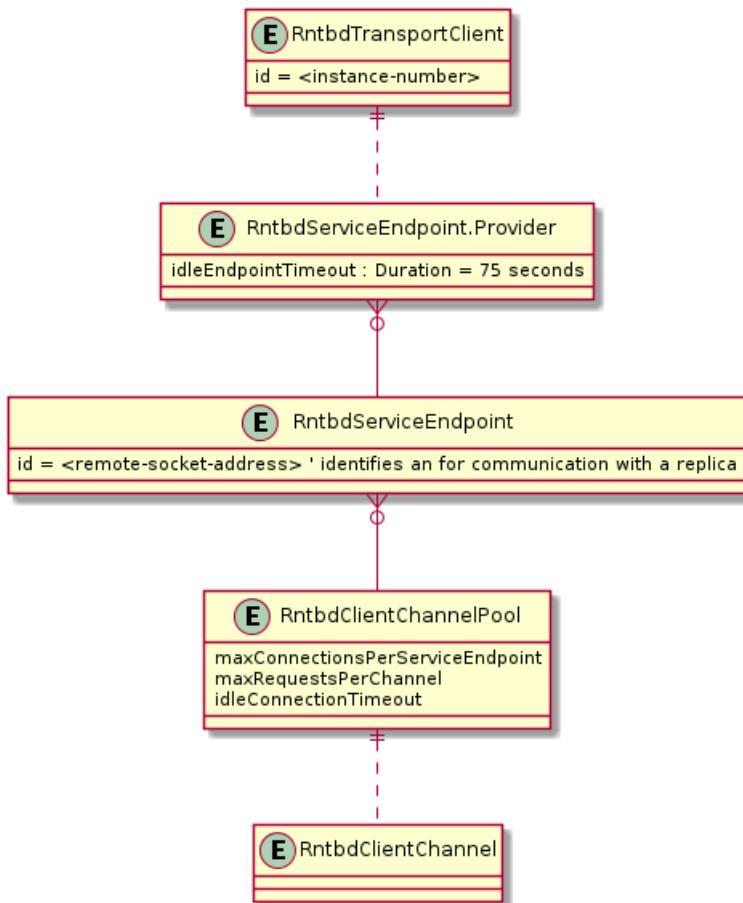
Each `AsyncDocumentClient` instance is thread-safe and performs efficient connection management and address caching. To allow efficient connection management and better performance by `AsyncDocumentClient`, it is recommended to use a single instance of `AsyncDocumentClient` per AppDomain for the lifetime of the application.

- **Tuning ConnectionPolicy**

By default, Direct mode Cosmos DB requests are made over TCP when using the Async Java SDK. Internally the SDK uses a special Direct mode architecture to dynamically manage network resources and get the best performance.

In the Async Java SDK, 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 Troubleshooting](#) article as a baseline for resolving any Async Java 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 Async Java SDK 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 SQL Async Java SDK 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 Async Java SDK 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:

```
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.

```
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 Async Java SDK.

- **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
```

- **Use native SSL implementation for netty**

Netty can use OpenSSL directly for SSL implementation stack to achieve better performance. In the absence of this configuration netty will fall back to Java's default SSL implementation.

on Ubuntu:

```
sudo apt-get install openssl  
sudo apt-get install libapr1
```

and add the following dependency to your project maven dependencies:

```
<dependency>  
  <groupId>io.netty</groupId>  
  <artifactId>netty-tcnative</artifactId>  
  <version>2.0.20.Final</version>  
  <classifier>linux-x86_64</classifier>  
</dependency>
```

For other platforms (Red Hat, Windows, Mac, etc.) refer to these instructions <https://netty.io/wiki/forked-tomcat-native.html>

## 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.

```
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>.

```
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 and scale testing with Azure Cosmos DB

1/21/2020 • 4 minutes to read • [Edit Online](#)

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

2/26/2020 • 2 minutes to read • [Edit Online](#)

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

ITEM SIZE	COST OF 1 MILLION READS	COST OF 1 MILLION WRITES
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.

# Troubleshoot query issues when using Azure Cosmos DB

2/26/2020 • 13 minutes to read • [Edit Online](#)

This article walks through a general recommended approach for troubleshooting queries in Azure Cosmos DB. While the steps outlined in this document should not be considered a "catch all" for potential query issues, we have included the most common performance tips here. You should use this document as a starting place for troubleshooting slow or expensive queries in Azure Cosmos DB's core (SQL) API. You can also use [diagnostics logs](#) to identify queries that are slow or consume significant amounts of throughput.

You can broadly categorize query optimizations in Azure Cosmos DB: Optimizations that reduce the Request Unit (RU) charge of the query and optimizations that just reduce latency. By reducing the RU charge of a query, you will almost certainly decrease latency as well.

This document will use examples that can be recreated using the [nutrition](#) dataset.

## Important

- For the best performance please follow the [Performance Tips](#).

### NOTE

Windows 64-bit host processing is recommended for improved performance. The SQL SDK includes a native ServiceInterop.dll to parse and optimize queries locally, and is only supported on the Windows x64 platform. For linux and other unsupported platforms where the ServiceInterop.dll is not available it will do an additional network call to the gateway to get the optimized query.

- Cosmos DB query does not support a minimum item count.
  - Code should handle any page size from 0 to max item count
  - The number of items in a page can and will change without any notice.
- Empty pages are expected for queries, and can appear at any time.
  - The reason empty pages are exposed in the SDKs is it allows more opportunities to cancel the query. It also makes it clear that the SDK is doing multiple network calls.
  - Empty pages can show up in existing workloads because a physical partition is split in Cosmos DB. First partition now has 0 results, which causes the empty page.
  - Empty pages are caused by the backend preempting the query because the query is taking more than some fixed amount of time on the backend to retrieve the documents. If Cosmos DB preempts a query it will return a continuation token which will allow the query to continue.
- Make sure to drain the query completely. Look at the SDK samples and use a while loop on the `FeedIterator.HasMoreResults` to drain the entire query.

### Obtaining query metrics:

When optimizing a query in Azure Cosmos DB, the first step is always to [obtain the query metrics](#) for your query. These are also available through the Azure portal as shown below:

The screenshot shows the Azure Cosmos DB Data Explorer interface. On the left, the sidebar includes sections like Overview, Activity log, Tags, Diagnose and solve problems, Quick start, Notifications, and Data Explorer (which is selected). Under Settings, there are options for Replicate data globally, Default consistency, Firewall and virtual networks, Private Endpoint Connections, CORS, Keys, Add Azure Search, Add Azure Function, Advanced security (preview), Preview Features, Locks, Export template, and Browse. The main area shows a SQL API query window with a query: "SELECT \* FROM c". Below it is a table titled "Query Stats" with the following data:

Metric	Value
Request Charge	13.690000000000001 RU
Showing Results	1 - 10
Retrieved document count	30
Retrieved document size	209013 bytes
Output document count	30
Output document size	209190 bytes
Index hit document count	30
Index lookup time	0 ms
Document load time	1.5699 ms
Query engine execution time	0.0401 ms
System function execution time	0 ms
User defined function execution time	0 ms
Document write time	0.12000000000000001 ms
Round Trips	1

At the bottom of the stats table, there's a link: "↓ Per-partition query metrics (CSV)".

After obtaining query metrics, compare the Retrieved Document Count with the Output Document Count for your query. Use this comparison to identify the relevant sections to reference below.

The Retrieved Document Count is the number of documents that the query 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, then there was at least one part of your query that was unable to utilize the index and needed to do a scan.

You can reference the below section to understand the relevant query optimizations for your scenario:

### **Query's RU charge is too high**

#### **Retrieved Document Count is significantly greater than Output Document Count**

- [Include necessary paths in the indexing policy](#)
- [Understand which system functions utilize the index](#)
- [Queries with 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**

- [Avoid cross partition queries](#)
- [Filters on multiple properties](#)
- [Queries with 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 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, then there was at least one part of your query that was unable to utilize the index and needed to do a scan.

Below is 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

Retrieved Document Count (60,951) is significantly greater than Output Document Count (7) so this query needed to do a scan. In this case, the system function `UPPER()` does not utilize the 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 path specified in the index policy should match (case-sensitive) the property in the JSON documents.

If we run a simple query on the `nutrition` dataset, we 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 additional properties to the indexing policy at any time, with no impact to write availability or performance. If you add a new property to the index, queries that use this property will immediately utilize the new available index. The query will utilize the new index while it is being built. As a result, query results may be inconsistent as the index rebuild is in progress. If a new property is indexed, queries that only utilize existing indexes will not be affected during the index rebuild. You can [track index transformation progress](#).

## Understand which system functions utilize the index

If the expression can be translated into a range of string values, then it can utilize the index; otherwise, it cannot.

Here is the list of string functions that can utilize the index:

- STARTSWITH(str\_expr, str\_expr)
- LEFT(str\_expr, num\_expr) = str\_expr
- SUBSTRING(str\_expr, num\_expr, num\_expr) = str\_expr, but only if first num\_expr is 0

Some common system functions that do not use the index and must load each document are below:

SYSTEM FUNCTION	IDEAS FOR OPTIMIZATION
CONTAINS	Use Azure Search for full text search

SYSTEM FUNCTION	IDEAS FOR OPTIMIZATION
UPPER/LOWER	Instead of using the system function to normalize data each time for comparisons, instead normalize the casing upon insertion. Then a query such as <pre>SELECT * FROM c WHERE UPPER(c.name) = 'BOB'</pre> simply becomes <pre>SELECT * FROM c WHERE c.name = 'BOB'</pre>
Mathematical functions (non-aggregates)	If you need to frequently compute a value in your query, consider storing this value as a property in your JSON document.

Other parts of the query may still utilize the index despite the system functions not using the index.

## Queries with both a filter and an ORDER BY clause

While queries with a filter and an `ORDER BY` clause will normally utilize a range index, they will 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 query modification will ensure that it utilizes 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 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
```

**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 serving 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  $1 \times 10 \times 10 \times 10$  (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)
```

**RU Charge:** 22.17 RUs

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  $1 \times 1 \times 5 \times 5 = 25$  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, it means the query did not have to scan many unnecessary documents. For many queries, such as those that use the TOP keyword, Retrieved Document Count may exceed Output Document Count by 1. This should not be cause for concern.

## Avoid 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 will only need to check the relevant partition's index. This optimization reduces the total number of RU's that the query requires.

If you have a large number of provisioned RU's (over 30,000) or a large amount of data stored (over approximately 100 GB), you likely have a large enough container to see a significant reduction in query RU charges.

For example, if we create a container with the partition key foodGroup, the following queries would only need to check a single physical partition:

```
SELECT * FROM c
WHERE c.foodGroup = "Soups, Sauces, and Gravies" and c.description = "Mushroom, oyster, raw"
```

These queries would also be optimized by including the partition key in the query:

```
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 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"
```

## Filters on multiple properties

While queries with filters on multiple properties will normally utilize a range index, they will be more efficient if they can be served from a composite index. For small amounts of data, this optimization will not have a significant impact. It may prove 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, you should pick one of them that will utilize the composite index. The remainder will continue to utilize 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 which 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 is 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, RU charge may be acceptable but query latency is still too high. The below sections give an overview of tips for reducing query latency. If you run the same query multiple times on the same dataset, it will have the same RU charge each time. However, query latency may vary between query executions.

### Improve proximity

Queries that are run from a different region than the Azure Cosmos DB account will have a higher latency than if they were run inside the same region. For example, if you were running code on your desktop computer, you should expect latency to be tens or hundreds (or more) milliseconds greater than if the query came from a Virtual Machine within the same Azure region as Azure Cosmos DB. It is 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 (RU's). Let's imagine you have a query that consumes 5 RU's of throughput. For example, if you provision 1,000 RU's, you would be able to run that query 200 times per second. If you attempted to run the query when there was not enough throughput available, Azure Cosmos DB would return an HTTP 429 error. Any of the current Core (SQL) API sdk's will automatically retry this query after waiting a brief period. Throttled requests take a longer amount of time, so increasing provisioned throughput can improve query latency. You can observe the [total number of throttled requests](#) in the Metrics blade of the Azure portal.

## Increase MaxConcurrency

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, adjusting the MaxConcurrency to the number of partitions 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 set the MaxConcurrency (or MaxDegreesOfParallelism in older sdk versions) to a high number, and the system chooses 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. The pre-fetching helps in overall latency improvement of a query. Setting the MaxBufferedItemCount limits the number of pre-fetched results. By setting this value to the expected number of results returned (or a higher number), the query can receive maximum benefit from pre-fetching. Setting this value to -1 allows the system to automatically decide the number of items to buffer.

## Next steps

Refer to documents below on how to measure RUs per query, get execution statistics to tune your queries, and more:

- [Get SQL query execution metrics using .NET SDK](#)
- [Tuning query performance with Azure Cosmos DB](#)
- [Performance tips for .NET SDK](#)

# Troubleshoot issues when you use the Java Async SDK with Azure Cosmos DB SQL API accounts

5/10/2019 • 7 minutes to read • [Edit Online](#)

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.

```

@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.

```

// 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.

```

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.

# Diagnose and troubleshoot issues when using Azure Cosmos DB .NET SDK

9/12/2019 • 5 minutes to read • [Edit Online](#)

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](#) 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.

### Requests timeouts

RequestTimeout usually happens when using Direct/TCP, but can happen in Gateway mode. These are the common known causes, and suggestions on how to fix the problem.

- CPU utilization is high, which will cause latency and/or request timeouts. The customer can scale up the host machine to give it more resources, or the load can be distributed across more machines.

- Socket / Port availability might be low. When running in Azure, clients using the .NET SDK can hit Azure SNAT (PAT) port exhaustion. To reduce the chance of hitting this issue, use the latest version 2.x or 3.x of the .NET SDK. This is an example of why it is recommended to always run the latest SDK version.
- Creating multiple DocumentClient instances might lead to connection contention and timeout issues. Follow the [performance tips](#), and use a single DocumentClient instance across an entire process.
- Users sometimes see elevated latency or request timeouts because their collections are provisioned insufficiently, the back-end throttles requests, and the client retries internally without surfacing this to the caller. Check the [portal metrics](#).
- Azure Cosmos DB distributes the overall provisioned throughput evenly across physical partitions. Check portal metrics to see if the workload is encountering a hot [partition key](#). This will cause the aggregate consumed throughput (RU/s) to appear to be under the provisioned RUs, but a single partition consumed throughput (RU/s) will exceed the provisioned throughput.
- Additionally, the 2.0 SDK adds channel semantics to direct/TCP connections. One TCP connection is used for multiple requests at the same time. This can lead to two issues under specific cases:
  - A high degree of concurrency can lead to contention on the channel.
  - 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.
  - If the case falls in any of these two categories (or if high CPU utilization is suspected), these are possible mitigations:
    - Try to scale the application up/out.
    - Additionally, SDK logs can be captured through [Trace Listener](#) to get more details.

## Connection throttling

Connection throttling can happen because of a connection limit on a host machine. Previous to 2.0, clients running in Azure could hit the [Azure SNAT \(PAT\) port exhaustion](#).

## 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.

## 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.

## Request rate too large

'Request rate too large' or error code 429 indicates that your requests are being throttled, because the consumed throughput (RU/s) has exceeded the provisioned throughput. The SDK will automatically retry 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 are getting 429 errors. Review your [partition key](#) to ensure it results in an even distribution of storage and request volume.

## **Slow query performance**

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.

- 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](#)

# How to configure and read the logs when using Azure Functions trigger for Cosmos DB

12/13/2019 • 2 minutes to read • [Edit Online](#)

This article describes how you can configure your Azure Functions environment to send the Azure Functions trigger for Cosmos DB logs to your configured [monitoring solution](#).

## Included 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](#). You have to 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"
```

## Next steps

- [Enable monitoring](#) in your Azure Functions applications.
- Learn how to [Diagnose and troubleshoot common issues](#) when using the Azure Functions trigger for Cosmos DB.

# How to configure the connection policy used by Azure Functions trigger for Cosmos DB

2/25/2020 • 2 minutes to read • [Edit Online](#)

This article describes how you can configure the connection policy when using the Azure Functions trigger for Cosmos DB to connect to your Azure Cosmos account.

## Why is the connection policy important?

There are two connection modes - Direct mode and Gateway mode. To learn more about these connection modes, see the [performance tips](#) 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](#).

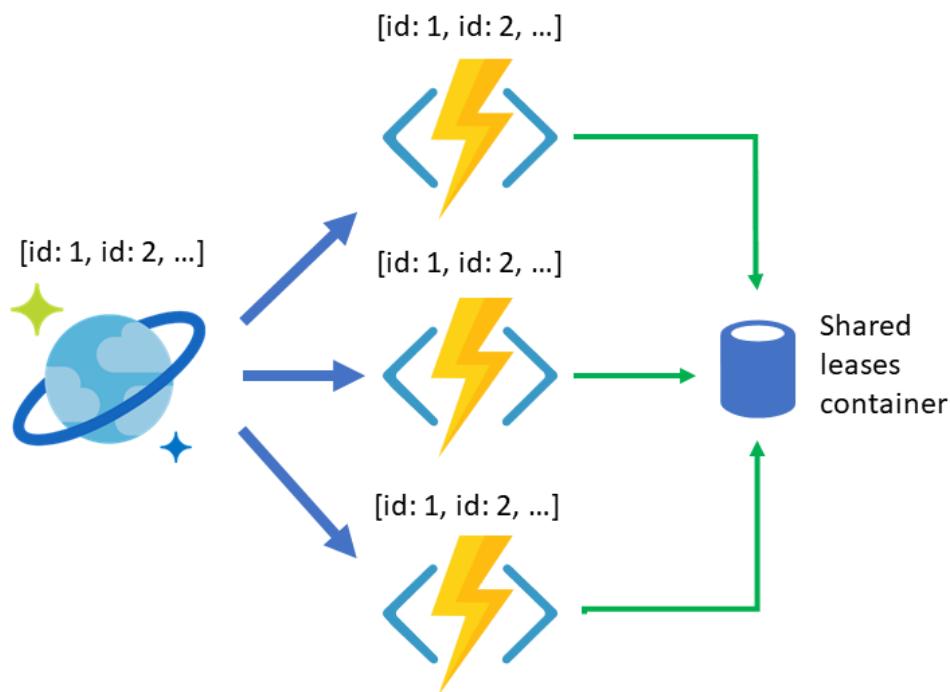
## Next steps

- [Connection limits in Azure Functions](#)
- [Azure Cosmos DB performance tips](#)
- [Code samples](#)

# Create multiple Azure Functions triggers for Cosmos DB

2/25/2020 • 3 minutes to read • [Edit Online](#)

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

2/25/2020 • 7 minutes to read • [Edit Online](#)

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 containers 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 if 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 "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 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 one "stealing" the changes.

One easy way to workaround 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 re-process all the items in my container from the beginning

To re-process 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 (i.e. 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 re-process 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 workaround 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](#)

# How to configure the change feed processor start time

2/24/2020 • 2 minutes to read • [Edit Online](#)

This article describes how you can configure the [change feed processor](#) to start reading from a specific date and time.

## Default behavior

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 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 set up 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.

**NOTE**

When specifying a point in time, only changes for items that currently exist in the container will be read. If an item was deleted, its history on the Change Feed is also removed.

## 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](#)

# Use the change feed estimator

2/24/2020 • 2 minutes to read • [Edit Online](#)

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

2/24/2020 • 2 minutes to read • [Edit Online](#)

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" })
    .WithProcessorOptions(new ChangeFeedProcessorLibrary.ChangeFeedProcessorOptions()
    {
        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)
    .WithInstanceId("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.

### NOTE

Migrations from applications using the library to the .NET V3 SDK are one-way, since the state (leases) will be migrated to the new schema. The migration is not backward compatible.

## 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](#)

# Accelerate big data analytics by using the Apache Spark to Azure Cosmos DB connector

7/19/2019 • 5 minutes to read • [Edit Online](#)

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_pcollections",
    "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_pcollections",
    "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.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	<a href="#">azure-cosmosdb-spark_2.4.0_2.11_1.4.0</a>
2.3.0	2.11	<a href="#">azure-cosmosdb-spark_2.3.0_2.11_1.3.3</a>
2.2.0	2.11	<a href="#">azure-cosmosdb-spark_2.2.0_2.11_1.1.1</a>
2.1.0	2.11	<a href="#">azure-cosmosdb-spark_2.1.0_2.11_1.2.2</a>

## 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

Note, the [Use the Azure Cosmos DB Spark Connector](#) page is currently not up-to-date. Instead of downloading the six separate jars into six different libraries, you can download the uber jar from maven at [https://search.maven.org/artifact/com.microsoft.azure/azure-cosmosdb-spark\\_2.4.0\\_2.11/1.4.0/jar](https://search.maven.org/artifact/com.microsoft.azure/azure-cosmosdb-spark_2.4.0_2.11/1.4.0/jar)) 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 notebooks 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

11/22/2019 • 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

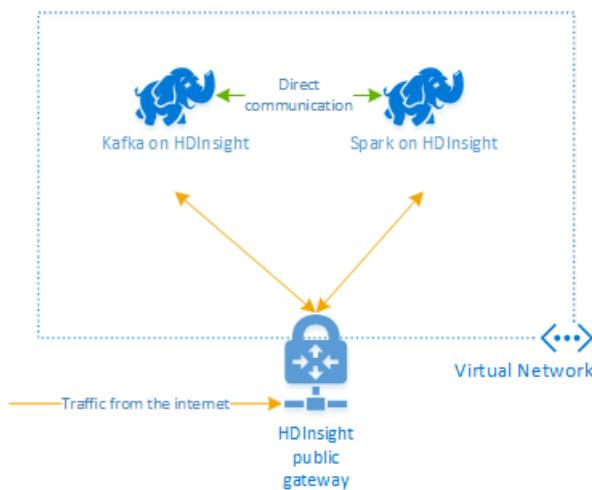
This example used Spark 2.2 on HDInsight 3.6.

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 URLs 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.



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 <b>myhdhi</b> creates a Spark cluster named <b>spark-myhdhi</b> and a Kafka cluster named <b>kafka-myhdhi</b> .
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.

Home > Custom deployment

## Custom deployment

Deploy from a custom template

---

### TEMPLATE

 Customized template  
5 resources

[Edit template](#) [Edit parameters](#) [Learn more](#)

---

### BASICS

\* Subscription:

\* Resource group:  Create new  Use existing  
 

\* Location:  

---

### SETTINGS

\* Cosmos DB Account Name   

\* Base Cluster Name   

Cluster Version 

Cluster Login User Name 

\* Cluster Login Password   

Ssh User Name 

\* Ssh Password   

---

### TERMS AND CONDITIONS

[Azure Marketplace Terms](#) | [Azure Marketplace](#)

By clicking "Purchase," I (a) agree to the applicable legal terms associated with the offering; (b) authorize Microsoft to charge or bill my current payment method for the fees associated with the offering(s), including applicable taxes, with the same billing frequency as my Azure subscription, until I discontinue use of the offering(s); and (c) agree that, if the deployment involves 3rd party offerings, Microsoft may share my contact information and other details of such deployment with the publisher of that offering.

I agree to the terms and conditions stated above

Pin to dashboard

[Purchase](#)

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
"YqPXw3RP7TsJoBF5imkYR0QNA02IrreNA1krUMkL8EW94YHs41bktBhIgWq4pqj6HCGYijQKMRkCTsSaKUO2pw=="
```

#### 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](#)

# Enable notebooks for Azure Cosmos DB accounts (preview)

1/28/2020 • 2 minutes to read • [Edit Online](#)

## 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**.

The screenshot shows the 'Create Azure Cosmos DB Account' page. At the top, there's a breadcrumb navigation: Home > Azure Cosmos DB accounts > Create Azure Cosmos DB Account. Below that is the title 'Create Azure Cosmos DB Account'. A note says: 'Select the subscription to manage deployed resources and costs. Use resource groups like folders to organize and manage all your resources.' There are two dropdown menus for 'Subscription Name' and 'Resource Group Name', both with a 'Create new' link below them. Under 'Instance Details', there are fields for 'Account Name' (set to 'cosmos-notebook-account'), 'API' (set to 'Core (SQL)'), and 'Jupyter Notebooks or Apache Spark' (with 'Notebooks' selected). Other options like 'Notebooks with Apache Spark' and 'None' are shown in a red box. Below these are fields for 'Location' (set to '(US) East US 2'), 'Geo-Redundancy' (with 'Enable' and 'Disable' buttons), and 'Multi-region Writes' (with 'Enable' and 'Disable' buttons). At the bottom, there are three buttons: 'Review + create' (highlighted with a red border), 'Previous', and 'Next: Network'.

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**.

 Your deployment is complete

Deployment name: Microsoft.Azure.CosmosDB-20190923212228  
 Subscription: [Subscription Name](#)  
 Resource group: [Resource Group Name](#)

Start time: 9/23/2019, 9:23:06 PM  
 Correlation ID: 0285b28c-4359-44a6-bfbd-239f2f77d569

^ Deployment details ([Download](#))

RESOURCE	TYPE	STATUS	OPERATION DETAILS
 contoso-account/default	Microsoft.DocumentDB/databaseAccounts/note...	OK	<a href="#">Operation details</a>
 contoso-account	Microsoft.DocumentDb/databaseAccounts	OK	<a href="#">Operation details</a>

^ Next steps

[Go to resource](#)

6. Select **Go to resource** to go to the Azure Cosmos DB account page.

 **contoso-account Quick start**  
 Azure Cosmos DB account

Search (Ctrl+ /)

- Overview
- Activity log
- Access control (IAM)
- Tags
- Diagnose and solve problems
- Quick start**
- Notifications
- Data Explorer
- Settings
- Replicate data globally

Congratulations! Your Azure Cosmos DB account was created.  
 Now, let's connect to it using a sample app:

Choose a platform: .NET, .NET Core, Xamarin, Java, Node.js, Python

- 1 Add a container  
 In Azure Cosmos DB, data is stored in containers.  
[Create 'Items' container](#)  
 Create 'Items' container with 10GB storage capacity and 400 Request Units per second (RU/s) throughput capacity, for up to 400 reads/sec. Estimated hourly bill: \$0.033 USD
- 2 Download and run your .NET app  
 Once container is created, download a sample .NET app connected to it, extract, build and run.  
[Download](#)

7. Navigate to the **Data Explorer** pane. You should now see your notebooks workspace.

 **contoso-account - Data Explorer**  
 Azure Cosmos DB account

Search (Ctrl+ /)

- Overview
- Activity log
- Access control (IAM)
- Tags
- Diagnose and solve problems
- Quick start
- Notifications
- Data Explorer**
- Settings
- Replicate data globally
- Default consistency
- Firewall and virtual networks
- CORS
- Keys
- Add Azure Search
- Add Azure Function
- Locks
- Export template

New Container | New Notebook | Open Terminal | Reset Workspace

**Welcome to Cosmos DB**  
 Globally distributed, multi-model database service for any scale

Common Tasks: Start with Sample, New Container, New Notebook

Recents: New Database

Tips: Data Modeling, Cost & Throughput Calculation, Configure automatic failover

-- See more Cosmos DB documentation

## 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**.

The screenshot shows the 'contoso-account - Data Explorer' blade. On the left, there's a sidebar with icons for Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Quick start, Notifications, and Data Explorer. The 'Data Explorer' icon is highlighted. At the top right, there are buttons for 'New Container' and 'Enable Notebooks'. The 'Enable Notebooks' button is highlighted with a yellow background. Below it, a message says: 'Looks like you have not yet created a notebooks workspace for this account. To proceed and start using notebooks, we'll need to create a default notebooks workspace in this account.' A blue 'Complete setup' button is at the bottom of this section, with a red box drawn around it.

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**.

The screenshot shows the 'contoso-account - Data Explorer' blade with the 'Notebooks' section expanded. Under 'Sample Notebooks (View Only)', the 'GettingStarted.ipynb' file is selected and highlighted with a yellow box. The main pane displays the contents of the notebook, titled 'Getting started with Cosmos notebooks'. It includes a brief introduction, instructions for creating a database and container, and a code cell showing Python code for creating a database and container in Azure Cosmos DB. The code is as follows:

```
[1]: 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 Website Data created')
```

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.

The screenshot shows the Azure Cosmos DB Jupyter notebook interface. At the top, there's a toolbar with 'Save', 'Python 3', 'Run' (highlighted with a yellow circle), 'Clear outputs', 'New Cell', 'Code', 'Copy', and 'Undo'. Below the toolbar, the left sidebar has sections for 'SQL API', 'DATA' (with 'RetailDemo' and 'WebsiteData' expanded), and 'NOTEBOOKS' (with 'Sample Notebooks (View Only)' containing 'GettingStarted.ipynb', 'GlobalDistribution.ipynb', 'Indexing.ipynb', 'RequestUnits.ipynb', and 'Visualization.ipynb'). Under 'My Notebooks/' is another 'GettingStarted.ipynb'. A 'Refresh pane' button is highlighted with a yellow circle. The main content area shows the 'Getting started with Cosmos notebooks' page. A code cell [1] is selected (highlighted with a blue box) and contains Python code for creating a database and container:

```
[1]: 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')

Database RetailDemo created
Container WebsiteData created
```

A 'Select cell' button (highlighted with a yellow circle) is located next to the code cell. The status bar at the bottom says '1 cell'.

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.

The screenshot shows the Azure Cosmos DB Data Explorer interface. At the top, it says 'Home > contoso-account - Data Explorer'. The main area is titled 'contoso-account - Data Explorer' and 'Azure Cosmos DB account'. It has buttons for 'New Container', 'New Notebook' (highlighted with a yellow circle), and 'Open Terminal'. The left sidebar shows 'SQL API' and 'DATA' (with 'RetailDemo' and 'WebsiteData' expanded). The 'NOTEBOOKS' section shows 'Sample Notebooks (View Only)' with files like 'GettingStarted.ipynb', 'GlobalDistribution.ipynb', etc., and 'My Notebooks/' which is expanded. A context menu is open over 'My Notebooks/' with options: 'Refresh' (highlighted with a yellow circle), 'Rename', 'New Directory', 'New Notebook' (highlighted with a yellow circle), and 'Upload File' (highlighted with a green box).

## Next steps

- Learn about the benefits of [Azure Cosmos DB Jupyter notebooks](#)

# Use built-in notebook commands and features in Azure Cosmos DB (preview)

1/22/2020 • 5 minutes to read • [Edit Online](#)

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.

## 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 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
```

## 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.

## 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}
```

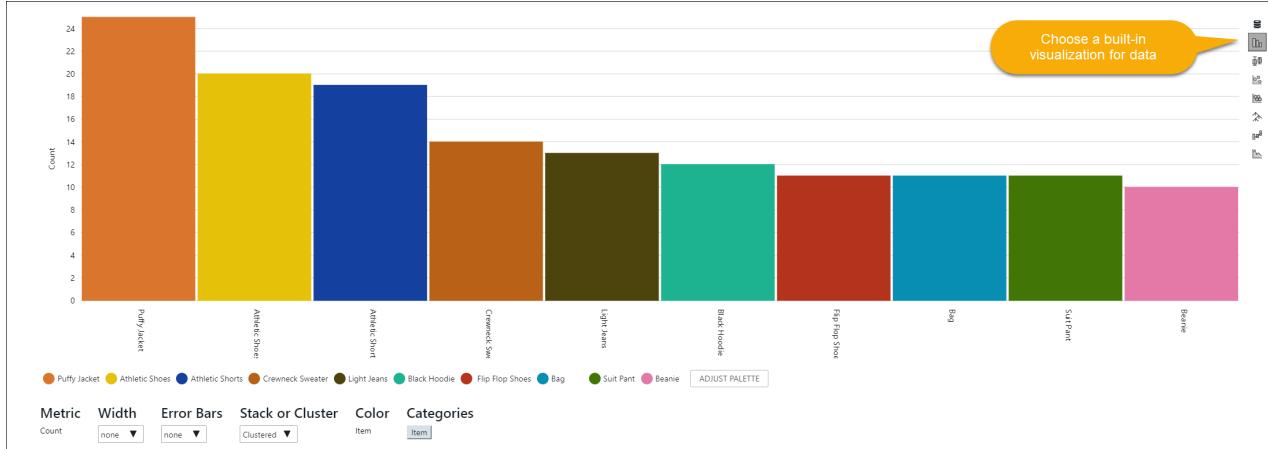
Run `%container?` in a cell to see documentation in the notebook.

## 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 os
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=os.environ["COSMOS_ENDPOINT"], credential=os.environ["COSMOS_KEY"],
connection_policy=custom_connection_policy)
```

## Access the account endpoint and primary key env variables

```
import os

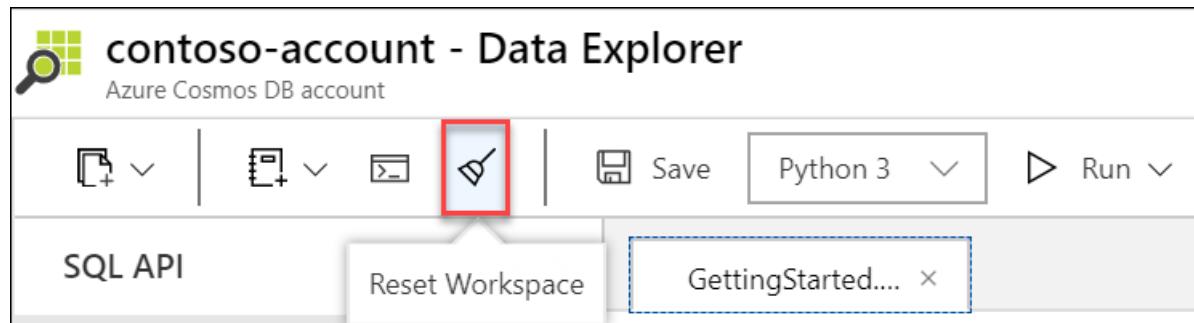
endpoint = os.environ["COSMOS_ENDPOINT"]
primary_key = os.environ["COSMOS_KEY"]
```

#### IMPORTANT

The `COSMOS_ENDPOINT` and `COSMOS_KEY` environment 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](#)

# How to write stored procedures, triggers, and user-defined functions in Azure Cosmos DB

12/13/2019 • 10 minutes to read • [Edit Online](#)

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 applies 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.

### Example

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 a new Azure Cosmos item as input, inserts it into the Azure Cosmos container and returns the ID for the newly created item. In this example, we are leveraging the ToDoList sample from the [Quickstart .NET SQL API](#)

```
function createToDoItem(itemToCreate) {  
  
    var context = getContext();  
    var container = context.getCollection();  
  
    var accepted = container.createDocument(container.getSelfLink(),  
        itemToCreate,  
        function (err, itemCreated) {  
            if (err) throw new Error('Error' + err.message);  
            context.getResponse().setBody(itemCreated.id)  
        });  
    if (!accepted) return;  
}
```

### 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);
        }
    }
}
```

## 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

12/13/2019 • 2 minutes to read • [Edit Online](#)

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

2/24/2020 • 11 minutes to read • [Edit Online](#)

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 = "spCreateToDoItem";
StoredProcedure new.StoredProcedure = new StoredProcedure
{
    Id = storedProcedureId,
    Body = File.ReadAllText(@"..\js\{storedProcedureId}.js")
};
Uri containerUri = UriFactory.CreateDocumentCollectionUri("myDatabase", "myContainer");
var response = await client.CreateStoredProcedureAsync(containerUri, new.StoredProcedure);
StoredProcedure created.StoredProcedure = response.Resource;
```

The following code shows how to call a stored procedure by using the .NET SDK V2:

```
dynamic newItem = new
{
    category = "Personal",
    name = "Groceries",
    description = "Pick up strawberries",
    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, newItem);
```

## Stored procedures - .NET SDK V3

The following example shows how to register a stored procedure by using the .NET SDK V3:

```
StoreProcedureResponse storedProcedureResponse = await client.GetContainer("database",
    "container").Scripts.CreateStoreProcedureAsync(new StoreProcedureProperties
{
    Id = "spCreateToDoItem",
    Body = File.ReadAllText(@"..\js\spCreateToDoItem.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
    }
};

var result = await client.GetContainer("database", "container").Scripts.ExecuteStoreProcedureAsync<string>
("spCreateToDoItem", new PartitionKey("Personal"), 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");
StoreProcedure newStoreProcedure = new StoreProcedure(
    "{" +
        "  'id':'spCreateToDoItem'," +
        "  'body':"+ new String(Files.readAllBytes(Paths.get("../\\js\\spCreateToDoItem.js")))) +
    "}");
//toBlocking() blocks the thread until the operation is complete and is used only for demo.
StoreProcedure createdStoreProcedure = asyncClient.createStoreProcedure(containerLink, newStoreProcedure,
    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, "spCreateToDoItem");
final CountDownLatch successfulCompletionLatch = new CountDownLatch(1);

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;

RequestOptions requestOptions = new RequestOptions();
requestOptions.setPartitionKey(new PartitionKey("Personal"));

Object[] storedProcedureArgs = new Object[] { newItem };
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 = "spCreateToDoItem";
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 = "spCreateToDoItem";
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

```

with open('../js/spCreateToDoItem.js') as file:
    file_contents = file.read()
container_link = ' dbs/myDatabase/colls/myContainer'
sproc_definition = {
    'id': 'spCreateToDoItem',
    'serverScript': file_contents,
}
sproc = client.CreateStoredProcedure(container_link, sproc_definition)

```

The following code shows how to call a stored procedure by using the Python SDK

```

sproc_link = ' dbs/myDatabase/colls/myContainer/sprocs/spCreateToDoItem'
new_item = [{ 
    'category':'Personal',
    'name':'Groceries',
    'description':'Pick up strawberries',
    'isComplete': False
}]
client.ExecuteStoredProcedure(sproc_link, new_item, {'partitionKey': 'Personal'})

```

## 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:

```

with open('../js/trgPreValidateToDoItemTimestamp.js') as file:
    file_contents = file.read()
container_link = 'dbs/myDatabase/colls/myContainer'
trigger_definition = {
    'id': 'trgPreValidateToDoItemTimestamp',
    'serverScript': file_contents,
    'triggerType': documents.TriggerType.Pre,
    'triggerOperation': documents.TriggerOperation.Create
}
trigger = client.CreateTrigger(container_link, trigger_definition)

```

The following code shows how to call a pre-trigger using the Python SDK:

```

container_link = 'dbs/myDatabase/colls/myContainer'
item = {'category': 'Personal', 'name': 'Groceries',
        'description': 'Pick up strawberries', 'isComplete': False}
client.CreateItem(container_link, item, {
    'preTriggerInclude': '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:

```

with open('../js/trgPostUpdateMetadata.js') as file:
    file_contents = file.read()
container_link = 'dbs/myDatabase/colls/myContainer'
trigger_definition = {
    'id': 'trgPostUpdateMetadata',
    'serverScript': file_contents,
    'triggerType': documents.TriggerType.Post,
    'triggerOperation': documents.TriggerOperation.Create
}
trigger = client.CreateTrigger(container_link, trigger_definition)

```

The following code shows how to call a post-trigger using the Python SDK:

```

container_link = 'dbs/myDatabase/colls/myContainer'
item = {'name': 'artist_profile_1023', 'artist': 'The Band',
        'albums': ['Hellujah', 'Rotators', 'Spinning Top']}
client.CreateItem(container_link, item, {
    'postTriggerInclude': 'trgPostUpdateMetadata'})

```

## 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:

```
with open('../js/udfTax.js') as file:
    file_contents = file.read()
container_link = 'dbs/myDatabase/colls/myContainer'
udf_definition = {
    'id': 'Tax',
    'serverScript': file_contents,
}
udf = client.CreateUserDefinedFunction(container_link, udf_definition)
```

The following code shows how to call a user-defined function using the Python SDK:

```
container_link = 'dbs/myDatabase/colls/myContainer'
results = list(client.QueryItems(
    container_link, "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](#)

# Configure IP firewall in Azure Cosmos DB

11/4/2019 • 7 minutes to read • [Edit Online](#)

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**.

The screenshot shows the Azure portal interface for managing a Cosmos DB account named 'cdbaccountvnet'. The left sidebar contains navigation links like Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Quick start, Data Explorer, and several settings options. The 'Firewall and virtual networks' link is highlighted with a red box. The main content area displays the 'Firewall and virtual networks' configuration page. It shows the 'Allow access from' setting is set to 'Selected networks' (radio button selected). Below this, there's a section for 'Virtual networks' with buttons for 'Add existing virtual network' and 'Add new virtual network'. The 'Firewall' section allows adding IP ranges, with a note that no IP range filter is configured. The 'Exceptions' section has two checked checkboxes: 'Accept connections from within public Azure datacenters' and 'Allow access from Azure Portal'. At the bottom, there are 'Save' and 'Discard' buttons, both of which are also highlighted with red boxes.

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 access to the Azure portal by selecting the **Allow access from Azure portal** option, as shown in the following screenshot:

The screenshot shows the Azure portal interface for managing network security. On the left, there's a sidebar with various options like Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Quick start, and Data Explorer. The 'Firewall and virtual networks' option is highlighted with a red box. The main content area has a title 'cdbaccountvnet - Firewall and virtual networks'. It shows the 'Allow access from' section with 'Selected networks' selected (radio button highlighted with a red box). Below it, there's a note about configuring network security for the Azure Cosmos DB account. The 'Virtual networks' section shows a table with columns VIRTUAL NETWO..., SUBNET, ADDRESS RANGE, ENDPOINT STAT..., RESOURCE GRO..., and SUBSCRIPTION. A message says 'No network selected.' The 'Firewall' section allows adding IP ranges or CIDR ranges, with a note that no range is currently added. The 'IP (SINGLE IPV4 OR CIDR RANGE)' field is empty. The 'Exceptions' section contains two checkboxes: 'Accept connections from within public Azure datacenters' (checked) and 'Allow access from Azure Portal' (checked and highlighted with a red box). At the bottom, there are 'Save' and 'Discard' buttons.

### 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. To allow access to the Azure Cosmos DB account from such services, add the IP address 0.0.0.0 to the list of allowed IP addresses. The 0.0.0.0 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.

You can enable access to the Azure portal by selecting the **Accept connections from within Azure datacenters** option, as shown in the following screenshot:

The screenshot shows the 'Firewall and virtual networks' section of the Azure Cosmos DB settings. On the left, a sidebar lists various management options. The 'Firewall and virtual networks' item is highlighted with a red box. On the main page, there are two radio button options for 'Allow access from': 'All networks' (unchecked) and 'Selected networks' (checked). Below this, a note says 'Configure network security for your Azure Cosmos DB account. [Learn more.](#)'. Under 'Virtual networks', there's a table header with columns: VIRTUAL NETWO..., SUBNET, ADDRESS RANGE, ENDPOINT STAT..., RESOURCE GRO..., and SUBSCRIPTION. A note below says 'No network selected.'. Under 'Firewall', there's a note 'Add IP ranges to allow access from the internet or your on-premises networks.' followed by '+ Add my current IP ( )'. Below this is a section for 'IP (SINGLE IPV4 OR CIDR RANGE)' with a text input field and an ellipsis button. A note says 'No IP range filter configured.'. Under 'Exceptions', there are two checked checkboxes: 'Accept connections from within public Azure datacenters' (with a note) and 'Allow access from Azure Portal' (with a note). At the bottom are 'Save' and 'Discard' buttons.

#### 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**.

Home > cdbaccountvnet - Firewall and virtual networks

cdbaccountvnet - Firewall and virtual networks  
Azure Cosmos DB account

Allow access from  
 All networks  Selected networks

Configure network security for your Azure Cosmos DB account. [Learn more.](#)

Virtual networks  
Secure your Azure Cosmos DB account with virtual networks. [+ Add existing virtual network](#) [+ Add new virtual network](#)

VIRTUAL NETWO...	SUBNET	ADDRESS RANGE	ENDPOINT STAT...	RESOURCE GRO...	SUBSCRIPTION
No network selected.					

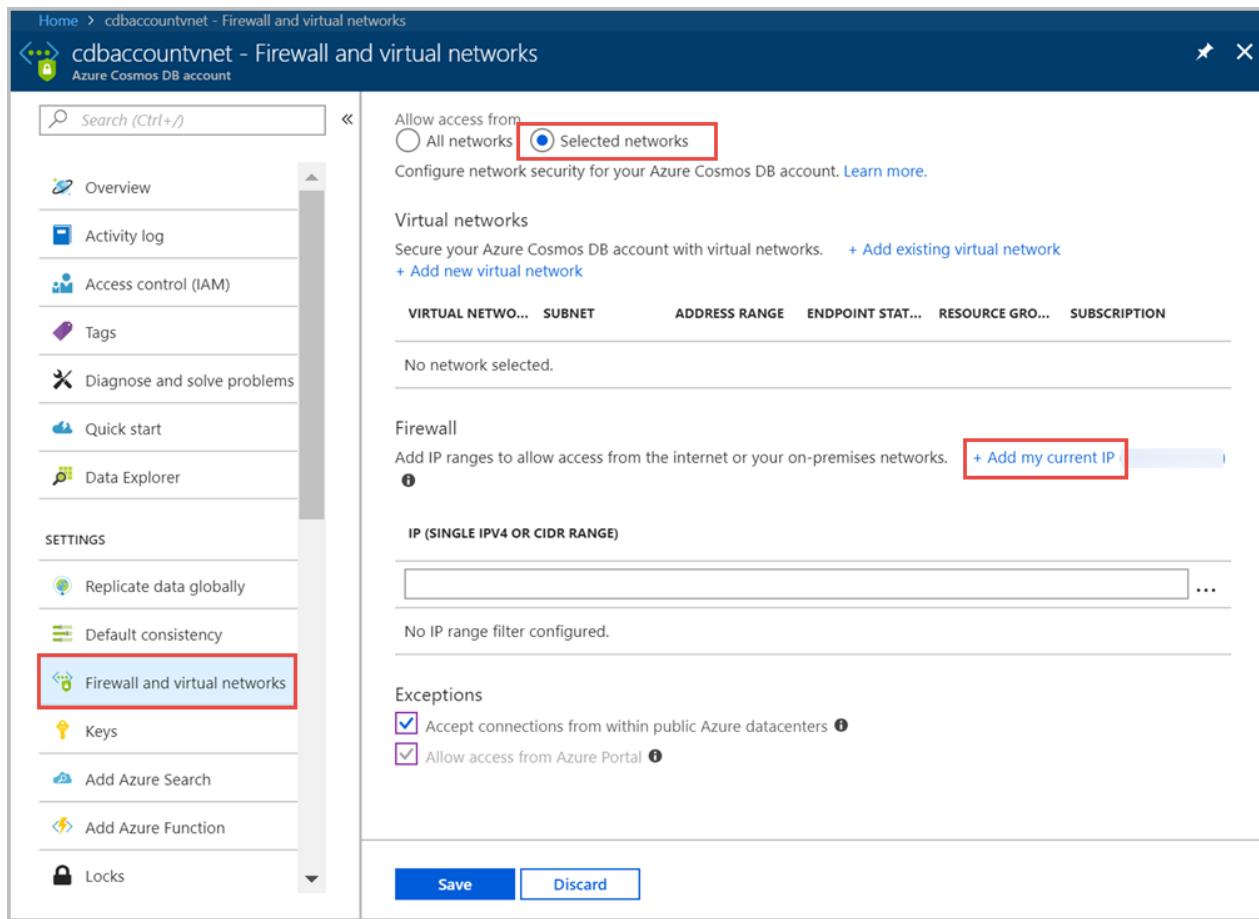
Firewall  
Add IP ranges to allow access from the internet or your on-premises networks. [+ Add my current IP](#)

IP (SINGLE IPV4 OR CIDR RANGE)

No IP range filter configured.

Exceptions  
 Accept connections from within public Azure datacenters [?](#)  
 Allow access from Azure Portal [?](#)

Save Discard



## 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:

The screenshot shows the Azure portal interface for managing a cloud service. On the left, there's a navigation menu with options like Overview, Activity log, Access control (IAM), Settings (which includes Antimalware, Certificates, Configuration, Extensions, Remote Desktop, Scale, and Properties), and Monitoring (with Alert rules). The 'Properties' tab is currently selected, indicated by a blue highlight. In the main pane, detailed information about the service is displayed. Key details include:

- CURRENT SLOT: Production
- STATUS: Running
- SITE URL: http://telcoworkernortheastasia.cloudapp....
- NAME: 648e876725b946ec8f14ebadd4b82f92
- LABEL: ContosoTelcoWorker - 9/18/2016 4:41:35...
- DEPLOYMENT ID: d00bc8996e5240a793692432d0137ef1
- PUBLIC IP ADDRESSES: 13.74.156.31 (highlighted with a red box)
- INPUT ENDPOINTS: TelcoMetricWorker: 13.74.156.31:3389

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:

The screenshot shows the Azure portal interface for managing a virtual machine. On the left, there's a navigation menu with options like Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Settings (with Availability set and Disks), and Disks. The 'Properties' tab is currently selected, indicated by a blue highlight. In the main pane, detailed information about the virtual machine is displayed. Key details include:

- STATUS: Running
- COMPUTER NAME: cache
- PUBLIC IP ADDRESS/DNS NAME LABEL: 13.75.145.220/<none> (highlighted with a red box)
- PRIVATE IP ADDRESS: 10.0.0.4
- OPERATING SYSTEM: Windows

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 **ipRangeFilter** attribute with a list 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](#)

```
{  
    "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"
$ipRangeFilter = "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;
    "ipRangeFilter"=$ipRangeFilter
}

New-AzResource -ResourceType "Microsoft.DocumentDb/databaseAccounts" ` 
    -ApiVersion "2015-04-08" -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.

## 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 from virtual networks (VNet)

1/14/2020 • 10 minutes to read • [Edit Online](#)

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.

## 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 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**.

**Add networks**

Allow access from  
Selected networks

Virtual networks

VIRTUAL NETWORK	SUBNET	ADDRESS RANGE	ENDPOINT STATUS	RESOURCE GROUP	SUBSCRIPTION
demo-vnet	subnet1	10.23.0.0/16	Enabled	demo	CosmosDB-Networking... ...

Firewall

No IP range filter configured.

Exceptions

Allow access from Azure Portal

**Add**

- 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:

**Add networks**

Allow access from  
Selected networks

Virtual networks

VIRTUAL NETWORK	SUBNET	ADDRESS RANGE	ENDPOINT STATUS	RESOURCE GROUP	SUBSCRIPTION
demo-vnet	1	10.23.0.0/16	Enabled	demo	CosmosDB-Networking... ...
demo-vnet	subnet1	10.23.1.0/24	Enabled	demo	CosmosDB-Networking... ...

Firewall

No IP range filter configured.

#### 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

- From the **All resources** blade, find the Azure Cosmos DB account that you want to secure.
- Select **Firewalls and Azure virtual networks** from the settings menu, and choose to allow access from **Selected networks**.
- To grant access to a new Azure virtual network, under **Virtual networks**, select **Add new virtual network**.

- 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.

The screenshot shows the Azure portal's 'Firewall and virtual networks' blade for the 'demo-db' account. On the left, there's a sidebar with various settings like Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Quick start, Notifications, Data Explorer, and Settings. Under Settings, 'Firewall and virtual networks' is selected. The main area shows a table of existing virtual networks ('demo-vnet') and their subnets ('demo-vnet/subnet1'). A 'Create virtual network' dialog is open on the right, with its contents highlighted by a red box. The dialog fields include:

- Name:** demo-vnet2
- Address space:** 10.23.0.0/16 (10.23.0.0 - 10.23.255.255 (65536 addresses))
- Subscription:** CosmosDB-Networking-VNET\_TestClient
- Resource group:** demo
- Location:** Central US
- Subnet:**
  - Name:** subnet1
  - Address range:** 10.23.0.0/24 (10.23.0.0 - 10.23.255 (256 addresses))
- DDoS protection:** Basic
- Service endpoint:** Microsoft.AzureCosmosDB
- Firewall:** Disabled (button labeled '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

- From the **All resources** blade, find the Azure Cosmos DB account for which you assigned service endpoints.
- Select **Firewalls and virtual networks** from the settings menu.
- To remove a virtual network or subnet rule, select ... next to the virtual network or subnet, and select **Remove**.

The screenshot shows the 'Firewalls and virtual networks' blade for the 'demo-db' account. It lists two virtual networks: 'demo-vnet' and 'demo-vnet2'. Each entry includes a 'SUBSCRIPTION' column. For 'demo-vnet2', the 'More options' button (three dots) is highlighted with a red box. A dropdown menu appears, with the 'Remove' option also highlighted with a red box.

VIRTUAL NETWORK	SUBNET	ADDRESS RANGE	ENDPOINT STATUS	RESOURCE GROUP	SUBSCRIPTION
demo-vnet	1	10.23.0.0/16	✓ Enabled	demo	CosmosDB-Networking-VNET... ...
demo-vnet2	1	10.23.0.0/16	✓ Enabled	demo	CosmosDB-Networking-VNET... ...

- Select **Save** to apply your changes.

## Configure a service endpoint by using Azure PowerShell

**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.

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.

```
$rgname = "<Resource group name>"  
$vnName = "<Virtual network name>"  
$sname = "<Subnet name>"  
$subnetPrefix = "<Subnet address range>"  
  
Get-AzVirtualNetwork `  
-ResourceGroupName $rgname `  
-Name $vnName | Set-AzVirtualNetworkSubnetConfig `  
-Name $sname `  
-AddressPrefix $subnetPrefix `  
-ServiceEndpoint "Microsoft.AzureCosmosDB" | Set-AzVirtualNetwork
```

3. Get virtual network information.

```
$vnProp = Get-AzVirtualNetwork `  
-Name $vnName `  
-ResourceGroupName $rgName
```

4. Get properties of the Azure Cosmos DB account by running the following cmdlet:

```
$apiVersion = "2015-04-08"  
$acctName = "<Azure Cosmos DB account name>"  
  
$cosmosDBConfiguration = Get-AzResource `  
-ResourceType "Microsoft.DocumentDB/databaseAccounts" `  
-ApiVersion $apiVersion `  
-ResourceGroupName $rgName `  
-Name $acctName
```

5. Initialize the variables for use later. Set up all the variables from the existing account definition.

```

$locations = @()

foreach ($readLocation in $cosmosDBConfiguration.Properties.readLocations) {
    $locations += , @{
        locationName      = $readLocation.locationName;
        failoverPriority = $readLocation.failoverPriority;
    }
}

$virtualNetworkRules = @(@{
    id = "$($vnProp.Id)/subnets/$sname";
})

if ($cosmosDBConfiguration.Properties.isVirtualNetworkFilterEnabled) {
    $virtualNetworkRules = $cosmosDBConfiguration.Properties.virtualNetworkRules + $virtualNetworkRules
}

```

6. Update Azure Cosmos DB account properties with the new configuration by running the following cmdlets:

```

$cosmosDBProperties = @{
    databaseAccountOfferType      = $cosmosDBConfiguration.Properties.databaseAccountOfferType;
    consistencyPolicy             = $cosmosDBConfiguration.Properties.consistencyPolicy;
    ipRangeFilter                 = $cosmosDBConfiguration.Properties.ipRangeFilter;
    locations                     = $locations;
    virtualNetworkRules           = $virtualNetworkRules;
    isVirtualNetworkFilterEnabled = $True;
}

Set-AzResource ` 
    -ResourceType "Microsoft.DocumentDB/databaseAccounts" ` 
    -ApiVersion $apiVersion ` 
    -ResourceGroupName $rgName ` 
    -Name $acctName ` 
    -Properties $CosmosDBProperties

```

7. 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:

```

$UpdatedcosmosDBConfiguration = Get-AzResource ` 
    -ResourceType "Microsoft.DocumentDB/databaseAccounts" ` 
    -ApiVersion $apiVersion ` 
    -ResourceGroupName $rgName ` 
    -Name $acctName

$UpdatedcosmosDBConfiguration.Properties

```

## 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

```

## Migrating from an IP firewall rule to a virtual network ACL

Use the following steps only for Azure Cosmos DB accounts with existing IP firewall rules that allow a subnet, when you want to use virtual network and subnet-based ACLs instead of an IP firewall rule.

After a service endpoint for an Azure Cosmos DB account is turned on for a subnet, the requests are sent with a source that contains virtual network and subnet information instead of a public IP. These requests don't match an IP filter. This source switch happens for all Azure Cosmos DB accounts accessed from the subnet with a service endpoint enabled. To prevent downtime, use the following steps:

- Get properties of the Azure Cosmos DB account by running the following cmdlet:

```
$apiVersion = "2015-04-08"
$acctName = "<Azure Cosmos DB account name>"

$cosmosDBConfiguration = Get-AzResource ` 
    -ResourceType "Microsoft.DocumentDB/databaseAccounts" ` 
    -ApiVersion $apiVersion ` 
    -ResourceGroupName $rgName ` 
    -Name $acctName
```

- Initialize the variables to use them later. Set up all the variables from the existing account definition. Add the virtual network ACL to all Azure Cosmos DB accounts being accessed from the subnet with `ignoreMissingVNetServiceEndpoint` flag.

```
$locations = @()

foreach ($readLocation in $cosmosDBConfiguration.Properties.readLocations) {
    $locations += , @{
        locationName      = $readLocation.locationName;
        failoverPriority = $readLocation.failoverPriority;
    }
}

$subnetID = "Subnet ARM URL" e.g "/subscriptions/f7ddba26-ab7b-4a36-a2fa-7d01778da30b/resourceGroups/testrg/providers/Microsoft.Network/virtualNetworks/testvnet/subnets/subnet1"

$virtualNetworkRules = @(@{
    id = $subnetID;
    ignoreMissingVNetServiceEndpoint = "True";
})

if ($cosmosDBConfiguration.Properties.isVirtualNetworkFilterEnabled) {
    $virtualNetworkRules = $cosmosDBConfiguration.Properties.virtualNetworkRules + $virtualNetworkRules
}
```

- Update Azure Cosmos DB account properties with the new configuration by running the following cmdlets:

```
$cosmosDBProperties = @{
    databaseAccountOfferType      = $cosmosDBConfiguration.Properties.databaseAccountOfferType;
    consistencyPolicy             = $cosmosDBConfiguration.Properties.consistencyPolicy;
    ipRangeFilter                 = $cosmosDBConfiguration.Properties.ipRangeFilter;
    locations                     = $locations;
    virtualNetworkRules           = $virtualNetworkRules;
    isVirtualNetworkFilterEnabled = $True;
}

Set-AzResource ` 
    -ResourceType "Microsoft.DocumentDB/databaseAccounts" ` 
    -ApiVersion $apiVersion ` 
    -ResourceGroupName $rgName ` 
    -Name $acctName ` 
    -Properties $CosmosDBProperties
```

- Repeat steps 1-3 for all Azure Cosmos DB accounts that you access from the subnet.
- Wait 15 minutes, and then update the subnet to enable the service endpoint.
- Enable the service endpoint for an existing subnet of a virtual network.

```
$rgname= "<Resource group name>"  
$vnName = "<virtual network name>"  
$sname = "<Subnet name>"  
$subnetPrefix = "<Subnet address range>"  
  
Get-AzVirtualNetwork `  
-ResourceGroupName $rgname `  
-Name $vnName | Set-AzVirtualNetworkSubnetConfig `  
-Name $sname `  
-AddressPrefix $subnetPrefix `  
-ServiceEndpoint "Microsoft.AzureCosmosDB" | Set-AzVirtualNetwork
```

7. Remove the IP firewall rule for the subnet.

## Next steps

- To configure a firewall for Azure Cosmos DB, see the [Firewall support](#) article.

# Configure Azure Private Link for an Azure Cosmos account (preview)

11/17/2019 • 17 minutes to read • [Edit Online](#)

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.

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**:

Connection name	Connection state	Private endpoint	Description
cdbPrivateEndpoint5	Approved	<a href="#">cdbPrivateEndpoint5</a>	
PrivateEndpointPortal	Approved	<a href="#">PrivateEndpointPortal</a>	
cdbPrivateEndpointPS	Approved	<a href="#">cdbPrivateEndpointPS</a>	

3. In the **Create a private endpoint (Preview) - Basics** pane, enter or select the following details:

SETTING	VALUE
<b>Project details</b>	
Subscription	Select your subscription.
Resource group	Select a resource group.
<b>Instance details</b>	
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 <b>Connect to an Azure resource in my directory</b> .  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.
Subscription	Select your subscription.
Resource type	Select <b>Microsoft.AzureCosmosDB/databaseAccounts</b> .
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 <b>Sql</b> because these APIs are interoperable with the SQL API.

6. Select **Next: Configuration**.

7. In **Create a private endpoint (Preview) - Configuration**, enter or select this information:

SETTING	VALUE
<b>Networking</b>	
Virtual network	Select your virtual network.
Subnet	Select your subnet.

SETTING	VALUE
<b>Private DNS Integration</b>	
Integrate with private DNS zone	Select <b>Yes</b> .  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 <b>privatelink.documents.azure.com</b> .  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.

The screenshot shows the Azure portal interface for managing a private endpoint. The left sidebar lists navigation options like Overview, Activity log, Access control (IAM), Tags, Settings (Properties, Locks, Export template), Monitoring, and Metrics. The main content area displays the details for 'cdbPrivateEndpoint3'. Key settings shown include the Resource group (cdbg), Location (East US), Subscription (Content Testing), Subscription ID (<Your subscription ID>), Provisioning state (Succeeded), Virtual network/subnet (cdbVNet/cdbSubnet), Network interface (cdbPrivateEndpoint3.nic.039046f0-efe8-4165-8570-95...), Private link resource (sqldb2), Target sub-resource (Sql), Connection status (Approved), and Request/Response (None). A 'Custom DNS settings' section is expanded, showing two entries: 'sqldb2.documents.azure.com' and 'sqldb2-eastus2.documents.azure.com', each associated with a specific Private IP address (10.1.255.6 and 10.1.255.7 respectively).

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"  
  
$cosmosDbResourceId =  
"/subscriptions/$($SubscriptionId)/resourceGroups/$($ResourceGroupName)/providers/Microsoft.DocumentDB/database  
Accounts/$($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 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"

$cosmosDbResourceId =
"/subscriptions/$($SubscriptionId)/resourceGroups/$($ResourceGroupName)/providers/Microsoft.DocumentDB/database
Accounts/$($CosmosDbAccountName)"
$VNetResourceId =
"/subscriptions/$($SubscriptionId)/resourceGroups/$($ResourceGroupName)/providers/Microsoft.Network/virtualNetw
orks/$($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 : PrivateCosmosDbEndpointDeployment
DeploymentName   :
CorrelationId    :
Provisioningstate : Succeeded
Timestamp        : 10/22/2019 6:03:14 PM
Mode             : Incremental
TemplateLink    :
TemplateLinkString:
DeploymentLogLevel:
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/
g/providers/Microsoft.DocumentDB/databaseAccounts/sqlcdb2
groupId          String        Sql
subnetid          String        /subscriptions/
g/providers/Microsoft.Network/virtualNetworks/cdbvnet/subnets/cdbsubnet

outputs          :
outputsString    : {[privateEndpointNetworkInterface, Microsoft.Azure.Commands.ResourceManager.Cmdlets.SdkModels.Deploymentvariable]}
Name              Type          Value
=====  =====  =====
privateEndpointNetworkInterface String        /subscriptions/
resourceGroups/cdbrg/providers/Microsoft.Network/networkInterfaces/cdbPrivateEndpoint3.nic.039046f0-efe8-4165-8570-9557ebb8cccd3

```

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"

$cosmosDbResourceId =
"/subscriptions/$($SubscriptionId)/resourceGroups/$($ResourceGroupName)/providers/Microsoft.DocumentDB/database
Accounts/$($CosmosDbAccountName)"
$VNetResourceId =
"/subscriptions/$($SubscriptionId)/resourceGroups/$($ResourceGroupName)/providers/Microsoft.Network/virtualNetw
orks/$($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
```

```

$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 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.

## Update a private endpoint when you add or remove a region

Adding or removing regions to an Azure Cosmos account requires you to add or remove DNS entries for that account. Update these changes accordingly in the private endpoint by using the following steps:

1. When the Azure Cosmos DB administrator adds or removes regions, the network administrator gets a notification about the pending changes. For the private endpoint mapped to an Azure Cosmos account, the value of the `ActionsRequired` property changes from `None` to `Recreate`. Then the network administrator updates the private endpoint by issuing a PUT request with the same Resource Manager payload that was used to create it.
2. After the private endpoint is updated, 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. By default, the new region is not accessible from the existing private endpoint. The Azure Cosmos account administrator should refresh the private endpoint connection before accessing it from the new region.

When you run the

```
Get-AzPrivateEndpoint -Name <your private endpoint name> -ResourceGroupName <your resource group name>
```

command, the output of the command contains the `actionsRequired` parameter. This parameter is set to `Recreate`. This value indicates that the private endpoint should be refreshed. Next, the Azure Cosmos account administrator runs the `Set-AzPrivateEndpoint` command to trigger the private endpoint refresh.

```
$pe = Get-AzPrivateEndpoint -Name <your private endpoint name> -ResourceGroupName <your resource group name>
Set-AzPrivateEndpoint -PrivateEndpoint $pe
```

A new private IP is automatically reserved in the subnet under this private endpoint. The value for `actionsRequired` becomes `None`. If you don't have any private DNZ zone integration (in other words, if you're using a custom private DNS zone), you have to configure your private DNS zone to add a new DNS record for the private IP that corresponds to the new region.

You can use the same steps when you remove a region. The private IP of the removed region is automatically reclaimed, and the `actionsRequired` flag becomes `None`. If you don't have any private DNZ zone integration, you must configure your private DNS zone to remove the DNS record for the removed region.

DNS records in the private DNS zone are not removed automatically when a private endpoint is deleted or a region from the Azure Cosmos account is removed. You must manually remove the DNS records.

## Current limitations

The following limitations apply when you're using Private Link with an Azure Cosmos account:

- Private Link support for Azure Cosmos accounts and virtual networks is available in specific regions only. For a list of supported regions, see the [Available regions](#) section of the Private Link article.

### NOTE

To create a private endpoint, make sure that both the virtual network and the Azure Cosmos account are in supported regions.

- When you're using Private Link with an Azure Cosmos account by using a direct mode connection, you can use only the TCP protocol. The HTTP protocol is not yet 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 the Azure Cosmos DB's API for MongoDB accounts that have Private Link, you can't use tools such as Robo 3T, Studio 3T, and Mongoose. The endpoint can have Private Link support only if the `appName=<account name>` parameter is specified. An example is `replicaSet=globaldb&appName=mydbaccountname`. Because these tools don't pass the app name in the connection string to the service, you can't use Private Link. But you can still access these accounts by using SDK drivers with the 3.6 version.

- You can't move or delete a virtual network if it contains Private Link.
- You can't delete an Azure Cosmos account if it's attached to a private endpoint.
- You can't fail over an Azure Cosmos account to a region that's not mapped to all private endpoints attached to the account.
- A network administrator should be granted at least the "/PrivateEndpointConnectionsApproval" permission at the Azure Cosmos account scope to create automatically approved private endpoints.

### Limitations to private DNS zone integration

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.

# Connect privately to an Azure Cosmos account using Azure Private Link

1/15/2020 • 4 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 privately with Private Link resources.

In this article, you will learn how to create a VM on an Azure virtual network and an Azure Cosmos account with a Private Endpoint using the Azure portal. Then, you can securely access the Azure Cosmos account from the VM.

## Sign in to Azure

Sign in to the [Azure portal](#).

## Create a VM

### Create the virtual network

In this section, you will create a virtual network and the subnet to host the VM that is used to access your Private Link resource (an Azure Cosmos account in this example).

1. On the upper-left side of the screen, select **Create a resource > Networking > Virtual network**.
2. In **Create virtual network**, enter or select this information:

SETTING	VALUE
Name	Enter <i>MyVirtualNetwork</i> .
Address space	Enter <i>10.1.0.0/16</i> .
Subscription	Select your subscription.
Resource group	Select <b>Create new</b> , enter <i>myResourceGroup</i> , then select <b>OK</b> .
Location	Select <b>WestCentralUS</b> .
Subnet - Name	Enter <i>mySubnet</i> .
Subnet - Address range	Enter <i>10.1.0.0/24</i> .

3. Leave the rest as default and select **Create**.

### Create the virtual machine

1. On the upper-left side of the screen in the Azure portal, select **Create a resource > Compute > Virtual machine**.
2. In **Create a virtual machine - Basics**, enter or select this information:

SETTING	VALUE
<b>PROJECT DETAILS</b>	
Subscription	Select your subscription.
Resource group	Select <b>myResourceGroup</b> . You created this in the previous section.
<b>INSTANCE DETAILS</b>	
Virtual machine name	Enter <i>myVm</i> .
Region	Select <b>WestCentralUS</b> .
Availability options	Leave the default <b>No infrastructure redundancy required</b> .
Image	Select <b>Windows Server 2019 Datacenter</b> .
Size	Leave the default <b>Standard DS1 v2</b> .
<b>ADMINISTRATOR ACCOUNT</b>	
Username	Enter a username of your choice.
Password	Enter a password of your choice. The password must be at least 12 characters long and meet the <a href="#">defined complexity requirements</a> .
Confirm Password	Reenter the password.
<b>INBOUND PORT RULES</b>	
Public inbound ports	Leave the default <b>None</b> .
<b>SAVE MONEY</b>	
Already have a Windows license?	Leave the default <b>No</b> .

3. Select **Next: Disks**.
4. In **Create a virtual machine - Disks**, leave the defaults and select **Next: Networking**.
5. In **Create a virtual machine - Networking**, select this information:

SETTING	VALUE
Virtual network	Leave the default <b>MyVirtualNetwork</b> .
Address space	Leave the default <b>10.1.0.0/24</b> .
Subnet	Leave the default <b>mySubnet (10.1.0.0/24)</b> .

SETTING	VALUE
Public IP	Leave the default ( <b>new</b> ) <b>myVm-ip</b> .
Public inbound ports	Select <b>Allow selected ports</b> .
Select inbound ports	Select <b>HTTP</b> and <b>RDP</b> .

6. Select **Review + create**. You're taken to the **Review + create** page where Azure validates your configuration.
7. When you see the **Validation passed** message, select **Create**.

## Create an Azure Cosmos account

Create an [Azure Cosmos SQL API account](#). For simplicity, you can create the Azure Cosmos account in the same region as the other resources (that is "WestCentralUS").

## Create a Private Endpoint for your Azure Cosmos account

Create a Private Link for your Azure Cosmos account as described in the [Create a Private Link using the Azure portal](#) section of the linked article.

## Connect to a VM from the internet

Connect to the VM *myVm* from the internet as follows:

1. In the portal's search bar, enter *myVm*.
2. Select the **Connect** button. After selecting the **Connect** button, **Connect to virtual machine** opens.
3. Select **Download RDP File**. Azure creates a Remote Desktop Protocol (.rdp) file and downloads it to your computer.
4. Open the downloaded .rdp file.
  - a. If prompted, select **Connect**.
  - b. Enter the username and password you specified when creating the VM.

### NOTE

You may need to select **More choices > Use a different account**, to specify the credentials you entered when you created the VM.

5. Select **OK**.
6. You may receive a certificate warning during the sign-in process. If you receive a certificate warning, select **Yes** or **Continue**.
7. Once the VM desktop appears, minimize it to go back to your local desktop.

## Access the Azure Cosmos account privately from the VM

In this section, you will connect privately to the Azure Cosmos account using the Private Endpoint.

- To include the IP address and DNS mapping, sign into your Virtual machine *myVM*, open the `c:\Windows\System32\Drivers\etc\hosts` file and include the DNS information from previous step in the following format:

[Private IP Address] [Account endpoint].documents.azure.com

**Example:**

10.1.255.13 mycosmosaccount.documents.azure.com

10.1.255.14 mycosmosaccount-eastus.documents.azure.com

- In the Remote Desktop of *myVM*, install [Microsoft Azure Storage Explorer](#).
- Select **Cosmos DB Accounts (Preview)** with the right-click.
- Select **Connect to Cosmos DB**.
- Select **API**.
- Enter the connection string by pasting the information previously copied.
- Select **Next**.
- Select **Connect**.
- Browse the Azure Cosmos databases and containers from *mycosmosaccount*.
- (Optionally) add new items to *mycosmosaccount*.
- Close the remote desktop connection to *myVM*.

## Clean up resources

When you're done using the Private Endpoint, Azure Cosmos account and the VM, delete the resource group and all of the resources it contains:

- Enter *myResourceGroup* in the **Search** box at the top of the portal and select *myResourceGroup* from the search results.
- Select **Delete resource group**.
- Enter *myResourceGroup* for **TYPE THE RESOURCE GROUP NAME** and select **Delete**.

## Next steps

In this article, you created a VM on a virtual network, an Azure Cosmos account and a Private Endpoint. You connected to the VM from the internet and securely communicated to the Azure Cosmos account using Private Link.

- To learn more about Private Endpoint, see [What is Azure Private Endpoint?](#).
- To learn more about limitation of Private Endpoint when using with Azure Cosmos DB, see [Azure Private Link with Azure Cosmos DB](#) article.

# Configure Cross-Origin Resource Sharing (CORS)

2/7/2020 • 3 minutes to read • [Edit Online](#)

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.

The screenshot shows the Azure portal interface for managing a Cosmos DB account named 'sqlcdb'. On the left, a sidebar lists various account settings: Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Quick start, Notifications, Data Explorer, Settings, Replicate data globally, Default consistency, Firewall and virtual networks, CORS (which is highlighted with a red box), and Keys. The main content area is titled 'sqlcdb - CORS' and describes Cross-Origin Resource Sharing (CORS). It explains that CORS is an HTTP feature allowing a web application running under one domain to access resources in another domain. It includes sections for 'Allowed Origins' where 'https://foo.com' is listed, and 'Cors rules' which are currently empty. At the top right are 'Submit' and 'Discard' buttons.

## 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](#)

# Secure Azure Cosmos keys using Azure Key Vault

10/21/2019 • 2 minutes to read • [Edit Online](#)

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**.

The screenshot shows the 'Create a secret' dialog in the Azure Key Vault interface. The 'Name' field is populated with 'cdbsecret1'. The 'Value' field contains a masked password. The 'Enabled?' switch is set to 'Yes'. A 'Create' button is at the bottom.

- 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

- Create an Azure web application or you can download the app from the [GitHub repository](#). It is a simple MVC application.
  - 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>")
```
- Save** the file, **Build** the solution.
  - 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**.
  - 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.

The screenshot shows the Azure portal interface for a web application named 'WebAppKeyVault1'. On the left, there's a navigation menu with items like Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Deployment (with sub-options Quickstart, Deployment credentials, Deployment slots, Deployment options, Deployment Center (Preview)), Settings (Application settings, Authentication / Authorization, Application Insights), and Managed service identity (which is highlighted with a red box). The main content area is titled 'Managed service identity' and contains a sub-section 'Register with Azure Active Directory' with a 'On' button (also highlighted with a red box) and 'Save' and 'Discard' buttons below it.

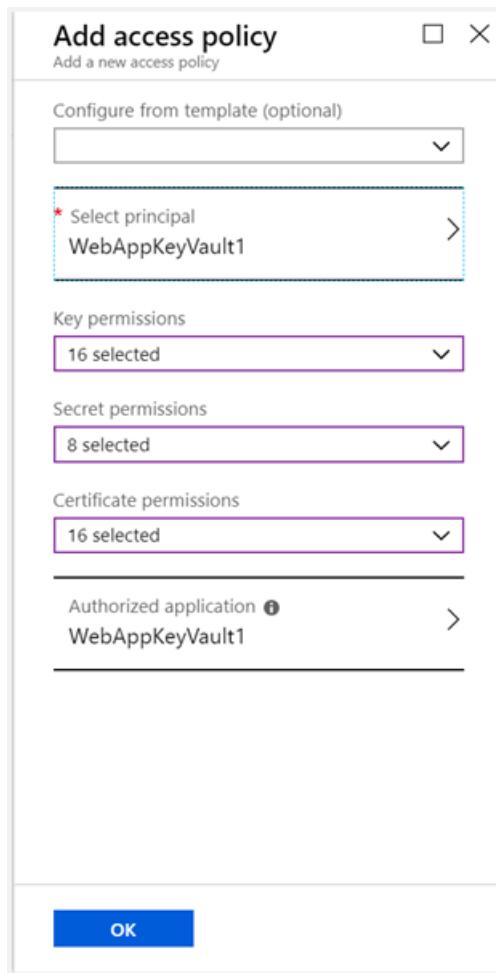
If you will run the application now, you will see the following error, as you have not given any permission to this application in Key Vault.

The screenshot shows a simple web application with a header bar containing 'Application name', 'Home', 'About', and 'Contact'. The main content area displays an error message: 'Principal Used: IsAuthenticated:True Type:App AppId:0bc6ca3c-0491-4fa8-a5d2-d90e1641a895 TenantId:7'. Below this, a yellow box highlights the text 'Something went wrong: Access denied'.

## 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.

The screenshot shows a simple web application interface. At the top is a dark navigation bar with the text 'Application name' and links for 'Home', 'About', and 'Contact'. Below the navigation bar is a white content area. In this content area, there is a message: 'Secret: xxxx' followed by 'Principal Used: IsAuthenticated:True Type:App AppId:0bc6ca3c-0491-4fa8-a5d2-d90e1641a895 TenantId: [redacted]'. The TenantId part is redacted with a grey box.

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.

# Certificate-based authentication for an Azure AD identity to access keys from an Azure Cosmos DB account

2/6/2020 • 6 minutes to read • [Edit Online](#)

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.

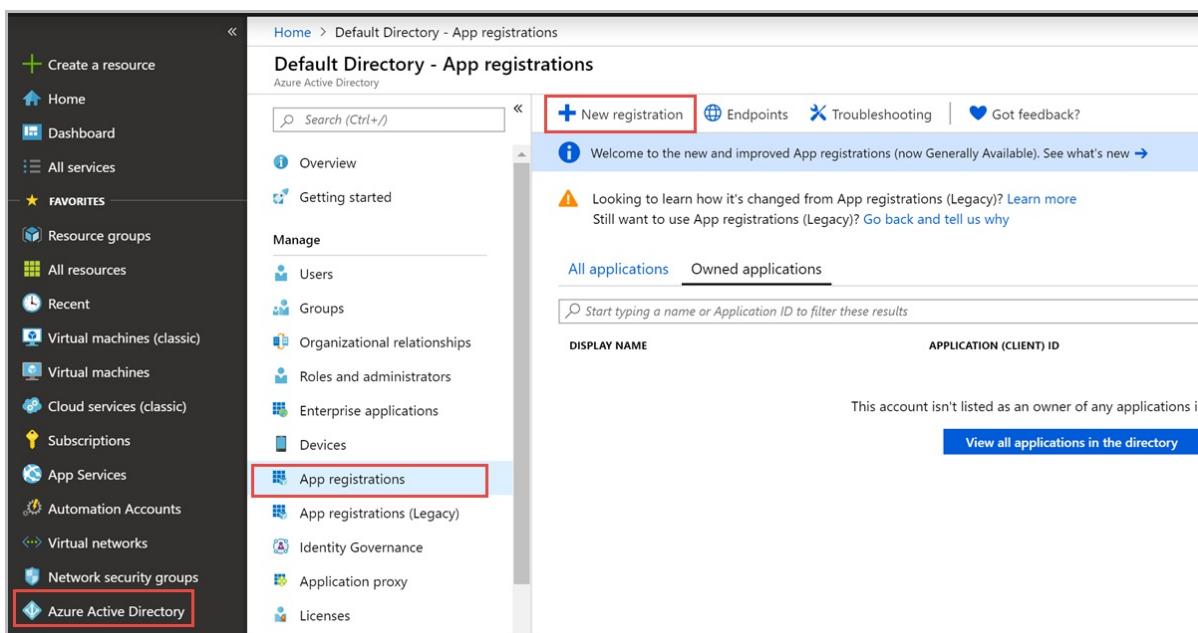
## Prerequisites

- 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**.



The screenshot shows the 'Default Directory - App registrations' page in the Azure Active Directory section. On the left, there's a sidebar with various service links like Home, Dashboard, and Azure Active Directory. Under 'FAVORITES', 'Azure Active Directory' is also listed. The main content area has a search bar, a 'New registration' button (which is highlighted with a red box), and tabs for 'Endpoints', 'Troubleshooting', and 'Got feedback?'. Below these are sections for 'Overview', 'Getting started', and 'Manage' (with sub-options like Users, Groups, etc.). A note about the transition from legacy app registrations is present. The 'All applications' tab is selected, showing a table with columns for DISPLAY NAME and APPLICATION (CLIENT) ID. A message at the bottom states 'This account isn't listed as an owner of any applications in this directory' and a 'View all applications in the directory' button.

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.

The screenshot shows the 'Register an application' page in the Azure portal. The 'Name' field is set to 'sampleApp'. Under 'Supported account types', the radio button for 'Accounts in this organizational directory only (Default Directory)' is selected. The 'Redirect URI (optional)' field is set to 'https://sampleApp.com'. A note at the bottom states: 'By proceeding, you agree to the Microsoft Platform Policies'. A blue 'Register' button is at the bottom.

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.

The screenshot shows the 'sampleApp' registration details. The 'Overview' tab is selected. Key information shown includes:  
Display name : sampleApp  
Application (client) ID : <Your\_application\_ID>  
Directory (tenant) ID : <Your\_tenant\_ID>  
Object ID : <Your\_object\_ID>  
Supported account types : My organization only  
Redirect URIs : 1 web, 0 public client  
Managed application in ... : sampleApp

## 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>

Invoke-AzResourceAction -Action listKeys -ResourceType "Microsoft.DocumentDB/databaseAccounts" -ApiVersion "2015-04-08" -ResourceGroupName <Resource_Group_Name_of_your_Azure_Cosmos_account> -ResourceName <Your_Azure_Cosmos_Account_Name>
```

The previous command will display the primary and secondary master 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.

OPERATION NAME	STATUS	TIME	TIME STAMP	SUBSCRIPTION	EVENT INITIATED BY
▶ List keys	Succeeded	3 min ago	Fri Jun 07 2...	Content Testing	sampleApp
▶ List keys	Succeeded	9 h ago	Fri Jun 07 2...	Content Testing	sampleApp

## 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)
        {
            MainAsync().Wait();
            Console.ReadKey();

            return 0;
        }

        static async Task MainAsync()
        {
            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.DocumentDB/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 master keys as shown in the following screenshot:

```
C:\Program Files\dotnet\dotnet.exe
Got result OK and keys {"primaryMasterKey":"6NIJl41yy06zCTRC10a0wqcR7wXA7HaLc1N1wke6uy5wdRhEu1311m2JR9QRSmKVyqDyiDwyGyrg ^ S9d0vBLk9Q==", "secondaryMasterKey":"T0362VfkXFT0FaaxL14pLUUKEhvGLOJzHCeZQxkxvAhh6kKrdKdnRbn3SawWwsMx07INvH3xEgd19RKDktP0 xA==", "primaryReadOnlyMasterKey":"WfSvnT6cAjM7vbS4oZ4cXcYOGfiHor9qMsXftRxgNFee90DtKPTxJWzcvdhUIJeeOs9kYtsN5RxMGQ4BM9L3WQ ==", "secondaryReadOnlyMasterKey":"q4ozGwu1C2Y1AXG2aIWjMIytNnDmutUIhrUqqatVihx1F0QoJpHamLYpatJDCywfQ8mFoc1d6NQe2pWH7rj220 =="} ^
```

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 controls for Azure Cosmos DB](#)

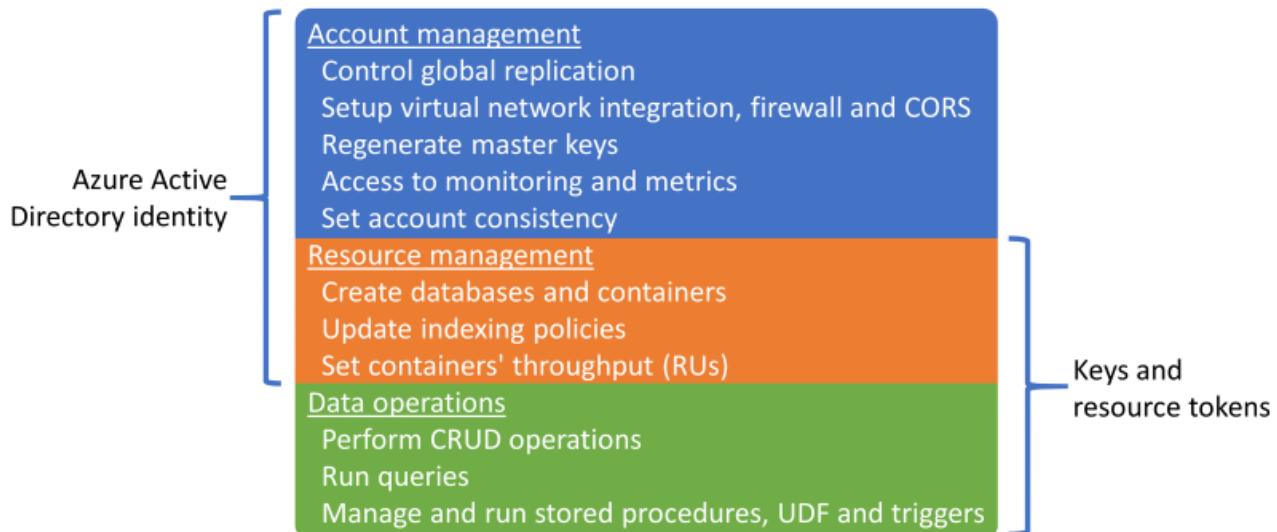
# Restrict user access to data operations only

12/9/2019 • 2 minutes to read • [Edit Online](#)

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: `mycosmosdbaccount`.

- `$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 [Custom roles for Azure resources](#) 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": "00000000-0000-0000-0000-000000000000",
  "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

2/20/2020 • 5 minutes to read • [Edit Online](#)

## NOTE

At this time, you must request access to use this capability. To do so, please contact [azurecosmosdbcmk@service.microsoft.com](mailto:azurecosmosdbcmk@service.microsoft.com).

Data stored in your Azure Cosmos account is automatically and seamlessly encrypted. Azure Cosmos DB offers two options to manage the keys used to encrypt the data at rest:

- **Service-managed keys:** By default, Microsoft manages the keys that are used to encrypt the data in your Azure Cosmos account.
- **Customer-managed keys (CMK):** You can optionally choose to add a second layer of encryption with your own keys.

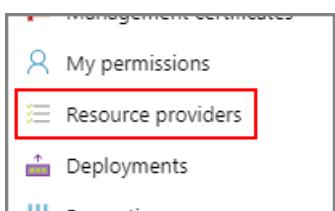
You must store customer-managed keys in [Azure Key Vault](#) and provide a key for each Azure Cosmos account that is enabled with customer-managed keys. This key is used to encrypt all the data stored in that account.

## NOTE

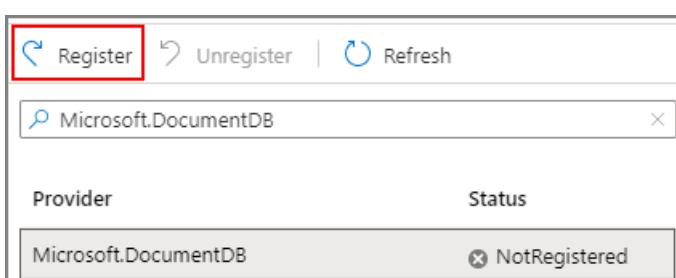
Currently, customer-managed keys are available only for new Azure Cosmos accounts. You should configure them during account creation.

## 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

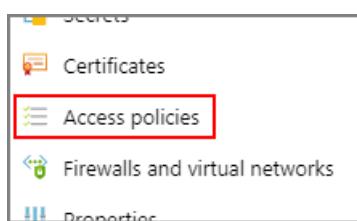
Using customer-managed keys with Azure Cosmos DB requires you to set two properties on the Azure Key Vault instance that you plan to use to host your encryption keys. These properties include **Soft Delete** and **Do Not Purge**. These properties aren't enabled by default. You can enable them by using either PowerShell or the Azure CLI.

To learn how to enable these properties on an existing Azure Key Vault instance, 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:



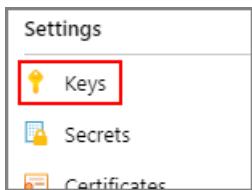
4. Under **Select principal**, select **None selected**. Then, search for **Azure Cosmos DB** principal and select it (to make it easier to find, you can also search by principal ID: `a232010e-820c-4083-83bb-3ace5fc29d0b` for any Azure region except Azure Government regions where the principal ID is `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.

The screenshot shows the 'Add access policy' dialog in the Azure portal. On the left, there are fields for 'Configure from template (optional)', 'Key permissions' (3 selected), 'Secret permissions' (0 selected), 'Certificate permissions' (0 selected), and 'Select principal' (None selected). Below these are sections for 'Authorized application' (None selected) and an 'Add' button. On the right, a modal titled 'Principal' is open, showing a list of principals with 'Azure Cosmos DB' selected. A red box highlights the 'Select' button at the bottom right of the modal.

5. Select **Add** to add the new access policy.

## Generate a key in Azure Key Vault

1. From the Azure portal, go to 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**:

The screenshot shows the 'Create Key' dialog. It includes fields for 'Name' (my-cosmos-db-key), 'Key Type' (RSA selected), 'RSA Key Size' (3072 selected), and 'Enabled?' (Yes selected). A red box highlights the 'Create' button at the bottom left.

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:

A screenshot of a web browser showing the URL of an Azure Key Vault key. The URL is highlighted with a red box: `https://<my-vault>.vault.azure.net/keys/my-cosmos-db-key/cdf3f4cb1f284cc3a0d00b53dc3e4617`. To the right of the URL is a blue copy icon.

## 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:

A screenshot of the Azure portal's 'Encryption' tab for account creation. The tab is highlighted with a red box. Below it, under 'Data Encryption', there are two options: 'Service-managed key' (radio button) and 'Custom-managed key (Enter key URI)' (radio button, which is selected and highlighted with a red box). In the 'Key URI' section, the copied URL from the previous step is pasted into the input field, which is also highlighted with a red box.

### 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** as the API version.

#### IMPORTANT

You must set the `Location` parameter 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
```

### 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** as the API version.

#### IMPORTANT

You must set the **Location** parameter 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
```

# Frequently asked questions

## Is there any additional charge for using customer-managed keys?

Yes. To account for the additional compute load that is required to manage data encryption and decryption with customer-managed keys, all operations executed against the Azure Cosmos account consume a 25 percent increase in [Request Units](#).

## 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 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:

The screenshot shows the 'Properties' section of an Azure Key Vault key named 'cdf3f4cb1f284cc3a0d00b53dc3e4617'. It includes fields for Key Type (RSA), RSA Key Size (2048), and creation/updated dates. Below this is the 'Key Identifier' URL. The 'Settings' section contains options for activation and expiration dates, with 'Enabled?' set to 'No' (highlighted by a red box). The 'Tags' section shows 0 tags. Under 'Permitted operations', all six checkboxes are selected: Encrypt, Decrypt, Sign, Verify, Wrap Key, and Unwrap 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:

Current Access Policies						
Name	Category	Email	Key Permissions	Secret Permissions	Certificate Permissions	Action
APPLICATION						
<input checked="" type="checkbox"/> Azure Cosmos DB	APPLICATION		<input checked="" type="checkbox"/> 3 selected	<input type="checkbox"/> 0 selected	<input type="checkbox"/> 0 selected	<input type="button" value="Delete"/>

#### 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](#).

# Restore data from a backup in Azure Cosmos DB

9/3/2019 • 3 minutes to read • [Edit Online](#)

If you accidentally delete your database or a container, 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. 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.

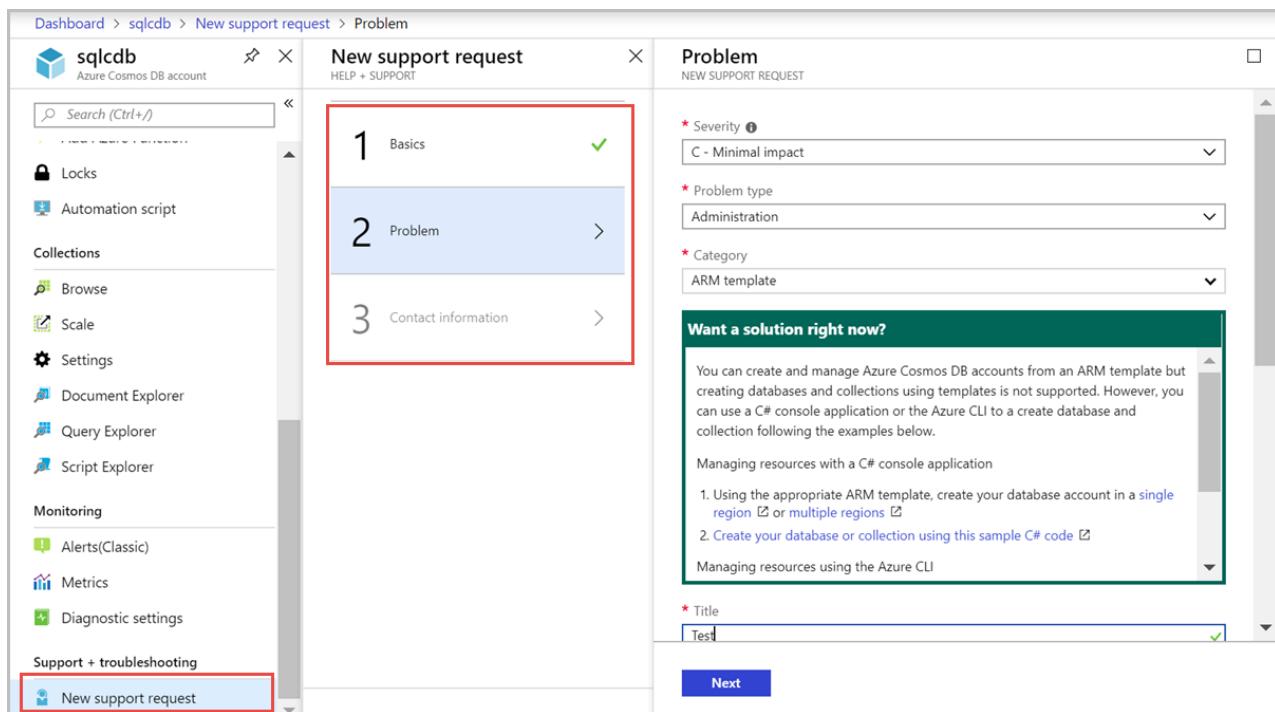
## Request a restore

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 of the state of the 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.
  - If you have accidentally deleted your database or container, open a Sev B or Sev C Azure support case.
  - If you have accidentally deleted or corrupted some documents within the container, open a Sev A support case.

When data corruption occurs and if the documents within a container are modified or deleted, **delete the container as soon as possible**. By deleting the container, you can avoid Azure Cosmos DB from overwriting the backups. If for some reason the deletion is not possible, you should file a ticket as soon as possible. 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.



## Post-restore actions

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 coadmin can see the restored account.

After the data is restored, you should inspect and validate the data in the restored account and make sure it contains the version that you are expecting. If everything looks good, you should migrate the data back to your original account using [Azure Cosmos DB change feed](#) or [Azure Data Factory](#).

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 migrate the data back to your original account using the following articles:

- 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.

# Monitoring Azure Cosmos DB

12/13/2019 • 5 minutes to read • [Edit Online](#)

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.

## 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.

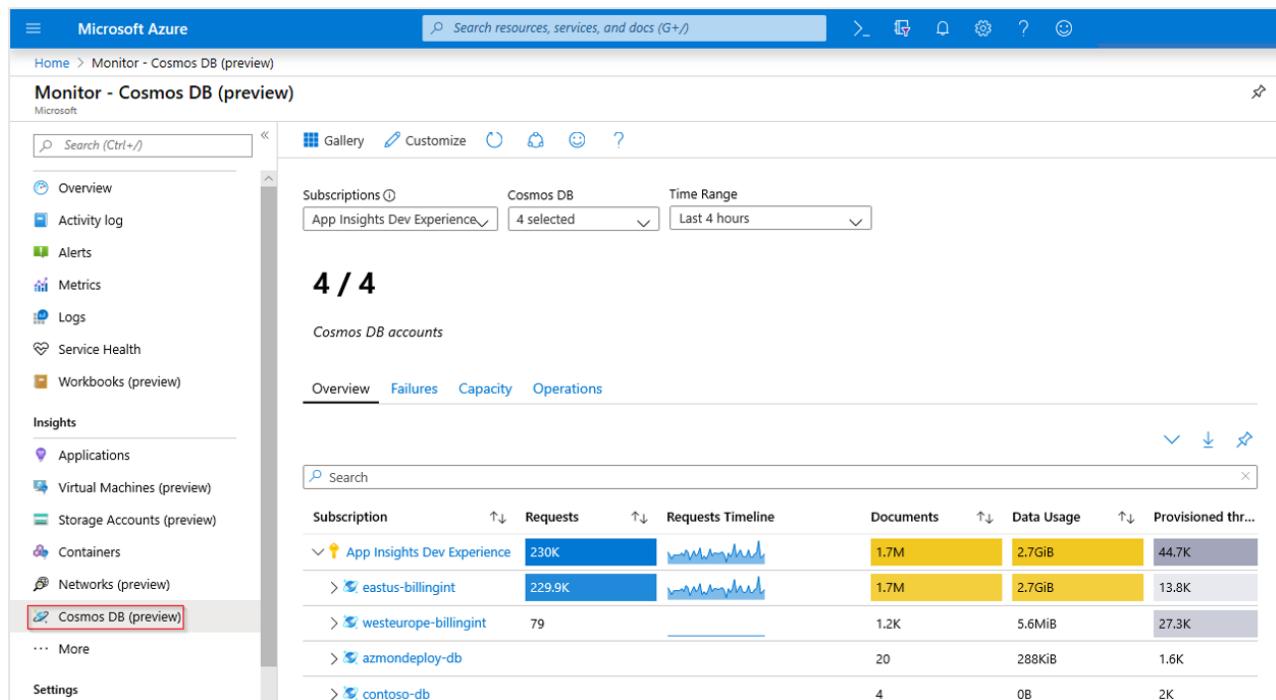
If you're not already familiar with monitoring Azure services, start with the article [Monitoring Azure resources with Azure Monitor](#) which describes the following:

- 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 Cosmos DB (Preview)

[Azure Monitor for Azure Cosmos DB](#) is based on the [workbooks feature of Azure Monitor](#) and uses the same monitoring data collected for Cosmos DB described in the sections below. Use this tool 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.

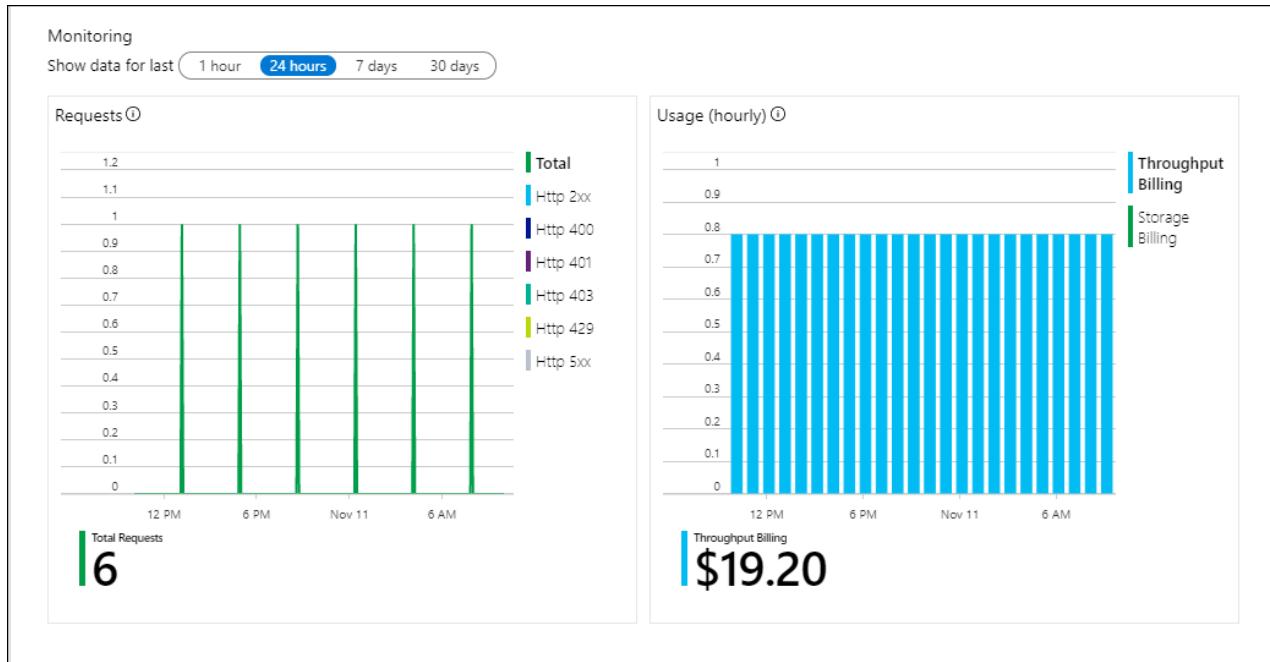


Subscription	Requests	Requests Timeline	Documents	Data Usage	Provisioned thr...
App Insights Dev Experience	230K		1.7M	2.7GiB	44.7K
eastus-billingint	229.9K		1.7M	2.7GiB	13.8K
westeurope-billingint	79		1.2K	5.6MiB	27.3K
azmondeploy-db			20	288KiB	1.6K
contoso-db			4	0B	2K

# Monitoring data collected from Azure Cosmos DB

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.



## Analyzing metric data

Azure Cosmos DB provides a custom experience for working with metrics. See [Monitor and debug Azure Cosmos DB metrics from Azure Monitor](#) for details on using this experience and for analyzing different Azure Cosmos DB scenarios.

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. 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

## Analyzing log data

Data in Azure Monitor Logs is stored in tables which each table having its own set of unique properties. Azure Cosmos DB stores data in the following tables.

TABLE	DESCRIPTION
-------	-------------

TABLE	DESCRIPTION
AzureDiagnostics	Common table used by multiple services to store Resource logs. Resource logs from Azure Cosmos DB can be identified with <code>MICROSOFT.DOCUMENTDB</code> .
AzureActivity	Common table that stores all records from the Activity log.

### 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 database. This means that log queries will only include data from that resource. If you want to run a query that includes data from other databases 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.

## Azure Cosmos DB Log Analytics queries in Azure Monitor

Here are some queries that you can enter into the **Log search** search bar to help you monitor your Azure Cosmos containers. These queries work with the [new language](#).

Following are queries that you can use to help you monitor your Azure Cosmos databases.

- 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 the 10 most recently logged events:

```
AzureDiagnostics
| where ResourceProvider=="MICROSOFT.DOCUMENTDB" and Category=="DataPlaneRequests"
| limit 10
```

- To query for all operations, grouped by operation type:

```
AzureDiagnostics
| where ResourceProvider=="MICROSOFT.DOCUMENTDB" and Category=="DataPlaneRequests"
| summarize count() by OperationName
```

- 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
```

- To get all queries greater than 100 RUs 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
```

- To query for which operations take longer than 3 milliseconds:

```
AzureDiagnostics
| where toint(duration_s) > 3 and ResourceProvider=="MICROSOFT.DOCUMENTDB" and
Category=="DataPlaneRequests"
| summarize count() by clientIpAddress_s, TimeGenerated
```

- To query for which agent is running the operations:

```
AzureDiagnostics
| where ResourceProvider=="MICROSOFT.DOCUMENTDB" and Category=="DataPlaneRequests"
| summarize count() by OperationName, userAgent_s
```

- To query for when the long running operations were performed:

```
AzureDiagnostics
| where ResourceProvider=="MICROSOFT.DOCUMENTDB" and Category=="DataPlaneRequests"
| project TimeGenerated , duration_s
| summarize count() by bin(TimeGenerated, 5s)
| render timechart
```

- To get Partition Key statistics to evaluate skew across top 3 partitions for database account:

```
AzureDiagnostics
| where ResourceProvider=="MICROSOFT.DOCUMENTDB" and Category=="PartitionKeyStatistics"
| project SubscriptionId, regionName_s, databaseName_s, collectionname_s, partitionkey_s, sizeKb_s,
ResourceId
```

## 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/Microsoft.DocumentDb/databaseAccounts/{DocumentDBAccountName}/metricDefinitions?api-version=2015-04-08
```

Queries to retrieve individual metrics use the following format:

```
https://management.azure.com/subscriptions/{SubscriptionId}/resourceGroups/{ResourceGroup}/providers/Microsoft.DocumentDb/databaseAccounts/{DocumentDBAccountName}/metrics?api-version=2015-04-08&$filter=%28name.value%20eq%20%27Total%20Requests%27%29%20and%20timeGrain%20eq%20duration%27PT5M%27%20and%20startTime%20eq%202016-06-03T03%3A26%3A00.000000Z%20and%20endTime%20eq%202016-06-10T03%3A26%3A00.000000Z
```

## 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.

# Monitor Azure Cosmos DB data by using diagnostic settings in Azure

12/13/2019 • 5 minutes to read • [Edit Online](#)

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` , and `partitionID` .

```
{ "time": "2019-04-23T23:12:52.3814846Z", "resourceId": "/SUBSCRIPTIONS/<your_subscription_ID>/RESOURCEGROUPS/<your_resource_group>/PROVIDERS/MICROSOFT.DOCUMENTDB/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", "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. MongoDB requests will appear in MongoRequests as well as DataPlaneRequests. Key properties to note are:

`Requestcharge` , `opCode` .

```
{ "time": "2019-04-10T15:10:46.7820998Z", "resourceId": "/SUBSCRIPTIONS/<your_subscription_ID>/RESOURCEGROUPS/<your_resource_group>/PROVIDERS/MICROSOFT.DOCUMENTDB/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"}}
```

- **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.DOCUMENTATIONDB/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.DOCUMENTATIONDB/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 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

```

2. 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)

```

3. 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)

```

4. What is the maximum throughput that a partition provides?

```

AzureDiagnostics
| where ResourceProvider=="MICROSOFT.DOCUMENTDB" and Category=="DataPlaneRequests"
| where TimeGenerated >= ago(2h)
| summarize max(requestCharge_s) by bin(TimeGenerated, 1h), partitionId_g

```

5. 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

```

6. 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"

```

7. 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

```

8. 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
```

9. How to get partition Key statistics to evaluate skew across top three partitions for database account?

```
AzureDiagnostics  
| where ResourceProvider=="MICROSOFT.DOCUMENTDB" and Category=="PartitionKeyStatistics"  
| project SubscriptionId, regionName_s, databaseName_s, collectionname_s, partitionkey_s, sizeKb_s,  
ResourceId
```

## Next steps

- [Azure Monitor for Azure Cosmos DB](#)
- [Monitor and debug with metrics in Azure Cosmos DB](#)

# Explore Azure Monitor for Azure Cosmos DB (preview)

2/27/2020 • 6 minutes to read • [Edit Online](#)

Azure Monitor for Azure Cosmos DB (preview) 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 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**.

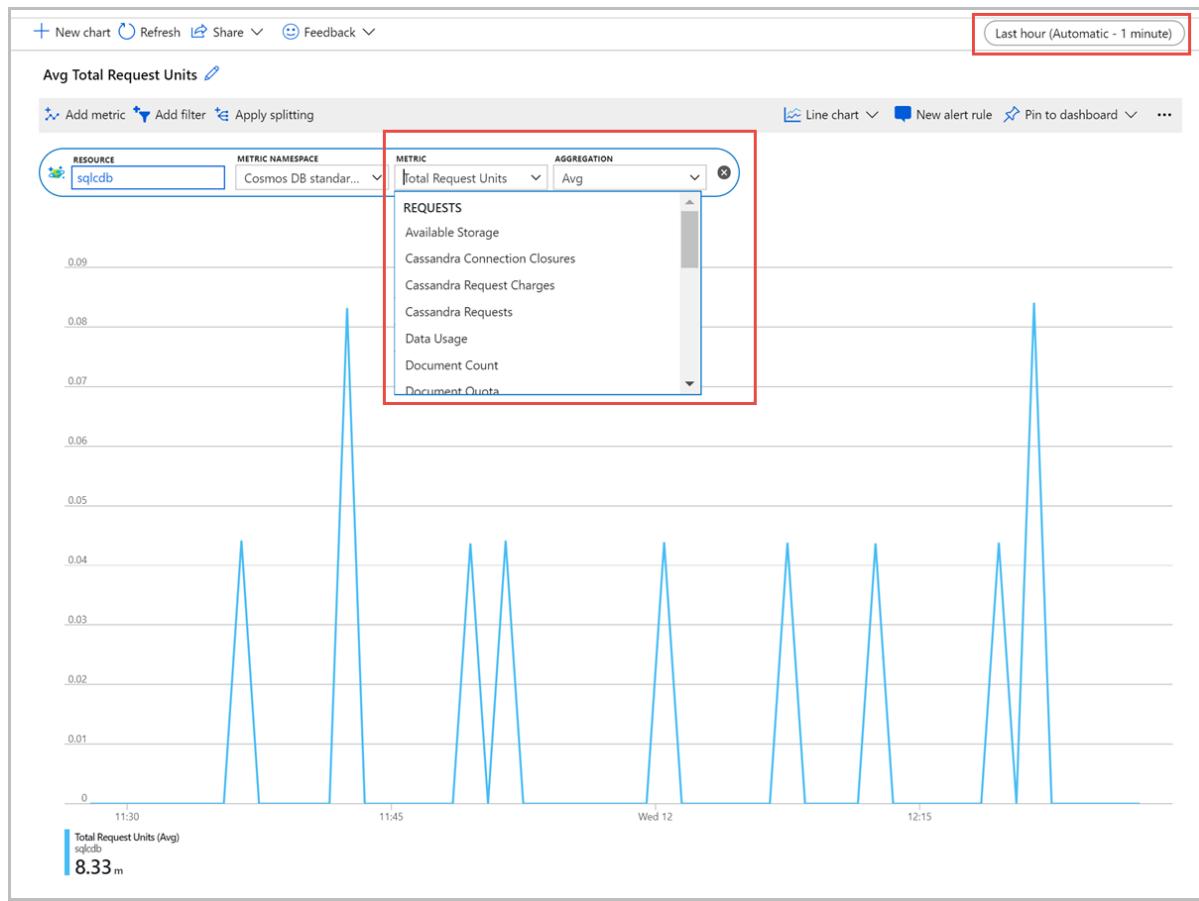
The screenshot shows the Microsoft Azure portal interface. On the left, there's a sidebar with various navigation options like 'Create a resource', 'Home', 'Dashboard', 'All services', and 'FAVORITES'. Under 'FAVORITES', 'Monitor' is selected and highlighted with a red box. The main content area is titled 'Monitor - Metrics' and contains sections for 'Overview', 'Activity log', 'Alerts', 'Metrics' (which is also highlighted with a red box), 'Logs', 'Service Health', 'Workbooks (preview)', 'Insights' (with sub-options like 'Applications', 'Virtual Machines (preview)', 'Containers', 'Network', and 'More'), 'Settings' (with 'Diagnostics settings' and 'Autoscale'), and 'Metrics' again at the bottom.

- 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**.

The screenshot shows the 'Select a resource' dialog box. It has tabs for 'Browse' and 'Recent'. Under 'Subscriptions', it says '1 of 11 selected'. The 'Resource type' dropdown is set to 'Azure Cosmos DB accounts'. The list of resources shows several entries: 'cdbgraph1', 'cdbmm', 'cdbmongo', 'cdbsm1', 'cdbtable', and 'sqlcdb'. A search bar at the bottom is highlighted with a red box.

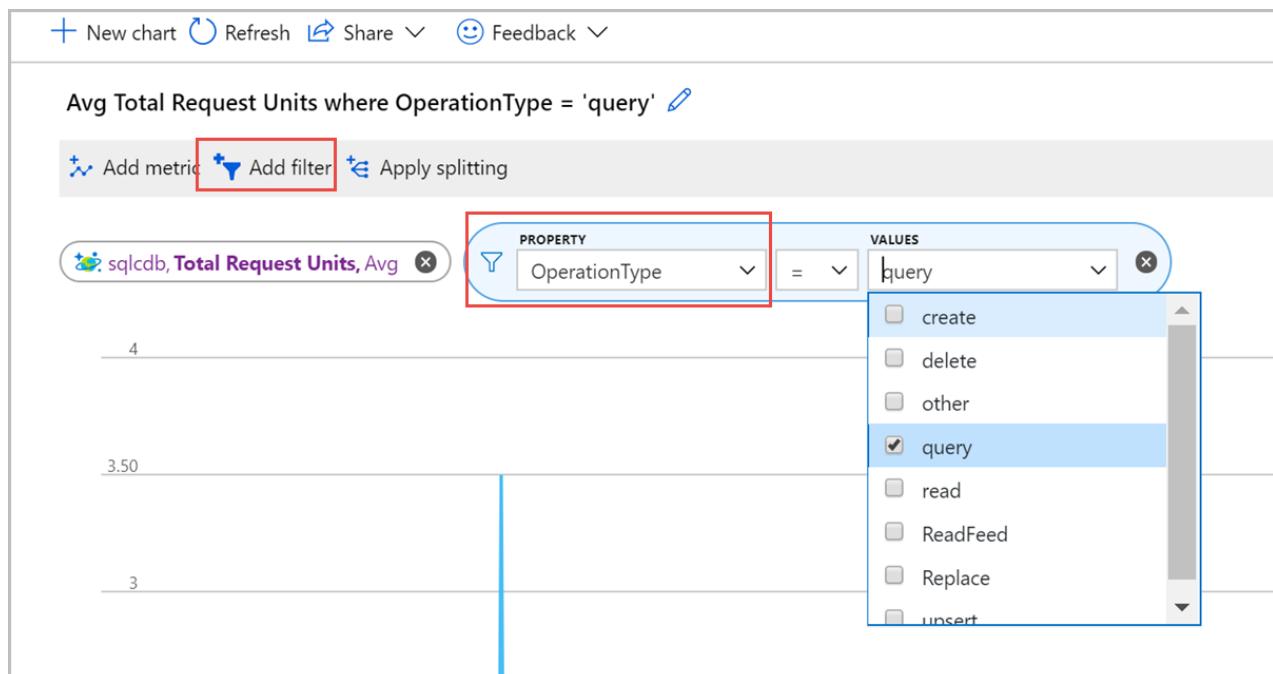
- Next you can select a metric from the list of available metrics. You can select metrics specific to request units, storage, latency, availability, Cassandra, and others. To learn in detail about all the available metrics in this list, see the [Metrics by category](#) article. In this example, let's 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.

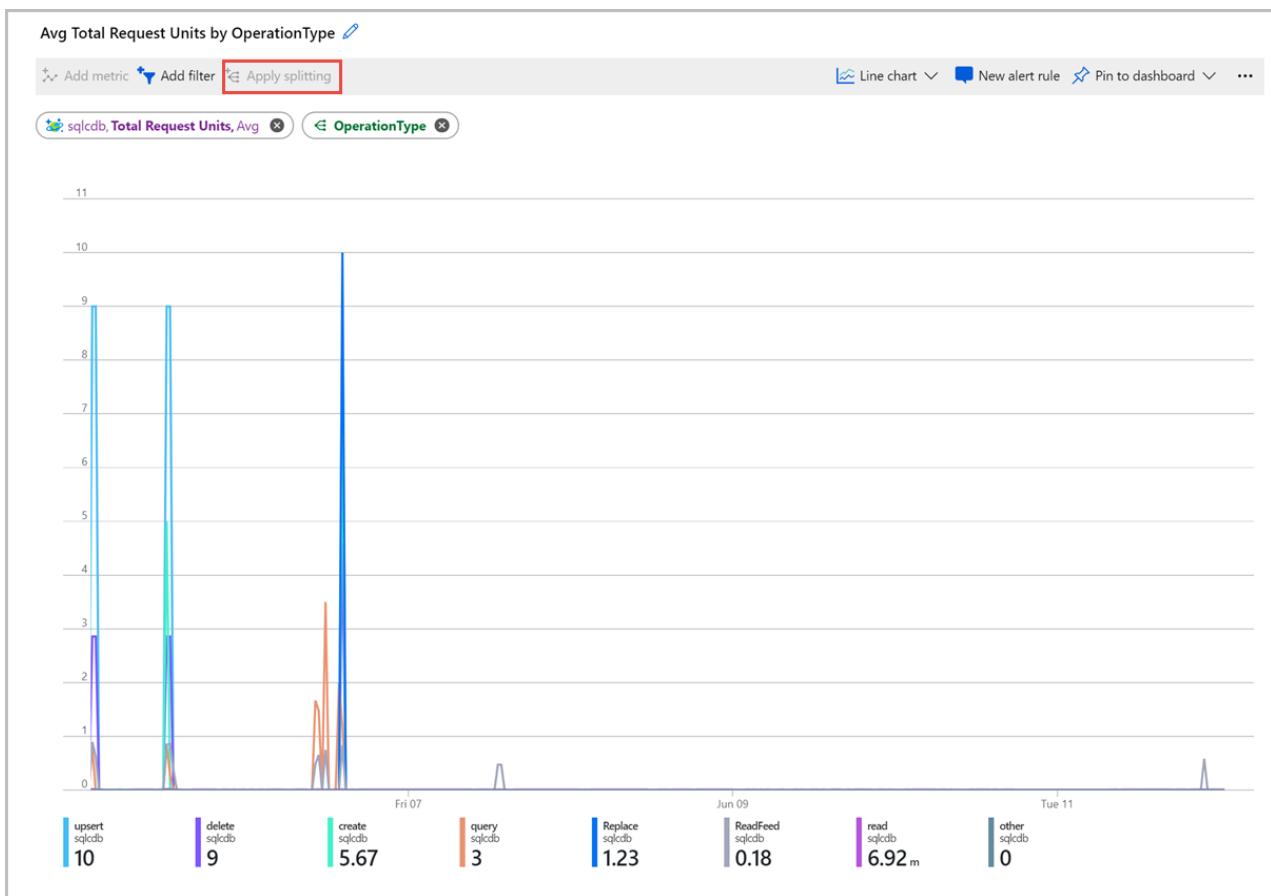


## Add filters to metrics

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.



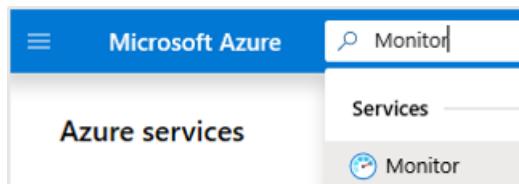
You can group metrics by using the **Apply splitting** option. For example, you can group the request units per operation type and view the graph for all the operations at once as shown in the following image:



## 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 (preview)**.

The screenshot shows the Microsoft Azure portal with the title "Microsoft Azure" at the top. Below it, the breadcrumb navigation shows "Home > Monitor - Cosmos DB (preview)". The main title is "Monitor - Cosmos DB (preview)" with a "Microsoft" logo. On the left, there's a sidebar with links like Overview, Activity log, Alerts, Metrics, Logs, Service Health, Workbooks (preview), Insights (with Applications, Virtual Machines, Storage Accounts, Containers, Networks, and Cosmos DB (preview) selected), and More. The main content area has a search bar and a "4 / 4" status indicator. It displays "Cosmos DB accounts" with tabs for Overview, Failures, Capacity, and Operations. The Overview tab is active, showing a table with columns: Subscription, Requests, Requests Timeline, Documents, Data Usage, and Provisioned thr... . The table lists several subscriptions: App Insights Dev Experience (230K requests, 1.7M documents, 2.7GiB data usage, 44.7K provisioned throughput), eastus-billingint (229.9K requests, 1.7M documents, 2.7GiB data usage, 13.8K provisioned throughput), westeurope-billingint (79 requests, 1.2K documents, 5.6MiB data usage, 27.3K provisioned throughput), azmondeploy-db (20 requests, 20 documents, 288KiB data usage, 1.6K provisioned throughput), and contoso-db (4 requests, 4 documents, 0B data usage, 2K provisioned throughput). Each row includes a "Search" button and a copy icon.

## 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.

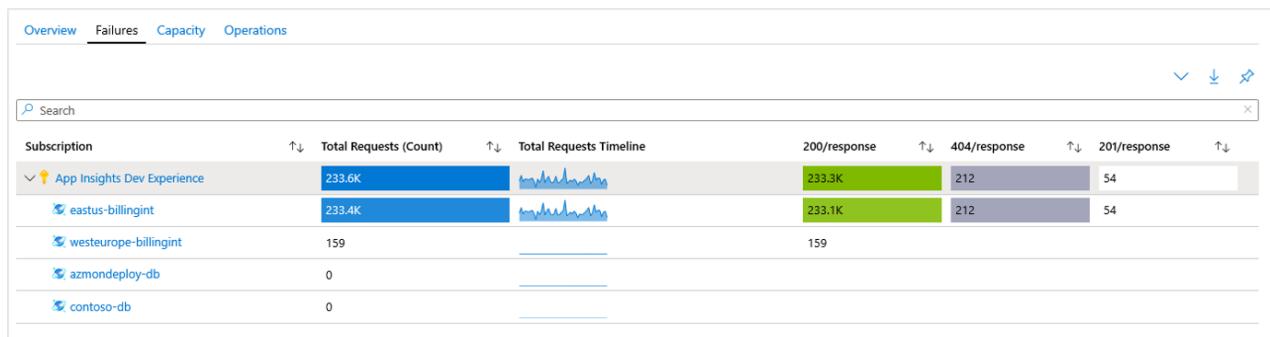
Selecting a drop-down arrow next to one of the Azure Cosmos DB resources will reveal a breakdown of the performance metrics at the individual database container level:

The screenshot shows the "Failures" tab of the Azure Cosmos DB monitor. It has a search bar and tabs for Overview, Failures, Capacity, and Operations. The table columns are: Subscription, Requests, Requests Timeline, Documents, Data Usage, and Provisioned thr... . The table lists the same subscriptions as the Overview tab, but with more detailed data for each. For example, the "eastus-billingint" subscription is expanded to show its children: "current2" (231.3K requests, 497.9K documents, 781.5MiB data usage, 5K provisioned throughput), "history2" (25 requests, 981.8K documents, 1.6GiB data usage, 2.1K provisioned throughput), and "History" (96.2K documents, 164MiB data usage, 1.7K provisioned throughput). The "Current" row shows 65.9K documents and 100.1MiB data usage. Other rows include "westerneurope-billingint" (1.2K requests, 5.5MiB data usage, 27.3K provisioned throughput), "azmondeploy-db" (20 requests, 288KiB data usage, 1.6K provisioned throughput), and "contoso-db" (4 requests, 0B data usage, 2K provisioned throughput).

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:

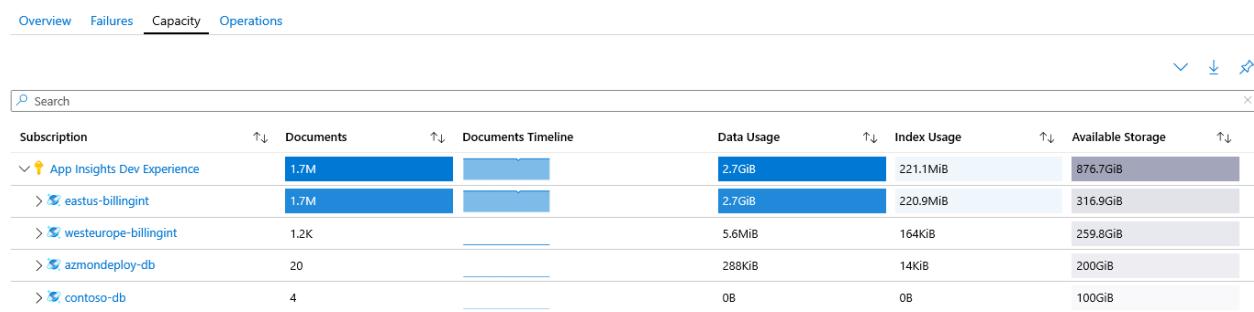


CODE	DESCRIPTION
200 OK	One of the following REST operations were successful: - GET on a resource. - PUT on a resource. - POST on a resource. - POST on a stored procedure resource to execute the stored procedure.
201 Created	A POST operation to create a resource is successful.
404 Not Found	The operation 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.

Subscription	↑↓ Requests	↑↓ Requests Timeline	read/operation ↑↓	upsert/operation ↑↓	create/operation ↑↓	Other/operation ↑↓
App Insights Dev Experience	251.5K		251.3K	105	105	0
eastus-billingint	251.2K		251K	105	105	0
westeurope-billingint	335		335			0
azmondeploy-db	0					

## 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.

Subscription	↑↓ Requests	↑↓ Requests Timeline	Documents	↑↓ Data Usage	↑↓ Provisioned throughput
App Insights Dev Experience	235.9K		1.7M	2.7GiB	44.7K
eastus-billingint	235.8K		1.7M	2.7GiB	13.8K
westeurope-billingint	148		1.2K	5.5MiB	27.3K
azmondeploy-db			20	288KiB	1.6K

To export your data into the Excel format, select the down arrow icon to the left of the pushpin icon.

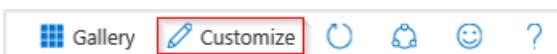


To expand or collapse all drop-down views in the workbook, select the expand icon to the left of the export icon:

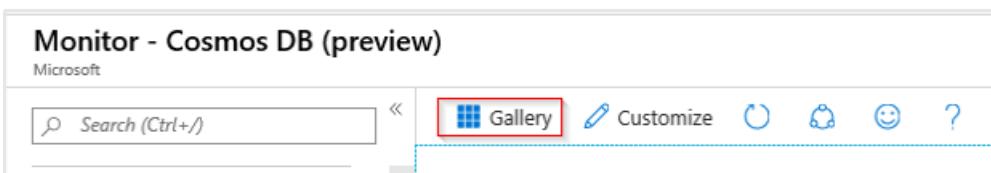


## Customize Azure Monitor for Azure Cosmos DB (preview)

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.



## 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

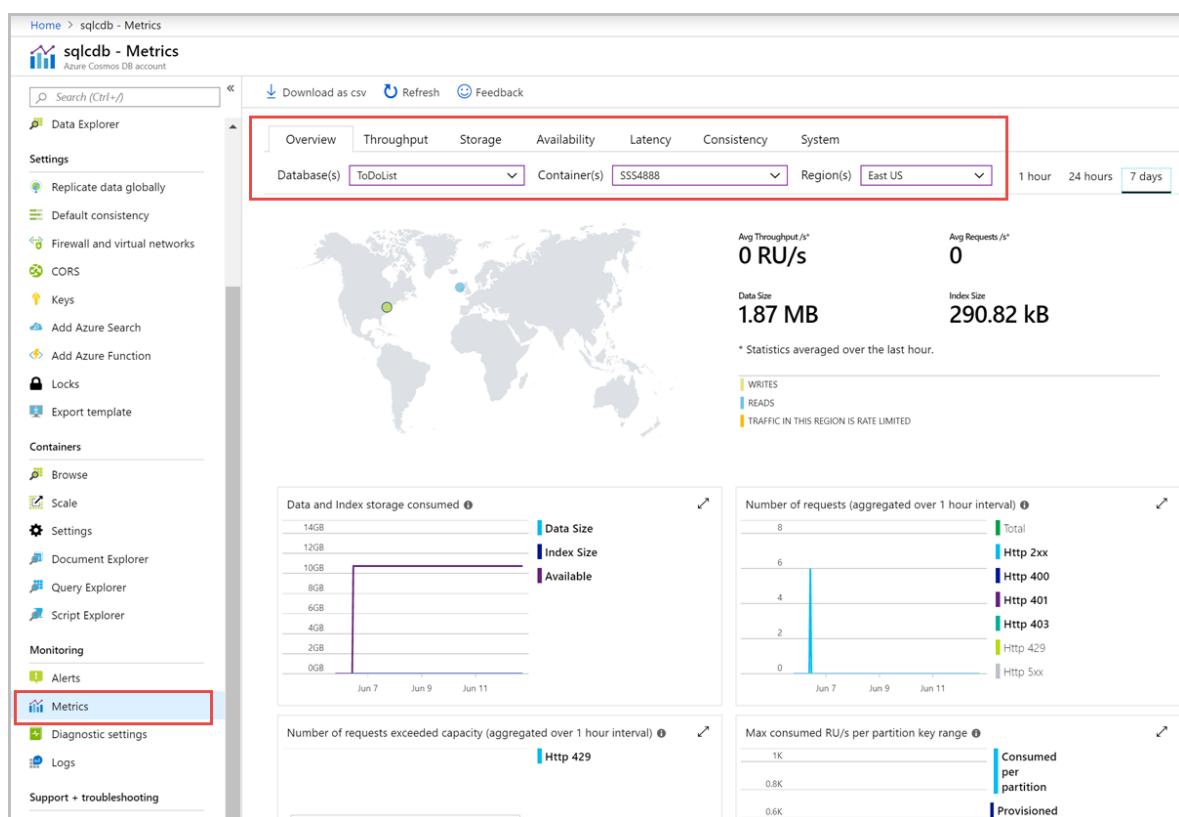
6/19/2019 • 4 minutes to read • [Edit Online](#)

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. 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.

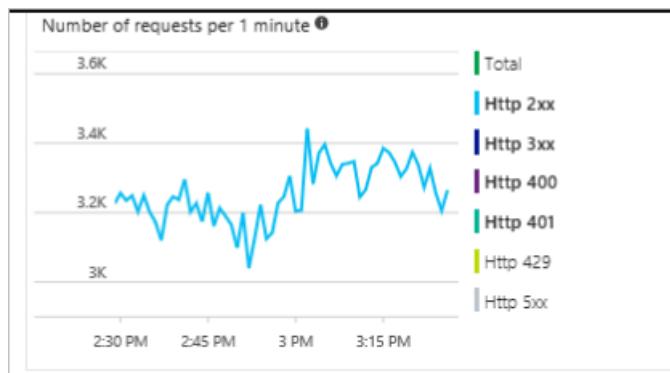
- **Latency metrics** - This metric shows the read and write latency observed by Azure Cosmos DB in the region where your account is operating. You can visualize latency across regions for a geo-replicated account. This metric doesn't represent the end-to-end request latency.
- **Consistency metrics** - 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 master 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

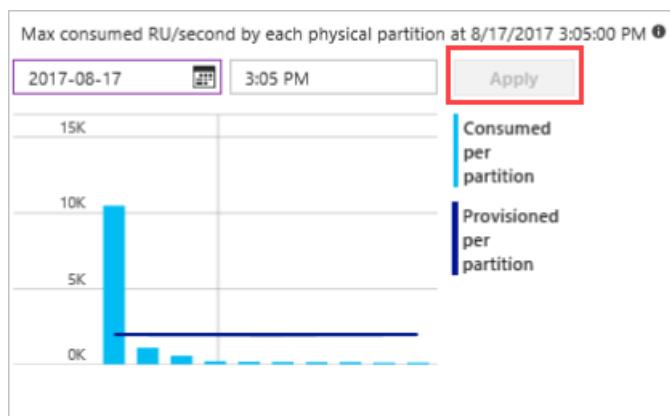
To get started, head to the [Azure portal](#) and navigate to the **Metrics** blade. In the blade, find the \*\*Number of requests exceeded capacity per 1-minute chart. This chart shows a minute by minute total requests segmented by the status code. For more information about HTTP status codes, see [HTTP 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.

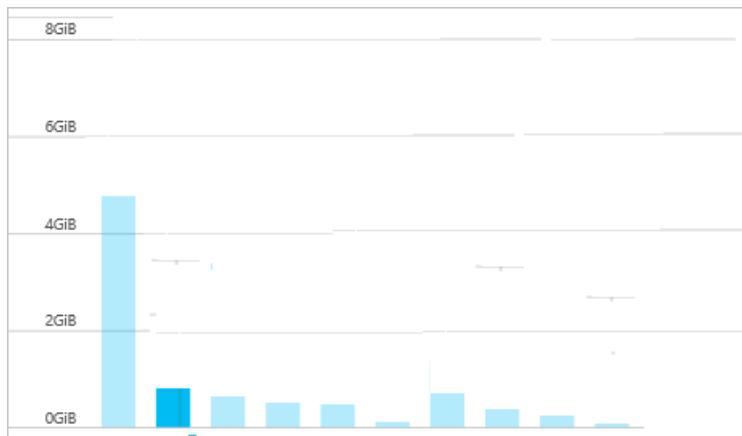


An uneven throughput distribution may cause *hot* partitions, which can result in throttled requests and may

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.



```
// 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](#)

# Scenario: Exception handling and error logging for logic apps

1/30/2020 • 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 PatientId**, **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

InsertLogEntry

PRESCRIBERID  
CRMID \*

OPERATION  
New\_Patient

You can insert data from previous steps...

Outputs from manual

Body CRMID recordType salesforceID  
update

SOURCE  
Headers \*

SALESFORCEID  
salesforceID \*

DATE  
Date \*

...

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('CreateErrorRecord')['status'], 'Failed')
```



# 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": "XngRAOT6IQEAAAAAAAAAAA==",
    "_self": " dbs/XngRAA==/colls/XngRAOT6IQE=/docs/XngRAOT6IQEAAAAAAAAAAA==/",
    "_ts": 1465597936,
    "_etag": "/0400fc2f-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: 001U0000001c83gK",
        "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_vod__c/": "/"/, /"NPI_vod__c/": "/"/, /"PDRP_MED__c/": false, /"PersonDoNotCall/": false, /"PersonEmail/": "/"/, /"PersonHasOptedOutOfEmail/": false, /"PersonHasOptedOutOfFax/": 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/", /"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": "sQx2APhVzAA8AAAAAAA==",
    "_self": "dbs/sQx2AA=/colls/sQx2APhVzAA=/docs/sQx2APhVzAA8AAAAAAA==/",
    "_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:/:"", /LastName:/:"BAILEY"/, /MterID_mp__c:/:"", /Medicis_ID_MED__c:/:"851588/", /Middle_vod__c:/:"", /NPI_vod__c:/:"", /PDRP_MED__c:/:false, /PersonDoNotCall:/:false, /PersonEmail:/:"", /PersonHasOptedOutOfEmail:/:false, /PersonHasOptedOutOfFax:/: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



#### Error management detail view

## 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":"001m000000X74NAAZ","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	<input checked="" type="checkbox"/>
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

Properties	
PeterJamesChalmers_1466191912	
_RID	Uw4fAJrEEwEqAAAAAAAAAA==
_TS	Fri, 17 Jun 2016 19:31:50 GMT
_SELF	dbs/Uw4fAA==/colls/Uw4fAJrEEwE=/docs/Uw4fAJrEEwE=/_self
_ETAG	"0000dc00-0000-0000-0000-576450290C
_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](#)

# Azure Cosmos DB monitoring data reference

2/24/2020 • 5 minutes to read • [Edit Online](#)

This article provides a reference of log and metric data collected to analyze the performance and availability of Azure Cosmos DB. See [Monitoring Cosmos DB](#) for details on collecting and analyzing monitoring data for Azure Cosmos DB.

## Resource logs

The following table lists the properties for Azure Cosmos DB resource logs when they're collected in Azure Monitor Logs or Azure Storage. In Azure Monitor Logs, they're collected in the **AzureDiagnostics** table with a **ResourceProvider** value of *MICROSOFT.DOCUMENTDB*.

AZURE STORAGE FIELD OR PROPERTY	AZURE MONITOR LOGS PROPERTY	DESCRIPTION
<b>time</b>	<b>TimeGenerated</b>	The date and time (UTC) when the operation occurred.
<b>resourceId</b>	<b>Resource</b>	The Azure Cosmos DB account for which logs are enabled.
<b>category</b>	<b>Category</b>	For Azure Cosmos DB logs, <b>DataPlaneRequests</b> , <b>MongoRequests</b> , <b>QueryRuntimeStatistics</b> , <b>PartitionKeyStatistics</b> , <b>PartitionKeyRUConsumption</b> , <b>ControlPlaneRequests</b> are the available log types.
<b>operationName</b>	<b>OperationName</b>	Name of the operation. This value can be any of the following operations: Create, Update, Read, ReadFeed, Delete, Replace, Execute, SqlQuery, Query, JQuery, Head, HeadFeed, or Upsert.
<b>properties</b>	n/a	The contents of this field are described in the rows that follow.
<b>activityId</b>	<b>activityId_g</b>	The unique GUID for the logged operation.
<b>userAgent</b>	<b>userAgent_s</b>	A string that specifies the client user agent that's performing the request. The format is {user agent name}/{version}.
<b>request ResourceType</b>	<b>request ResourceType_s</b>	The type of the resource accessed. This value can be any of the following resource types: Database, Container, Document, Attachment, User, Permission, StoredProcedure, Trigger, UserDefinedFunction, or Offer.

AZURE STORAGE FIELD OR PROPERTY	AZURE MONITOR LOGS PROPERTY	DESCRIPTION
<b>statusCode</b>	<b>statusCode_s</b>	The response status of the operation.
<b>requestResourceId</b>	<b>ResourceId</b>	The resourceId that pertains to the request. The value may point to databaseRid, collectionRid, or documentRid depending on the operation performed.
<b>clientIpAddress</b>	<b>clientIpAddress_s</b>	The client's IP address.
<b>requestCharge</b>	<b>requestCharge_s</b>	The number of RUs that are used by the operation
<b>collectionRid</b>	<b>collectionId_s</b>	The unique ID for the collection.
<b>duration</b>	<b>duration_s</b>	The duration of the operation, in milliseconds.
<b>requestLength</b>	<b>requestLength_s</b>	The length of the request, in bytes.
<b>responseLength</b>	<b>responseLength_s</b>	The length of the response, in bytes.
<b>resourceTokenUserId</b>	<b>resourceTokenUserId_s</b>	This value is non-empty when <a href="#">resource tokens</a> are used for authentication. The value points to the resource ID of the user.

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 CosmosDB), see [Azure Monitor supported metrics](#).

### Request metrics

METRIC (METRIC DISPLAY NAME)	UNIT (AGGREGATION TYPE)	DESCRIPTION	DIMENSIONS	TIME GRANULARITIES	LEGACY METRIC MAPPING	USAGE

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
------------------------------	-------------------------	-------------	------------	--------------------	-----------------------	-------

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 Cassandra API requests	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.

# Use the Azure Cosmos Emulator for local development and testing

2/25/2020 • 21 minutes to read • [Edit Online](#)

The Azure Cosmos Emulator provides a local environment that emulates the Azure Cosmos DB service for development purposes. Using the Azure Cosmos 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 Emulator, you can switch to using an Azure Cosmos account in the cloud.

You can develop using Azure Cosmos Emulator with [SQL](#), [Cassandra](#), [MongoDB](#), [Gremlin](#), and [Table API](#) accounts. However at this time the Data Explorer view in the emulator fully supports clients for SQL API only.

## How the emulator works

The Azure Cosmos Emulator provides a high-fidelity emulation of the Azure Cosmos DB service. It supports identical functionality as Azure Cosmos DB, including support for creating and querying data, provisioning and scaling containers, and executing stored procedures and triggers. You can develop and test applications using the Azure Cosmos Emulator, and deploy them to Azure at global scale by just making a single configuration change to the connection endpoint for Azure Cosmos DB.

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 Azure infrastructure like global replication, single-digit millisecond latency for reads/writes, and tunable consistency levels are not applicable.

You can migrate data between the Azure Cosmos Emulator and the Azure Cosmos DB service by using the [Azure Cosmos DB Data Migration Tool](#).

You can run Azure Cosmos Emulator on the Windows Docker container, see the [Docker Hub](#) for the docker pull command and [GitHub](#) for the `Dockerfile` and more information.

## Differences between the emulator and the service

Because the Azure Cosmos Emulator provides an emulated environment running on the local developer workstation, there are some differences in functionality between the emulator and an Azure Cosmos account in the cloud:

- Currently Data Explorer in the emulator supports clients for SQL API. The Data Explorer view and operations for Azure Cosmos DB APIs such as MongoDB, Table, Graph, and Cassandra APIs are not fully supported.
- The Azure Cosmos Emulator supports only a single fixed account and a well-known master key. Key regeneration is not possible in the Azure Cosmos Emulator, however the default key can be changed using the command-line option.
- The Azure Cosmos Emulator is not a scalable service and will not support a large number of containers.
- The Azure Cosmos Emulator does not offer different [Azure Cosmos DB consistency levels](#).
- The Azure Cosmos Emulator does not offer [multi-region replication](#).
- As your copy of the Azure Cosmos Emulator might not always be up-to-date with the most recent changes in the Azure Cosmos DB service, you should refer to the [Azure Cosmos DB capacity planner](#) to accurately estimate the production throughput (RUs) needs of your application.

- When using the Azure Cosmos Emulator, 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 Emulator. For more information about changing this value, see [Setting the PartitionCount value](#).

## System requirements

The Azure Cosmos Emulator has the following hardware and software requirements:

- Software requirements
  - Windows Server 2012 R2, Windows Server 2016, or Windows 10
  - 64-bit operating system
- Minimum Hardware requirements
  - 2-GB RAM
  - 10-GB available hard disk space

## Installation

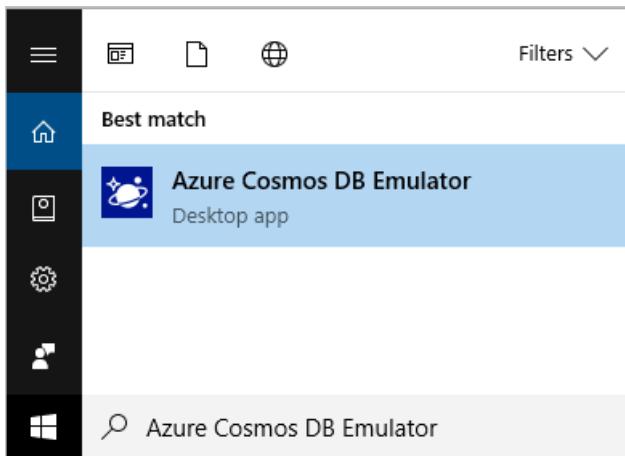
You can download and install the Azure Cosmos Emulator from the [Microsoft Download Center](#) or you can run the emulator on Docker for Windows. For instructions on using the emulator on Docker for Windows, see [Running on Docker](#).

### NOTE

To install, configure, and run the Azure Cosmos Emulator, you must have administrative privileges on the computer. The emulator will create/add a certificate and also set the firewall rules in order to run its services; therefore it's necessary for the emulator to be able to execute such operations.

## Running on Windows

To start the Azure Cosmos Emulator, select the Start button or press the Windows key. Begin typing **Azure Cosmos Emulator**, and select the emulator from the list of applications.



When the emulator is running, you'll see an icon in the Windows taskbar notification area.

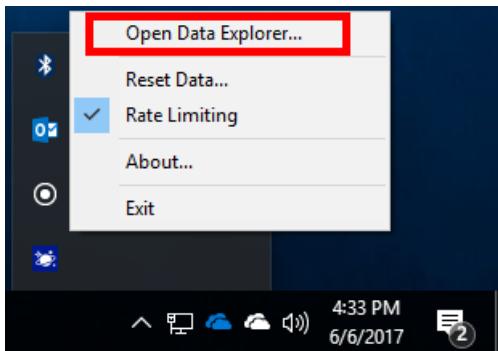


The Azure Cosmos Emulator by default runs on the local machine ("localhost") listening on port 8081.

The Azure Cosmos Emulator is installed to `C:\Program Files\Azure Cosmos DB Emulator` by default. You can also start and stop the emulator from the command-line. For more information, see the [command-line tool reference](#).

## Start Data Explorer

When the Azure Cosmos Emulator launches, it automatically opens the Azure Cosmos Data Explorer in your browser. 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 Emulator in the Windows Tray Icon as shown below.



## Checking for updates

Data Explorer indicates if there is a new update available for download.

### NOTE

Data created in one version of the Azure Cosmos Emulator (see %LOCALAPPDATA%\CosmosDBEmulator or data path optional settings) 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, rather than in the Azure Cosmos Emulator.

## Authenticating requests

As with Azure Cosmos DB in the cloud, every request that you make against the Azure Cosmos Emulator must be authenticated. The Azure Cosmos Emulator supports a single fixed account and a well-known authentication key for master key authentication. This account and key are the only credentials permitted for use with the Azure Cosmos Emulator. They are:

```
Account name: localhost:<port>
Account key: C2y6yDjf5/R+ob0N8A7Cgv30VRDJWEHLM+4QDU5DE2nQ9nDuVTqobD4b8mGGyPMbIZnqyMsEcaGQy67XIw/Jw==
```

### NOTE

The master key supported by the Azure Cosmos Emulator is intended for use only with the emulator. You cannot use your production Azure Cosmos DB account and key with the Azure Cosmos Emulator.

### NOTE

If you have started the emulator with the /Key option, then use the generated key instead of `C2y6yDjf5/R+ob0N8A7Cgv30VRDJWEHLM+4QDU5DE2nQ9nDuVTqobD4b8mGGyPMbIZnqyMsEcaGQy67XIw/Jw==`. For more information about /Key option, see [Command-line tool reference](#).

As with the Azure Cosmos DB, the Azure Cosmos Emulator supports only secure communication via SSL.

# Running 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).

## Developing with the emulator

### SQL API

Once you have the Azure Cosmos 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 Emulator also includes a built-in Data Explorer that lets you create containers for SQL API or Cosmos DB for Mongo DB API, and view and edit items without writing any code.

```
// Connect to the Azure Cosmos Emulator running locally
DocumentClient client = new DocumentClient(
    new Uri("https://localhost:8081"),
    "C2y6yDjf5/R+ob0N8A7Cgv30VRDJWEHLM+4QDU5DE2nQ9nDuVTqobD4b8mGGyPMbIZnqyMsEcaGQy67XIw/Jw==");
```

### Azure Cosmos DB's API for MongoDB

Once you have the Azure Cosmos Emulator running on your desktop, you can use the [Azure Cosmos DB's API for MongoDB](#) to interact with the emulator. Start 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+ob0N8A7Cgv30VRDJWEHLM+4QDU5DE2nQ9nDuVTqobD4b8mGGyPMbIZnqyMsEcaGQy67XIw/Jw==@localhost:10255/admin?ssl=true
```

### Table API

Once you have the Azure Cosmos Emulator running on your desktop, you can use the [Azure Cosmos DB Table API SDK](#) to interact with the emulator. Start 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+ob0N8A7Cgv30VRDJWEHLM+4QDU5DE2nQ9nDuVTqobD4b8mGGyPMbIZnqyMsEcaGQy67XIw/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`.

- [Install Python 2.7](#)
- [Install Cassandra CLI/CQLSH](#)
- 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+ob0N8A7Cgv30VRDJWEHLM+4QDU5DE2nQ9nDuVTqobD4b8mGGyPMbIZnqyMsEcaGQy67XIw/Jw== --ssl
```

- 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 prompt with "/EnableGremlinEndpoint". Alternatively you can also set the environment variable `AZURE_COSMOS_EMULATOR_GREMLIN_ENDPOINT=true`

- [Install apache-tinkerpop-gremlin-console-3.3.4.](#)
- In the emulator's Data Explorer create a database "db1" and a collection "coll1"; for the partition key, choose "/name"
- 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+ob0N8A7Cgv30VRDJWEHLM+4QDU5DE2nQ9nDuVTqobD4b8mGGyPMbIZnqyMsEcaGQy67XIw/Jw==
connectionPool: {
enableSsl: false}
serializer: { className: org.apache.tinkerpop.gremlin.driver.ser.GraphSONMessageSerializerV1d0,
config: { serializeResultToString: true }}
```

```
bin\gremlin.bat
```

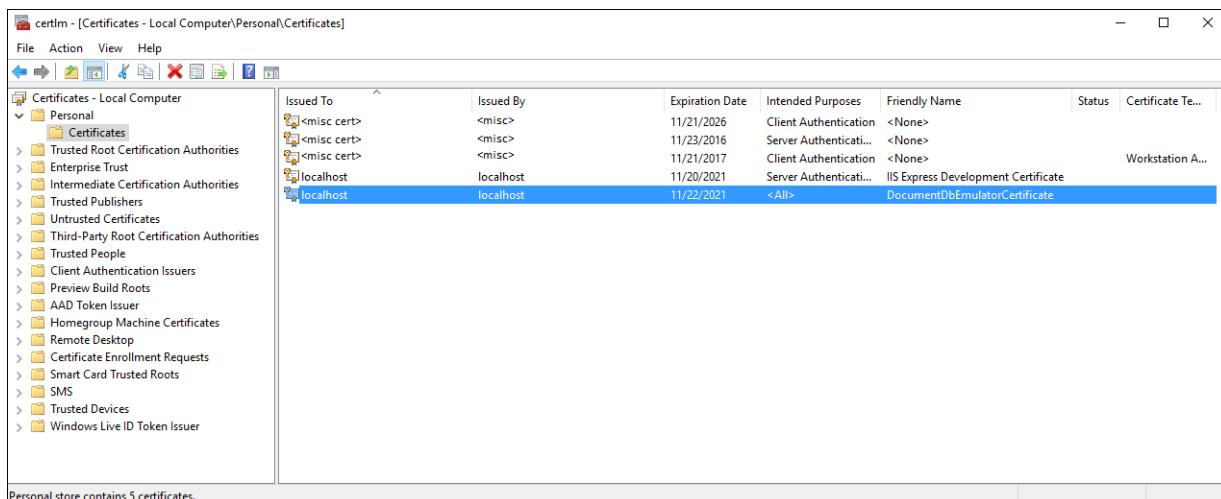
- In the Gremlin shell run the following commands to connect to the Gremlin endpoint:

```
: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()
```

## Export the SSL certificate

.NET languages and runtime use the Windows Certificate Store to securely connect to the Azure Cosmos DB local emulator. Other languages have their own method of managing and using certificates. Java uses its own [certificate store](#) whereas Python uses [socket wrappers](#).

In order to obtain a certificate to use with languages and runtimes that do not integrate with the Windows Certificate Store, you will need to export it using the Windows Certificate Manager. You can start it by running certlm.msc or follow the step by step instructions in [Export the Azure Cosmos Emulator Certificates](#). Once the certificate manager is running, open the Personal Certificates as shown below and export the certificate with the friendly name "DocumentDBEmulatorCertificate" as a BASE-64 encoded X.509 (.cer) file.



The X.509 certificate can be imported into the Java certificate store by following the instructions in [Adding a Certificate to the Java CA Certificates Store](#). Once 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 Emulator.

When connecting to the emulator from Python and Node.js SDKs, SSL verification is disabled.

## Command-line tool reference

From the installation location, you can use the command-line to start and stop the emulator, configure options, and perform other operations.

### 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
--------	-------------	---------	-----------

OPTION	DESCRIPTION	COMMAND	ARGUMENTS
[No arguments]	Starts up the Azure Cosmos Emulator with default settings.	Microsoft.Azure.Cosmos.E mulator.exe	
[Help]	Displays the list of supported command-line arguments.	Microsoft.Azure.Cosmos.E mulator.exe /?	
GetStatus	Gets the status of the Azure Cosmos 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.E mulator.exe /GetStatus	
Shutdown	Shuts down the Azure Cosmos Emulator.	Microsoft.Azure.Cosmos.E mulator.exe /Shutdown	
DataPath	Specifies the path in which to store data files. Default value is %LocalAppdata%\CosmosDBEmulator.	Microsoft.Azure.Cosmos.E mulator.exe /DataPath=<datopath>	<datopath>: An accessible path
Port	Specifies the port number to use for the emulator. Default value is 8081.	Microsoft.Azure.Cosmos.E mulator.exe /Port=<port>	<port>: Single port number
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.E mulator.exe /ComputePort=<computeport>	<computeport>: Single port number
EnableMongoDbEndpoint=3.2	Enables MongoDB API 3.2	Microsoft.Azure.Cosmos.E mulator.exe /EnableMongoDbEndpoint=3.2	
EnableMongoDbEndpoint=3.6	Enables MongoDB API 3.6	Microsoft.Azure.Cosmos.E mulator.exe /EnableMongoDbEndpoint=3.6	
MongoPort	Specifies the port number to use for MongoDB compatibility API. Default value is 10255.	Microsoft.Azure.Cosmos.E mulator.exe /MongoPort=<mongoport>	<mongoport>: Single port number

OPTION	DESCRIPTION	COMMAND	ARGUMENTS
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
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	

OPTION	DESCRIPTION	COMMAND	ARGUMENTS
FailOnSslCertificateNameMismatch	By default the Emulator regenerates its self-signed 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 SSL certificate.	Microsoft.Azure.Cosmos.Emulator.exe /FailOnSslCertificateNameMismatch	
GenCert	Generate and install a new self-signed 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
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 <a href="#">Change the number of containers</a> 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.

OPTION	DESCRIPTION	COMMAND	ARGUMENTS
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 <a href="#">consistency levels</a> : Session, Strong, Eventual, or BoundedStaleness. The default value is Session.
?	Show the help message.		

## Change the number of 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 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 Emulator, run the following steps:

1. Delete all local Azure Cosmos 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 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 .
```

## Controlling the emulator

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 `PSModulePath` and import it as shown in the following command:

```
$env:PSModulePath += "$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.

## Running on Docker

The Azure Cosmos Emulator can be run on Docker for Windows. The emulator does not work on Docker for Oracle Linux.

Once 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**.

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
```

To start the image, run the following commands.

From the command-line:

```
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
```

### NOTE

If you see a port conflict error (specified port is already in use) when you run the docker run command, you can pass a custom port by altering the port numbers. For example, you can change the "-p 8081:8081" to "-p 443:8081"

From PowerShell:

```
md $env:LOCALAPPDATA\CosmosDBEmulator\bind-mount 2>null

docker run --name azure-cosmosdb-emulator --memory 2GB --mount
"type=bind,source=$env: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
```

The response looks similar to the following:

```
Starting emulator
Emulator Endpoint: https://172.20.229.193:8081/
Master Key: C2y6yDjf5/R+ob0N8A7Cgv30VRDJWEHLM+4QDU5DE2nQ9nDuVTqobD4b8mGGyPMbIZnqyMsEcaGQy67XIw/Jw==
Exporting SSL Certificate
You can import the SSL certificate from an administrator command prompt on the host by running:
cd /d %LOCALAPPDATA%\CosmosDBEmulatorCert
powershell .\importcert.ps1
-----
Starting interactive shell
```

Now use the endpoint and master key-in from the response in your client and import the SSL certificate into your host. To import the SSL certificate, do the following from an admin command prompt:

From the command-line:

```
cd %LOCALAPPDATA%\CosmosDBEmulator\bind-mount
powershell .\importcert.ps1
```

From PowerShell:

```
cd $env:LOCALAPPDATA\CosmosDBEmulator\bind-mount
.\importcert.ps1
```

Closing the interactive shell once the emulator has been started will shut down the emulator's container.

To open 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 emulator on a host machine, please follow the below section for Linux to import the certificate into the Linux docker container.

## Running on Mac or Linux

Currently the Cosmos emulator can only be run on Windows. Users running Mac or Linux can run the emulator in a Windows virtual machine hosted a hypervisor such as Parallels or VirtualBox. Below are the steps to enable this.

Within the Windows VM run the command below and make note of the IPv4 address.

```
ipconfig.exe
```

Within your application you need to change the URI used as Endpoint to use the IPv4 address returned by `ipconfig.exe` instead of `localhost`.

The next step, from the within the Windows VM, launch the Cosmos emulator from the command line using the following options.

```
Microsoft.Azure.Cosmos Emulator.exe /AllowNetworkAccess
/Key=C2y6yDjf5/R+ob0N8A7Cgv30VRDJWEHLM+4QDU5DE2nQ9nDuVTqobD4b8mGGyPMbIZnqyMsEcaGQy67XIw/Jw==
```

Finally, we need to import the Emulator CA certificate into the Linux or Mac environment.

## Linux

If you are working on Linux, .NET relays on OpenSSL to do the validation:

1. [Export the certificate in PFX format](#) (PFX 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/
```

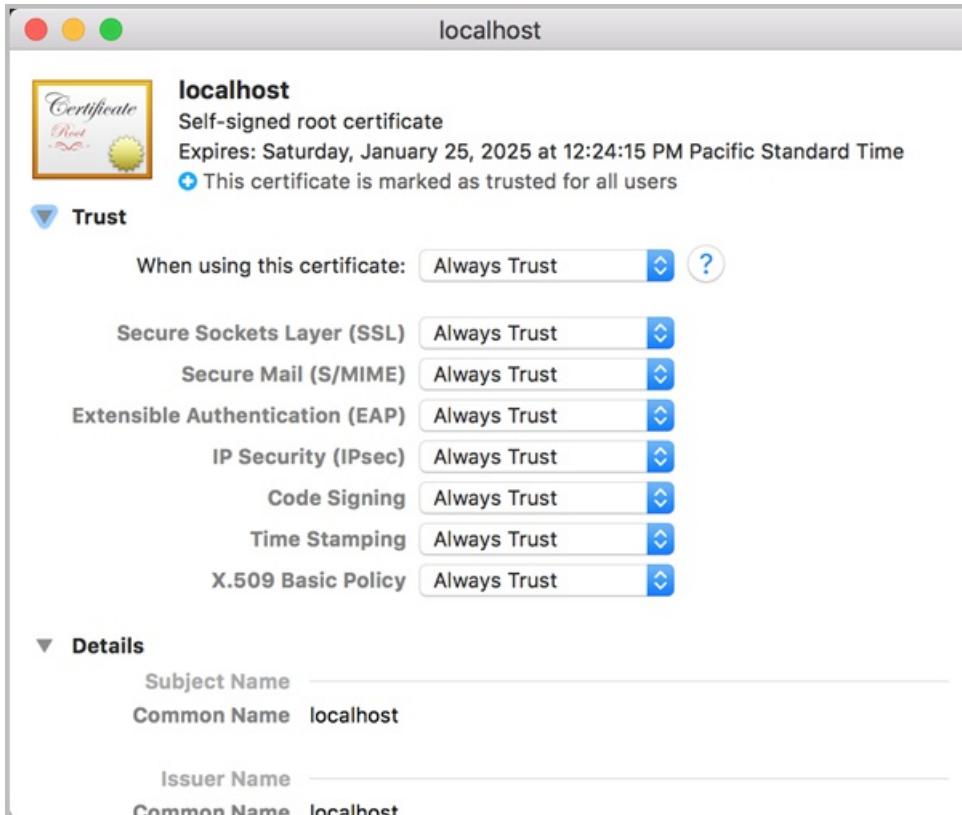
5. Update the CA certificates, which will update the `/etc/ssl/certs/` folder.

```
update-ca-certificates
```

## Mac OS

Use the following steps if you are working on Mac:

1. [Export the certificate in PFX format](#) (PFX 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*.



After following these steps, your environment will trust the certificate used by the Emulator when connecting to the IP address exposes by `/AllowNetworkAccess`.

## Troubleshooting

Use the following tips to help troubleshoot issues you encounter with the Azure Cosmos 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 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.
- If the Azure Cosmos Emulator crashes, collect 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
```

- 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.
- 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. `cd /d "%ProgramFiles%\Azure Cosmos DB Emulator"`
2. `Microsoft.Azure.Cosmos Emulator.exe /shutdown`. Watch the system tray to make sure the program has shut down, it may take a minute. You can also just click **Exit** in the Azure Cosmos Emulator user interface.
3. `Microsoft.Azure.Cosmos Emulator.exe /startwprtraces`
4. `Microsoft.Azure.Cosmos Emulator.exe`
5. Reproduce the problem. If Data Explorer is not working, you only need to wait for the browser to open for a few seconds to catch the error.
6. `Microsoft.Azure.Cosmos Emulator.exe /stopwprtraces`
7. Navigate to `%ProgramFiles%\Azure Cosmos DB Emulator` 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.

### Uninstall the local emulator

1. Exit all open instances of the local emulator by right-clicking the Azure Cosmos Emulator icon on the system tray, and then clicking Exit. It may take a minute for all instances to exit.
2. In the Windows search box, type **Apps & features** and click on the **Apps & features (System settings)**

result.

3. In the list of apps, scroll to **Azure Cosmos DB Emulator**, select it, click **Uninstall**, then confirm and click **Uninstall** again.
4. When the app is uninstalled, navigate to `%LOCALAPPDATA%\CosmosDBEmulator` and delete the folder.

## Next steps

In this tutorial, you've learned how to use the local emulator for free local development. You can now proceed to the next tutorial and learn how to export emulator SSL certificates.

[Export the Azure Cosmos Emulator certificates](#)

# Export the Azure Cosmos DB Emulator certificates for use with Java, Python, and Node.js

5/24/2019 • 3 minutes to read • [Edit Online](#)

## Download the Emulator

The Azure Cosmos DB Emulator provides a local environment that emulates the Azure Cosmos DB service for development purposes including its use of SSL connections. This post demonstrates how to export the SSL certificates for use in languages and runtimes that do not integrate with the Windows Certificate Store such as Java which uses its own [certificate store](#) and Python which uses [socket wrappers](#) and Node.js which uses [tlsSocket](#). You can read more about the emulator in [Use the Azure Cosmos DB Emulator for development and testing](#).

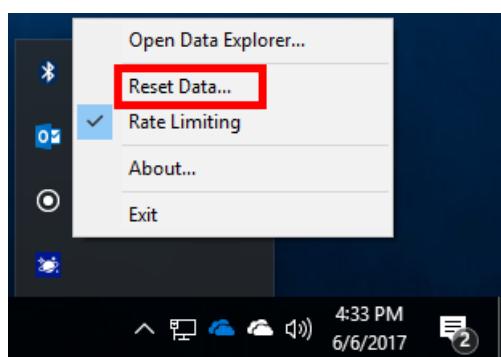
This tutorial covers the following tasks:

- Rotating certificates
- Exporting SSL certificate
- Learning how to use the certificate in Java, Python, and Node.js

## Certification rotation

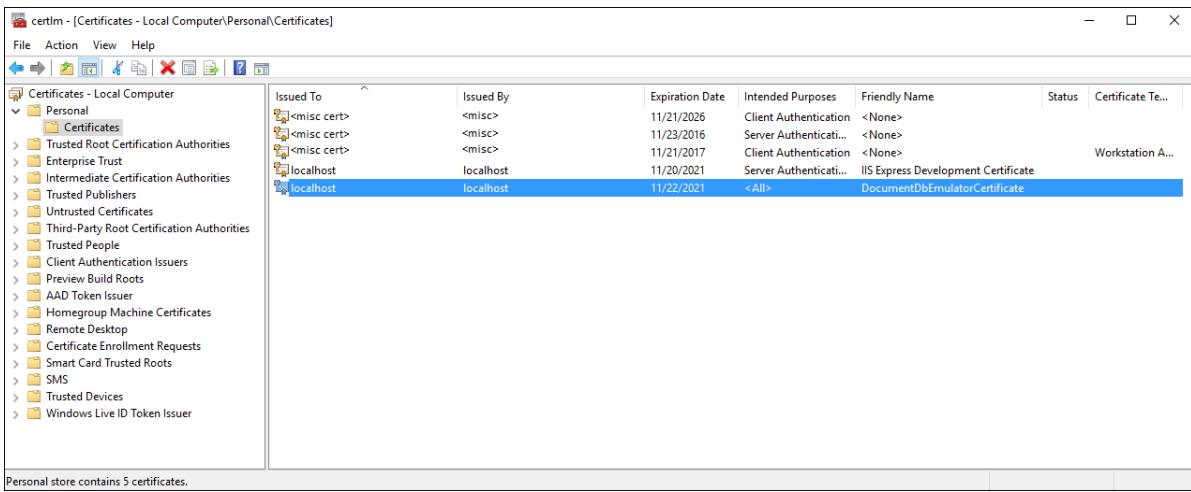
Certificates in the Azure Cosmos DB Local Emulator are generated the first time the emulator is run. There are two certificates. One used for connecting to the local emulator and one for managing secrets within the emulator. The certificate you want to export is the connection certificate with the friendly name "DocumentDBEmulatorCertificate".

Both certificates can be regenerated by clicking **Reset Data** as shown below from Azure Cosmos DB Emulator running in the Windows Tray. If you regenerate the certificates and have installed them into the Java certificate store or used them elsewhere you will need to update them, otherwise your application will no longer connect to the local emulator.

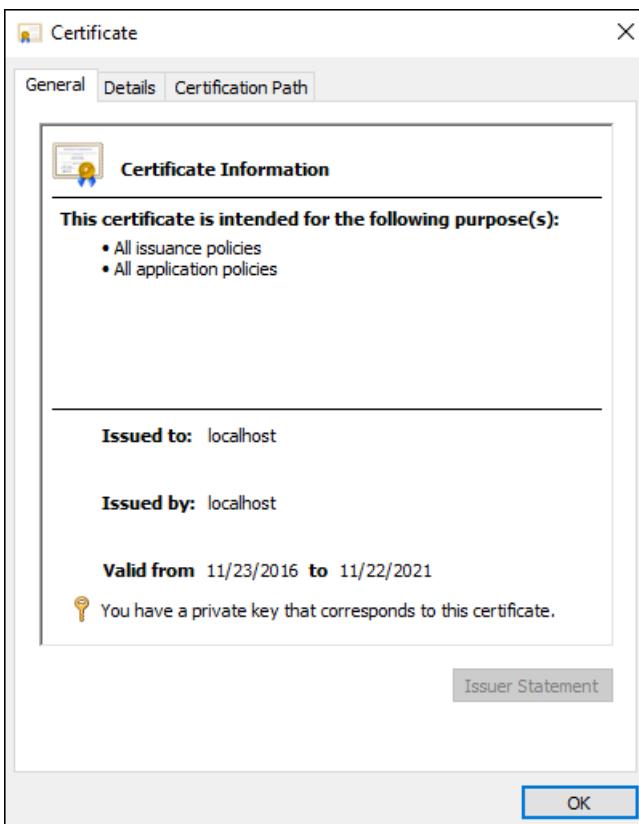


## How to export the Azure Cosmos DB SSL certificate

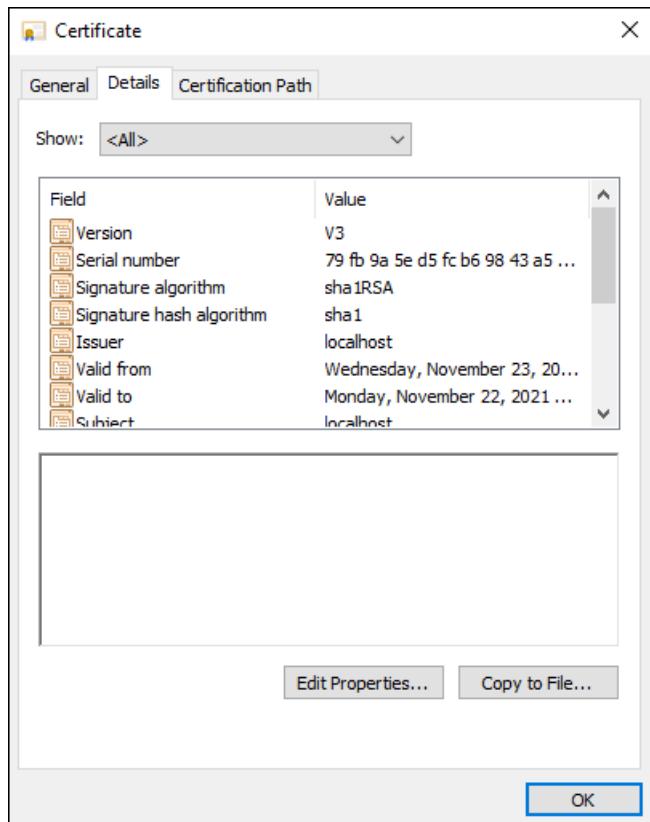
1. Start the Windows Certificate manager by running certlm.msc and navigate to the Personal->Certificates folder and open the certificate with the friendly name **DocumentDbEmulatorCertificate**.



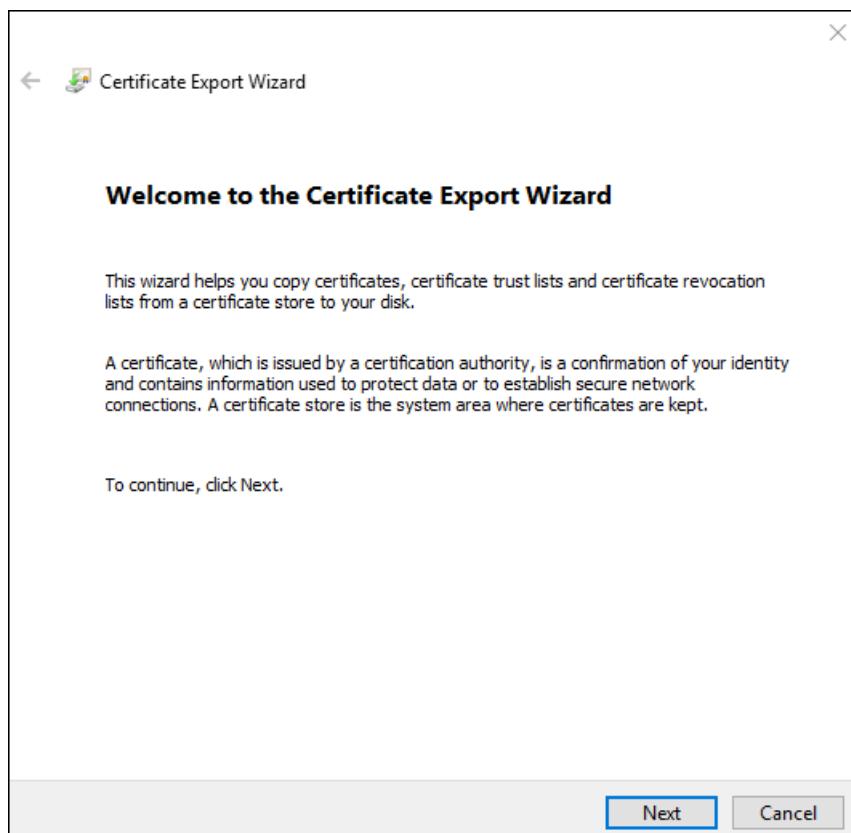
2. Click on **Details** then **OK**.



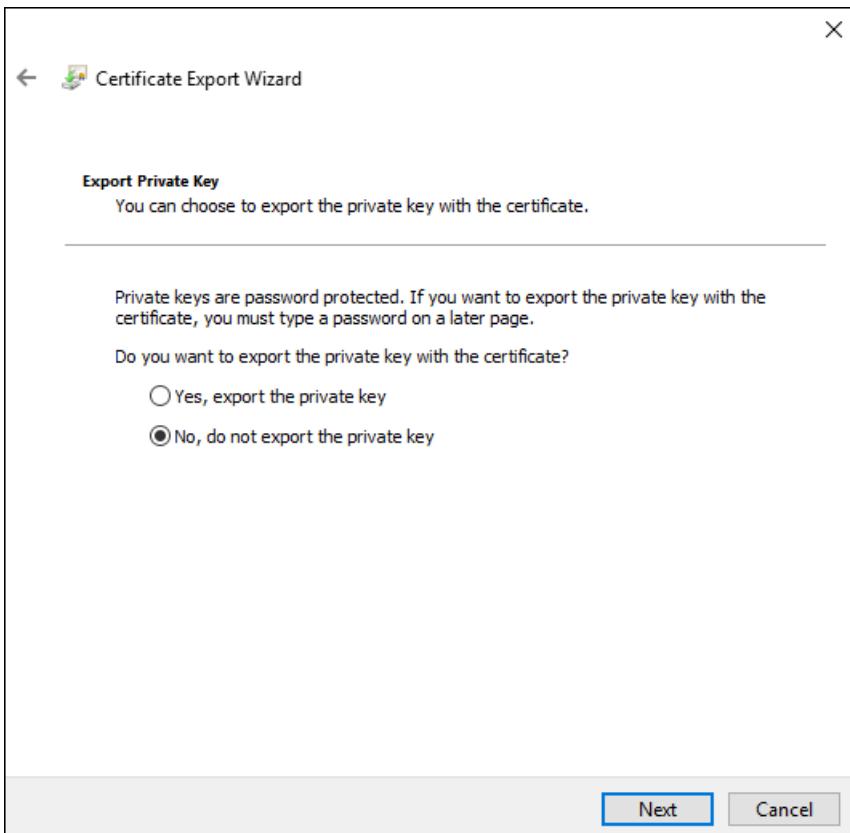
3. Click **Copy to File....**



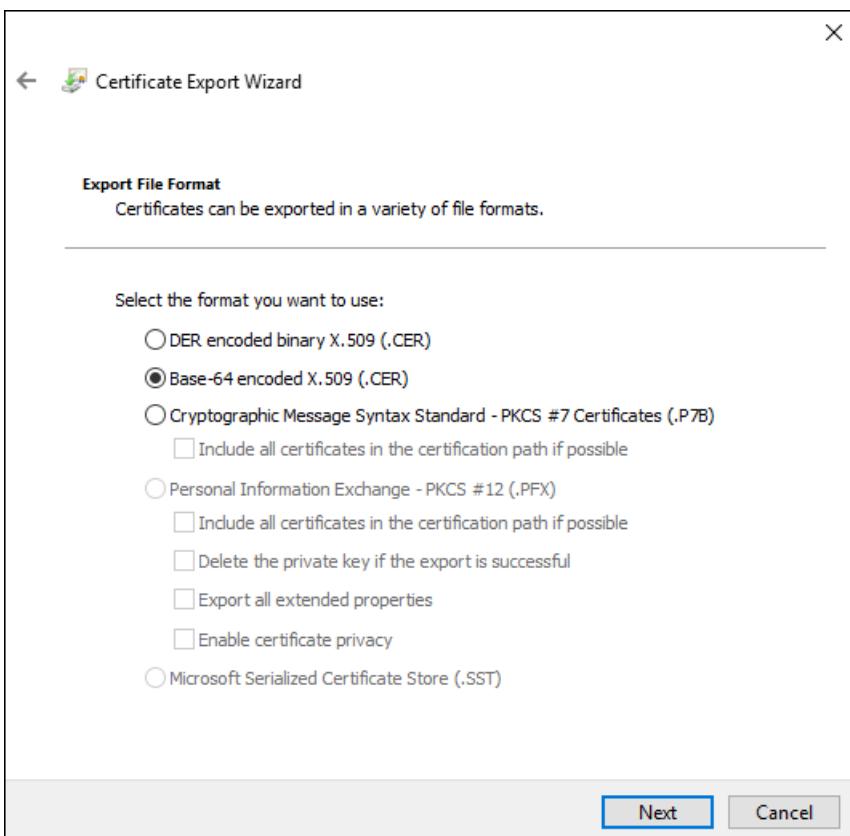
4. Click **Next**.



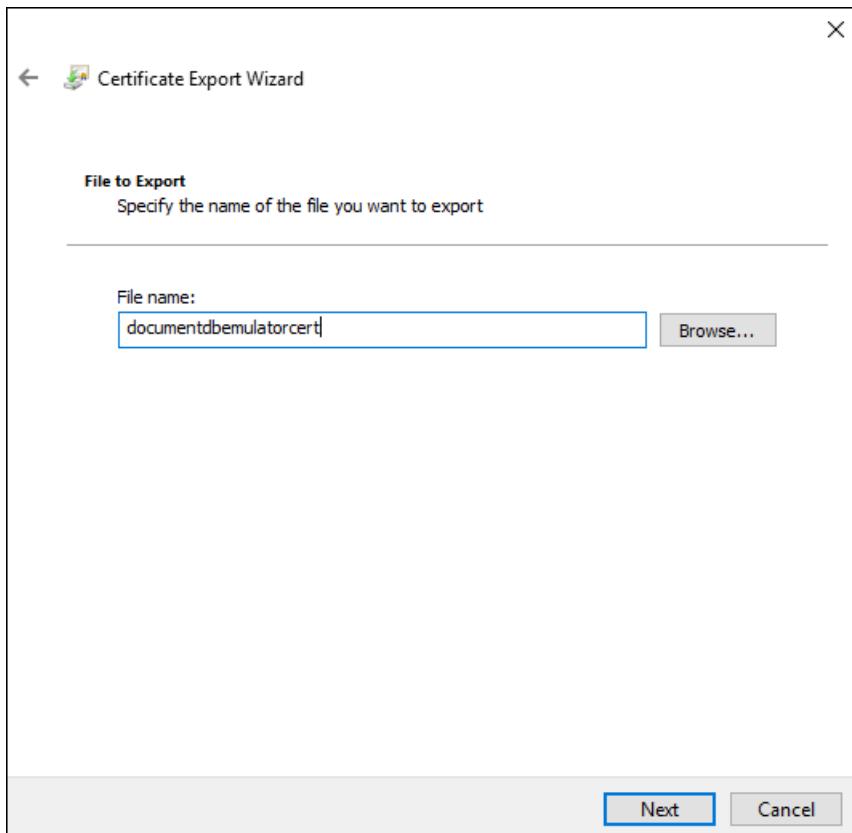
5. Click **No, do not export private key**, then click **Next**.



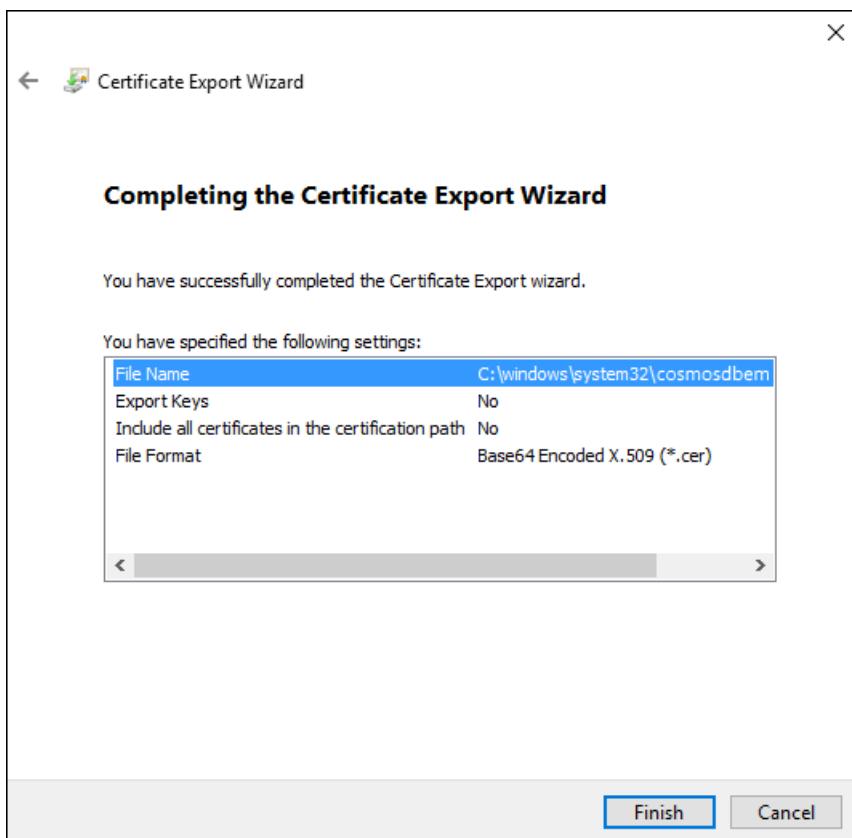
6. Click on **Base-64 encoded X.509 (.CER)** and then **Next**.



7. Give the certificate a name. In this case **documentdbemulatorcert** and then click **Next**.



8. Click **Finish**.



## How to use the certificate in Java

When running Java applications or MongoDB applications that use the Java 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](#) depends on the default certificate store.

Follow the instructions in the [Adding a Certificate to the Java CA 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.

Once the "CosmosDBEmulatorCertificate" SSL certificate is installed your application should be able to connect and use the local Azure Cosmos DB Emulator. If you continue to have trouble you may want to follow the [Debugging SSL/TLS Connections](#) article. It is very likely the certificate is not 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.

## How to use the certificate in Python

By default the [Python SDK\(version 2.0.0 or higher\)](#) for the SQL API will not try and use the SSL certificate when connecting to the local emulator. If however you want to use SSL validation you can follow the examples in the [Python socket wrappers](#) documentation.

## How to use the certificate in Node.js

By default the [Nodejs SDK\(version 1.10.1 or higher\)](#) for the SQL API will not try and use the SSL certificate when connecting to the local emulator. If however you want to use SSL validation you can follow the examples in the [Nodejs documentation](#).

## Next steps

In this tutorial, you've done the following:

- Rotated certificates
- Exported the SSL certificate
- Learned how to use the certificate in Java, Python and Node.js

You can now proceed to the concepts section for more information about Azure Cosmos DB.

[Tunable data consistency levels in Azure Cosmos DB](#)

# Set up a CI/CD pipeline with the Azure Cosmos DB emulator build task in Azure DevOps

2/21/2020 • 6 minutes to read • [Edit Online](#)

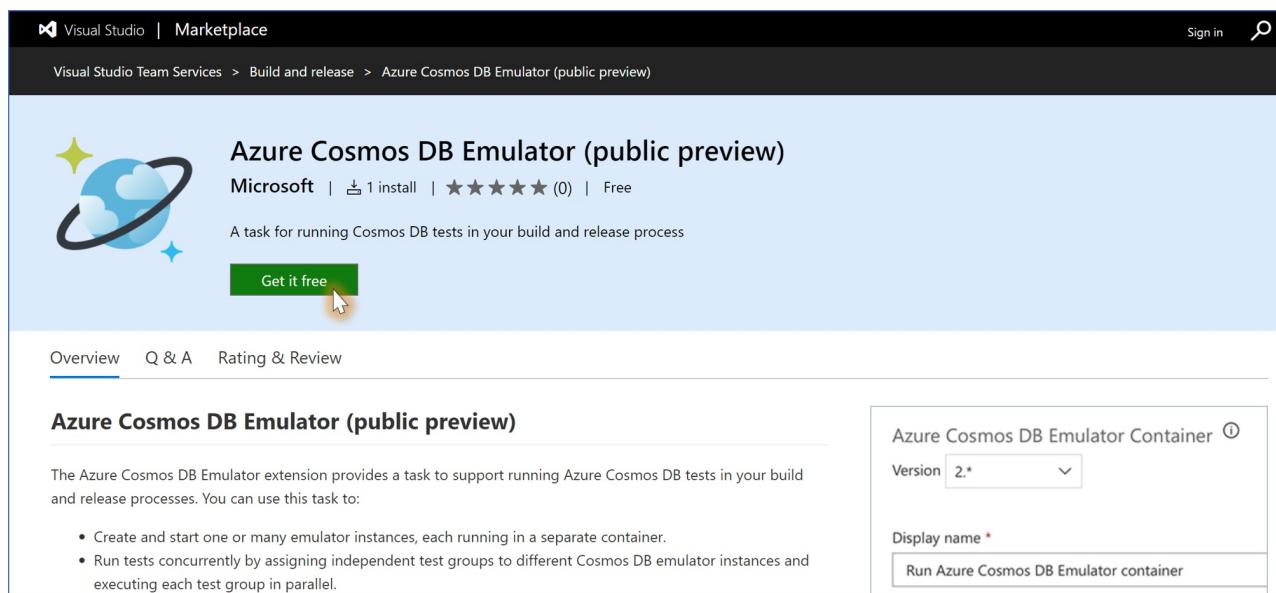
The Azure Cosmos DB emulator provides a local environment that emulates the Azure Cosmos DB service for development purposes. The emulator allows you to develop and test your application locally, without creating an Azure subscription or incurring any costs.

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**.

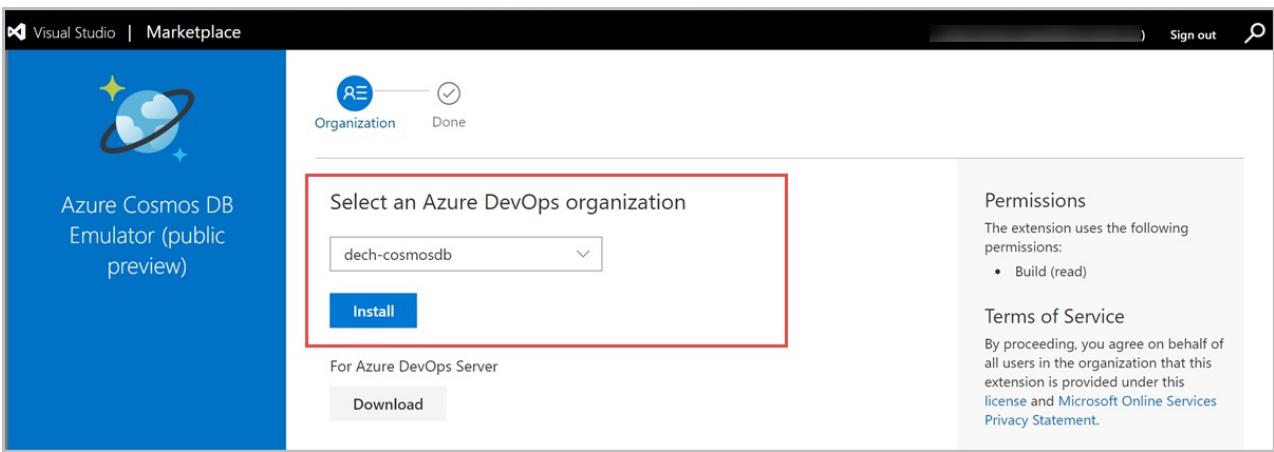


The screenshot shows the Azure Cosmos DB Emulator (public preview) extension page in the Visual Studio Marketplace. The page includes the extension icon (a blue planet with clouds), its name, developer (Microsoft), download count (1 install), rating (0 stars), and status (Free). A 'Get it free' button is prominently displayed. Below the main heading, there's a brief description: 'A task for running Cosmos DB tests in your build and release process'. At the bottom of the main section, there are links for 'Overview', 'Q & A', and 'Rating & Review'. To the right, there's a configuration panel titled 'Azure Cosmos DB Emulator Container' with fields for 'Version' (set to 2.\*), 'Display name' (with a placeholder 'Run Azure Cosmos DB Emulator container'), and a note about running the container.

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 account 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**

A screenshot of the Azure DevOps Pipelines screen. The left sidebar shows 'FabrikamProject' with 'Builds' highlighted by a red box. The main area shows a list of pipelines: 'Fabrikam' (master), 'FabrikamProject-ASP.NET-CI' (master), and 'FabrikamTodoApp-CI' (master). A context menu is open over the 'Fabrikam' pipeline, with 'New build pipeline' highlighted by a red box. Other options in the menu include 'Import a 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**

Select a source

Azure Repos Git GitHub GitHub Enterprise Subversion Bitbucket Cloud External Git

Team project

FabrikamProject

Repository

TodoApp

Default branch for manual and scheduled builds

master

Continue

- 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**

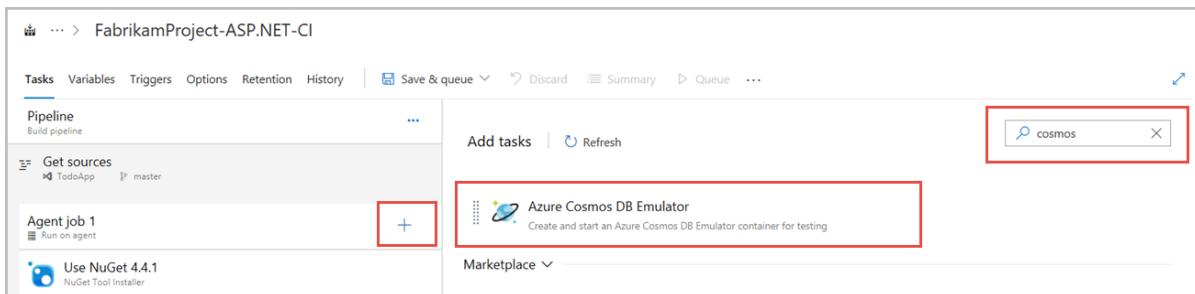
The agent pool to be selected for this CI should have Docker for Windows installed unless the installation is done manually in a prior task as a part of the CI. See [Microsoft hosted agents](#) article for a selection of agent pools; we recommend to start with `Hosted VS2017`.

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

- 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**.
- 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.

## 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+ob0N8A7Cgv30VRDJWEHLM+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"
            };

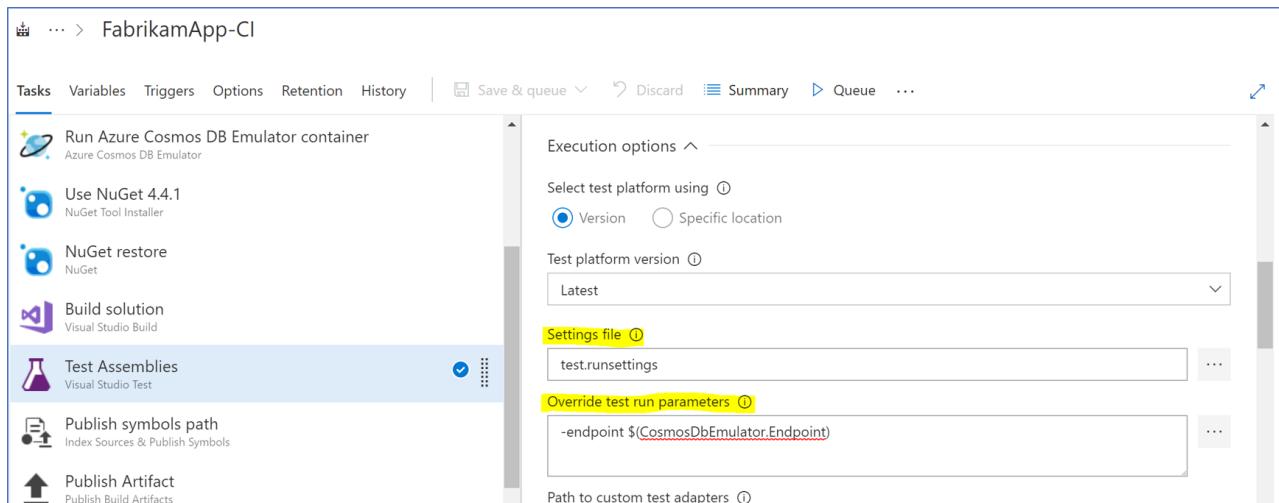
            // Create the item
            await DocumentDBRepository<Item>.CreateItemAsync(item);
            // Query for the item
            var returnedItem = await DocumentDBRepository<Item>.GetItemAsync(item.Id, item.Category);
            // Verify the item returned is correct.
            Assert.AreEqual(item.Id, returnedItem.Id);
            Assert.AreEqual(item.Category, returnedItem.Category);
        }

        [TestCleanup()]
        public void Cleanup()
        {
            DocumentDBRepository<Item>.Teardown();
        }
    }
}

```

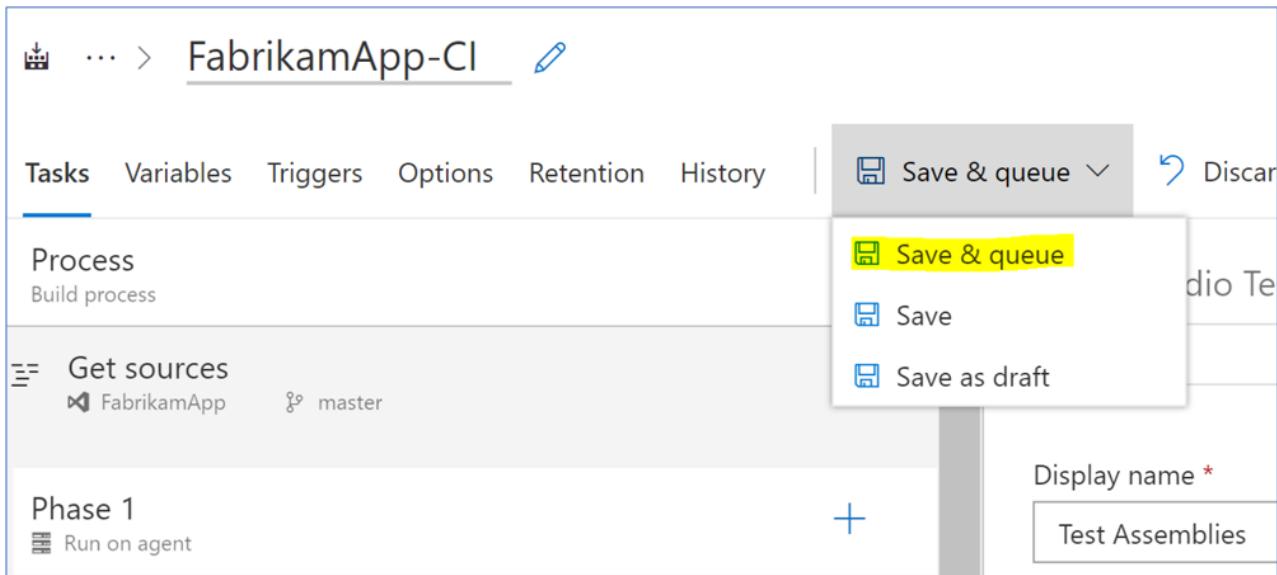
Navigate to the Execution Options in the Visual Studio Test task. In the **Settings file** option, specify that the tests are configured using the **.runsettings** file. In the **Override test run parameters** option, add in

`-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.



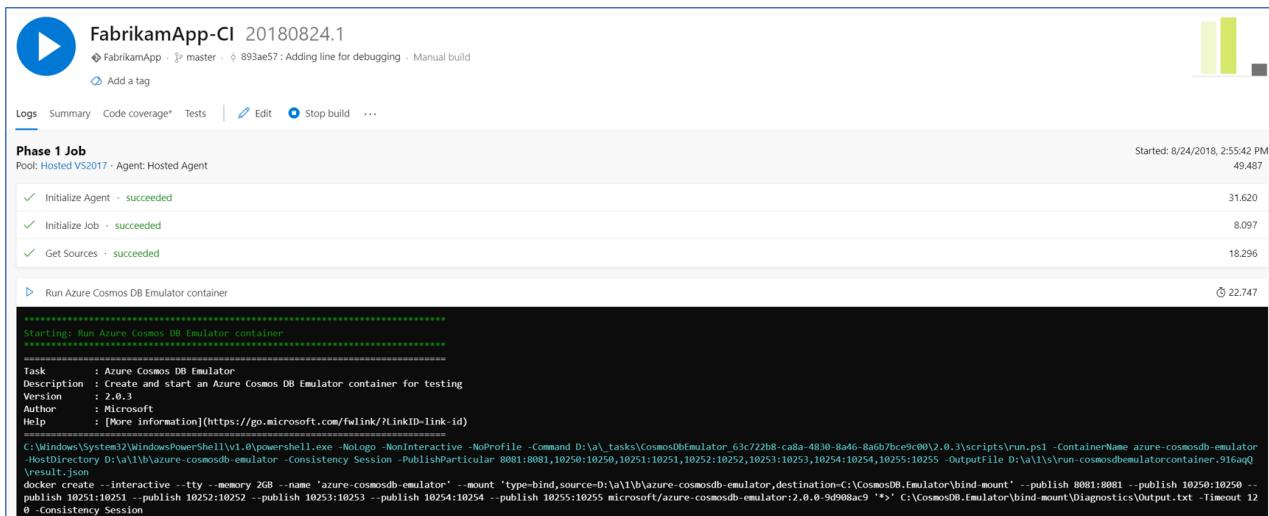
# Run the build

Now, **Save and queue** the build.



The screenshot shows the 'FabrikamApp-CI' build configuration in Azure DevOps. The 'Save & queue' button is highlighted in yellow. Other options shown are 'Save' and 'Save as draft'. The build process includes a 'Get sources' step for the 'FabrikamApp' repository, and a 'Phase 1' step with the 'Run on agent' option selected. A '+' button is available for adding more phases.

Once the build has started, observe the Cosmos DB emulator task has begun pulling down the Docker image with the emulator installed.

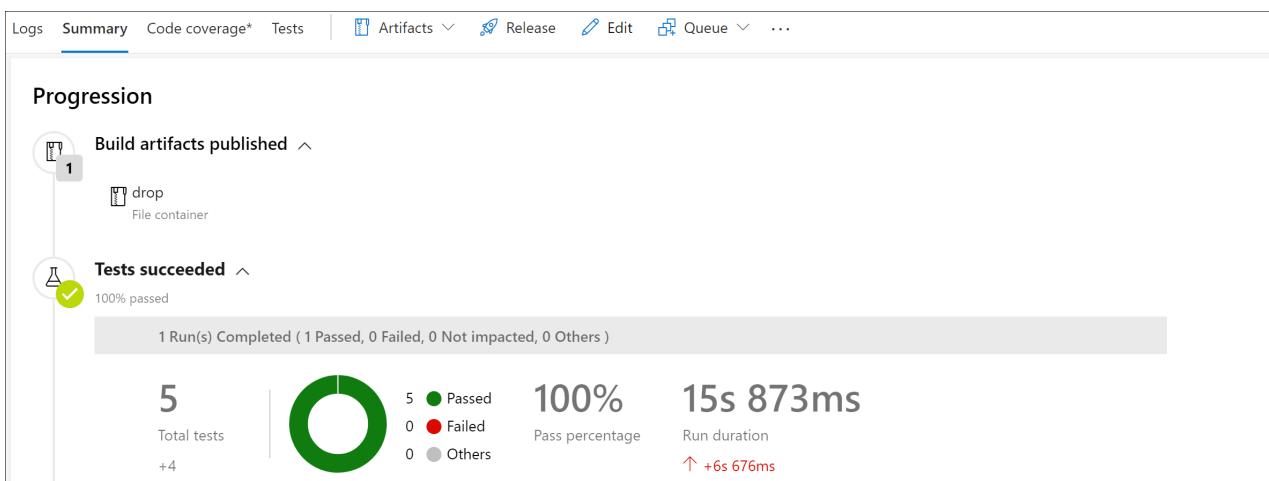


The screenshot shows the build log for 'FabrikamApp-CI'. The 'Run Azure Cosmos DB Emulator container' task is currently executing, as indicated by the progress bar at the top right. The log output shows the command being run to start the emulator container.

```
Starting: Run Azure Cosmos DB Emulator container
=====
Task : Azure Cosmos DB Emulator
Description : Create and start an Azure Cosmos DB Emulator container for testing
Version : 2.0.3
Author : Microsoft
Help : [More information](https://go.microsoft.com/fwlink/?LinkId=link-id)

C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe -NoLogo -NonInteractive -NoProfile -Command D:\a\Tasks\cosmosdbemulator_63c722b8-cafa-4830-8a46-8a6b7bce9c00\2.0.3\scripts\run.ps1 -ContainerName azure-cosmosdb-emulator -HostDirectory D:\a\1\azure-cosmosdb-emulator -Consistency Session -PublishParticular 8081:10250,10251:10251,10252:10252,10253:10253,10254:10254,10255:10255 -OutputFile D:\a\1\s\run-cosmosdbemulatorcontainer.916aqQ \result.json
docker create -interactive -tty -memory 2GB --name "azure-cosmosdb-emulator" --mount 'type=bind,source=D:\a\1\azure-cosmosdb-emulator,destination=C:\CosmosDB Emulator\bind-mount' --publish 8081:8081 --publish 10250:10250 --publish 10251:10251 --publish 10252:10252 --publish 10253:10253 --publish 10254:10254 --publish 10255:10255 microsoft/azure-cosmosdb-emulator:2.0.0-9d908ac9 *-> C:\CosmosDB Emulator\bind-mount\Diagnostics\Output.txt --Timeout 120 -Consistency Session
```

After the build completes, observe that your tests pass, all running against the Cosmos DB emulator from the build task!



The screenshot shows the build summary for 'FabrikamApp-CI'. It displays the progression of the build, including the publication of build artifacts and the successful completion of tests. The test results show 5 total tests, 0 failed, and 0 others, with a 100% pass percentage and a run duration of 15s 873ms.

Total tests	Passed	Failed	Others
5	5	0	0

Test results: 100% passed (1 Passed, 0 Failed, 0 Not impacted, 0 Others)

Progression: Build artifacts published (1), Tests succeeded (100% passed)

## Set up using YAML

If you are setting up the CI/CD pipeline by using a YAML task, 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+ob0N8A7Cgv30VRDJWEHLM+4QDU5DE2nQ9nDuVTqobD4b8mGKyPMbIZnqyMsEcaGQy67XIw/Jw==
    # The emulator uses a self-signed cert, disable TLS auth errors
    NODE_TLS_REJECT_UNAUTHORIZED: '0'
```

## 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 SSL certificates, see [Export the Azure Cosmos DB Emulator certificates for use with Java, Python, and Nodejs](#)

# Visualize Azure Cosmos DB data by using the Power BI connector

8/19/2019 • 8 minutes to read • [Edit Online](#)

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>.

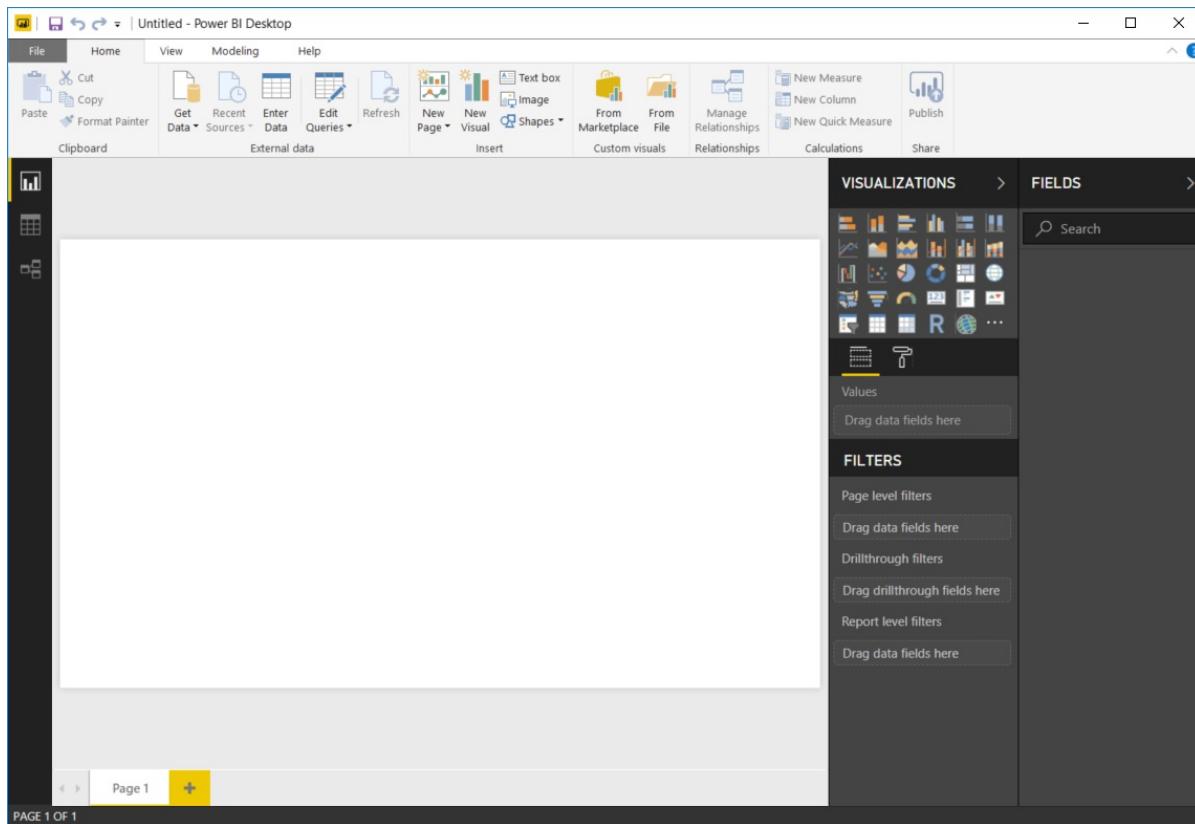
# Let's get started

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:

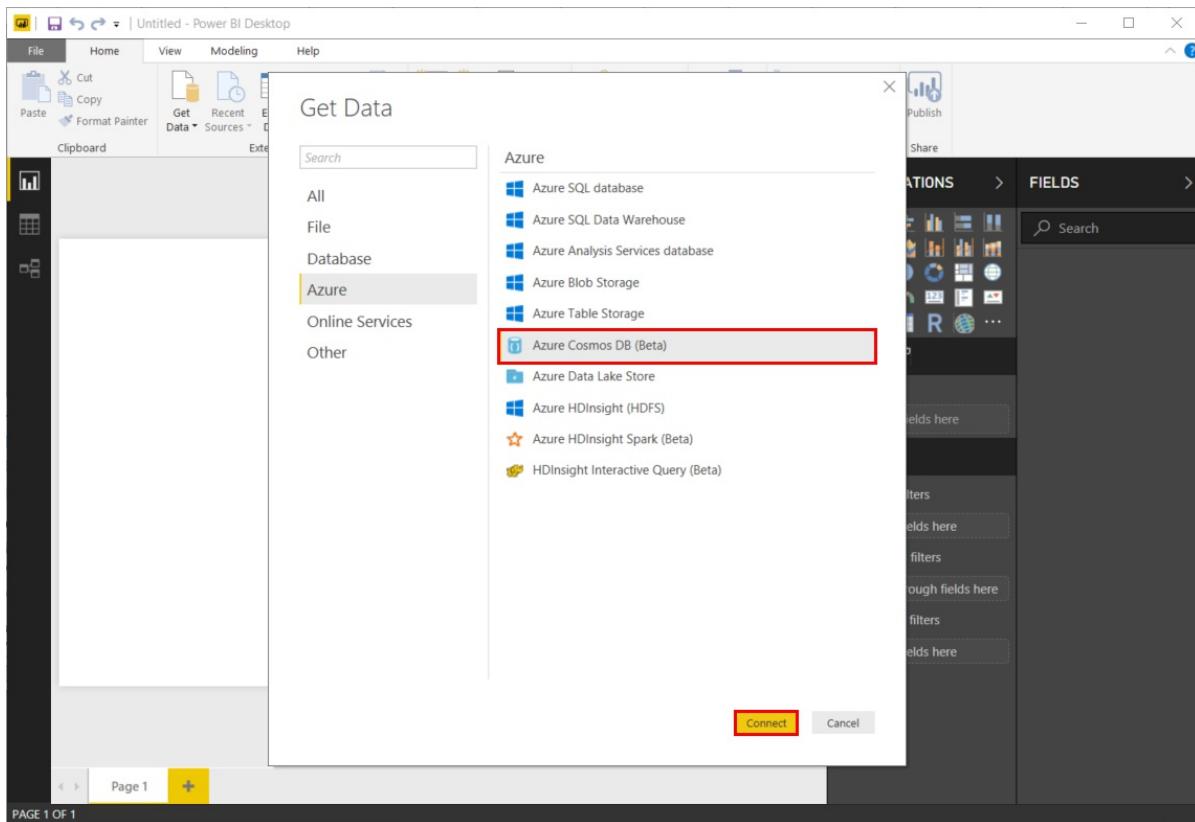
```
{  
    "Volcano Name": "Rainier",  
    "Country": "United States",  
    "Region": "US-Washington",  
    "Location": {  
        "type": "Point",  
        "coordinates": [  
            -121.758,  
            46.87  
        ]  
    },  
    "Elevation": 4392,  
    "Type": "Stratovolcano",  
    "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.

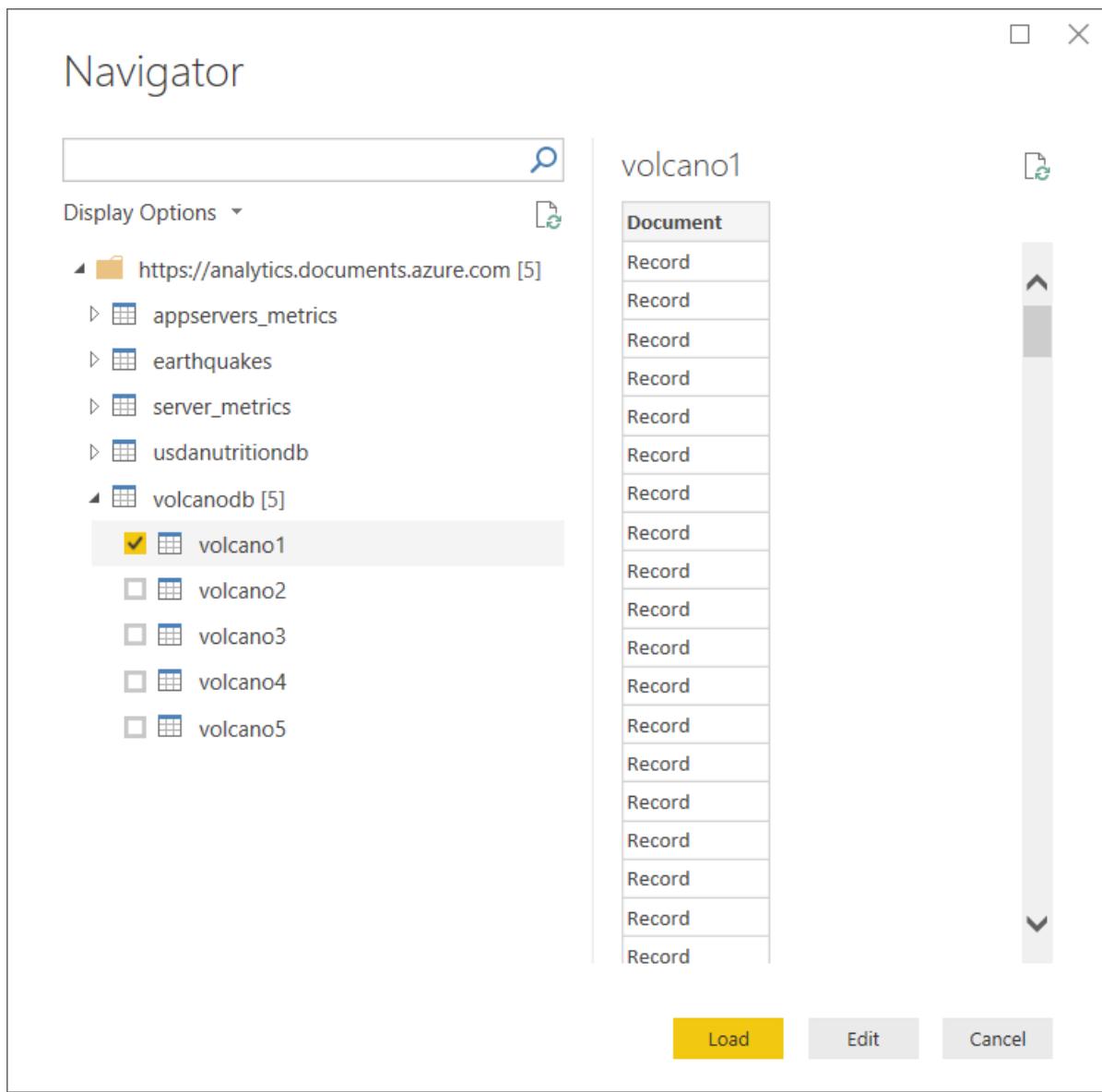


3. Select the **Home** ribbon, then click on **Get Data**. The **Get Data** window should appear.
4. Click on **Azure**, select **Azure Cosmos DB (Beta)**, and then click **Connect**.



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**.
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.

1 COLUMN, 298 ROWS

PREVIEW DOWNLOADED AT 9:15 AM

- 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 Known Eruption. Uncheck the **Use original column name as prefix** box, and then click **OK**.

Search Columns to Expand

- (Select All Columns)
- Volcano Name
- Country
- Region
- Location
- Elevation
- Type
- Status
- Last Known Eruption
- id
- \_rid
- \_ts
- \_self
- \_etag
- attachments

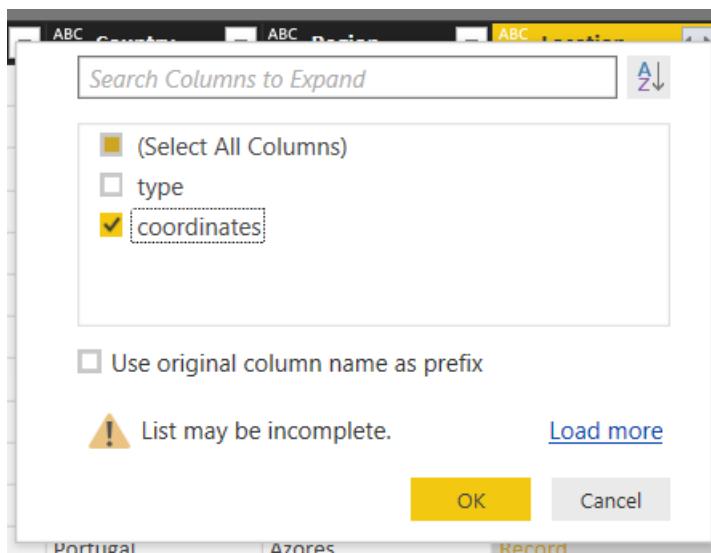
Use original column name as prefix

**⚠ List may be incomplete.** [Load more](#)

**OK** **Cancel**

- 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.

The screenshot shows the Power Query Editor interface with the following details:

- File**, **Home**, **Transform**, **Add Column**, **View**, **Help** menu items.
- Toolbar with icons for Close & Apply, New Query, Data Sources, Parameters, Refresh, Manage, Properties, Advanced Editor, Choose Columns, Remove Columns, Keep Rows, Remove Rows, Sort, Split Column, Group By, Replace Values, Transform, and Merge Queries.
- Queries [1]** pane showing **volcano1**.
- PREVIEW** pane displaying the data with 8 columns and 298 rows.
- QUERY SETTING** pane on the right showing properties like Name (volcano1) and applied steps (Expand...).

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](#).

The 'Custom Column' dialog box contains the following elements:

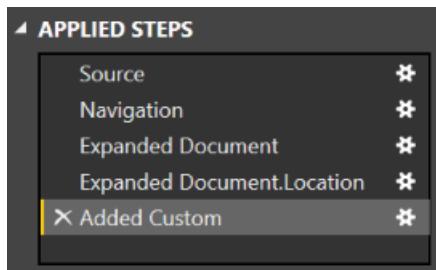
- New column name:** LatLong
- Custom column formula:**

```
=Text.From([coordinates]{1})&","&Text.From([coordinates]{0})
```
- Available columns:**
  - Volcano Name
  - Country
  - Region
  - coordinates
  - Elevation
  - Type
  - Status
- Status:** No syntax errors have been detected.
- Buttons:** OK and Cancel.

10. Now, the center pane shows the new LatLong columns populated with the values.

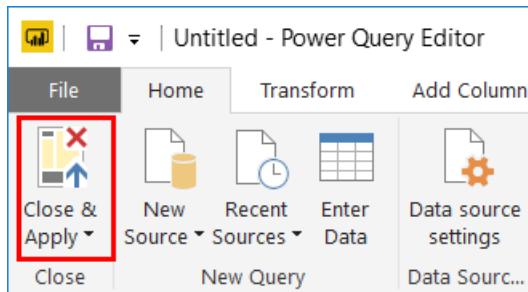
ABC	LatLong
123	28.07,60
	-30.542,-178.561
	42.55,44
	37.67,33.65
	40.75,42.9
	28.58,-17.83
	38.483,14.95
	22.28,95.1
	-25.887,-177.188
	38.65,42.02
	-39.28,175.57
	37.87,-25.78
	-36.321,178.028
	17.683,8.5
	39.25,45.167
	41.55,43.6
	20.92,17.28
	38.404,14.962
	40.275,44.75

If you receive an Error in the new column, make sure that the applied steps under Query Settings match the following figure:



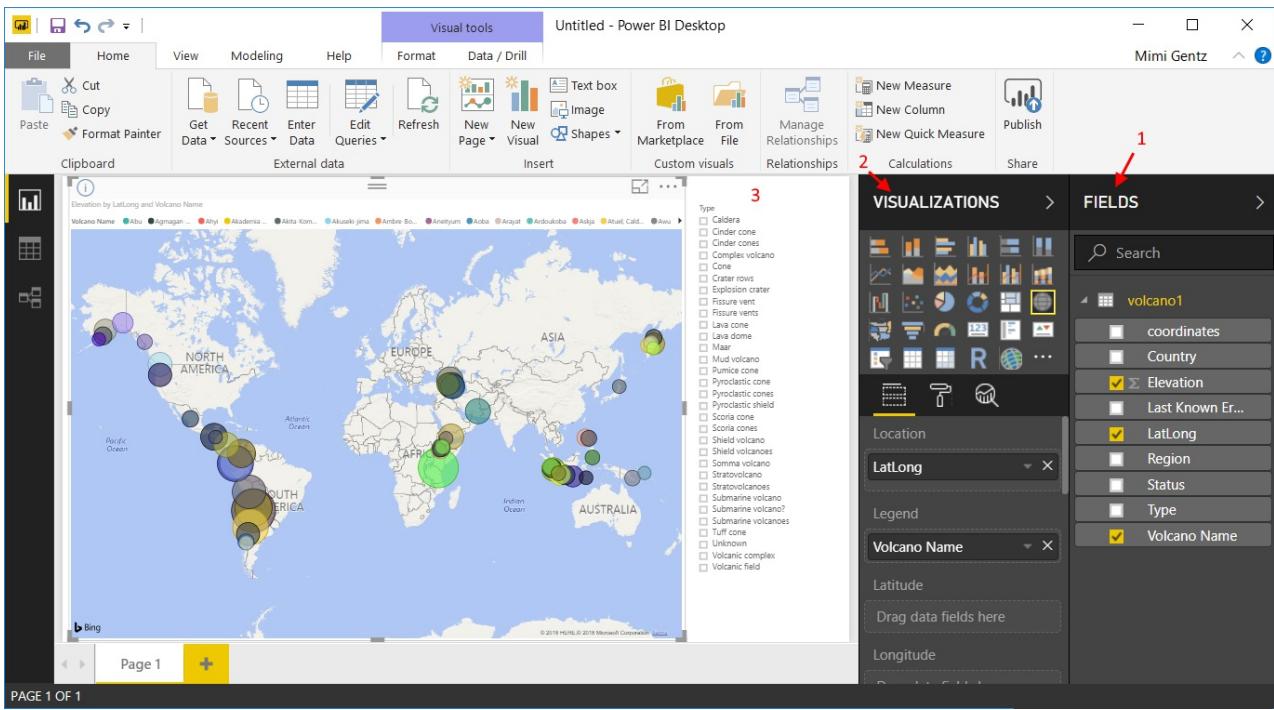
If your steps are different, delete the extra steps and try adding the custom column again.

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.

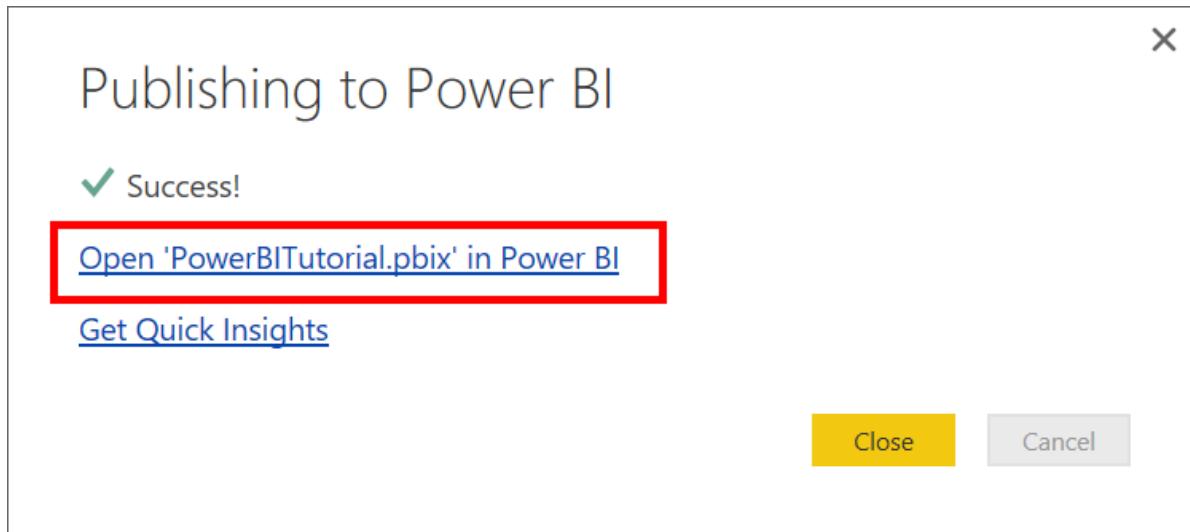
1. For our example, we will create a map view showing the location of each volcano. In the **Visualizations** pane, click on the Map visual type as highlighted in the screenshot above. You should see the Map visual type painted on the **Report** canvas. The **Visualization** pane should also display 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 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.

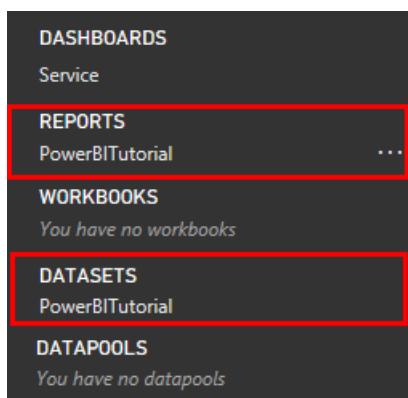
4. Click **Open 'PowerBITutorial.pbix' in Power BI** to see and share your report on PowerBI.com.



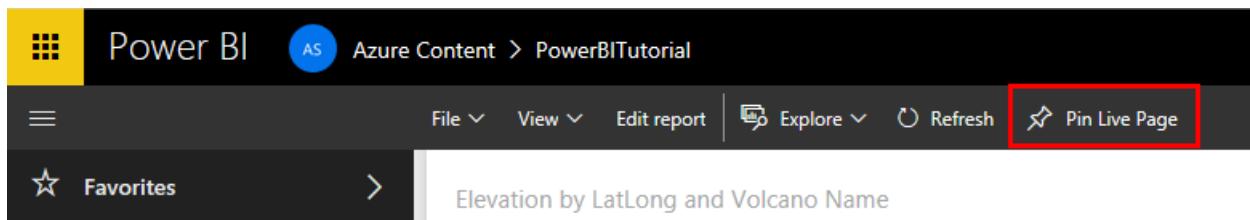
## Create a dashboard in PowerBI.com

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

9/5/2019 • 7 minutes to read • [Edit Online](#)

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)	<a href="#">Power BI Azure Cosmos DB connector with import mode</a>
2. Generating ad-hoc reports with periodic refresh	<a href="#">Power BI Azure Cosmos DB connector with import mode (Scheduled periodic refresh)</a>
3. Reporting on large data sets (< 10 GB)	<a href="#">Power BI Azure Cosmos DB connector with incremental refresh</a>
4. Reporting real time on large data sets	<a href="#">Power BI Azure Analysis Services connector with direct query + Azure Analysis Services (Azure Cosmos DB connector)</a>
5. Reporting on live data with aggregates	<a href="#">Power BI Spark connector with direct query + Azure Databricks + Cosmos DB Spark connector.</a>
6. Reporting on live data with aggregates on large data sets	<a href="#">Power BI Azure Analysis Services connector with direct query + Azure Analysis Services + Azure Databricks + Cosmos DB Spark connector.</a>

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.

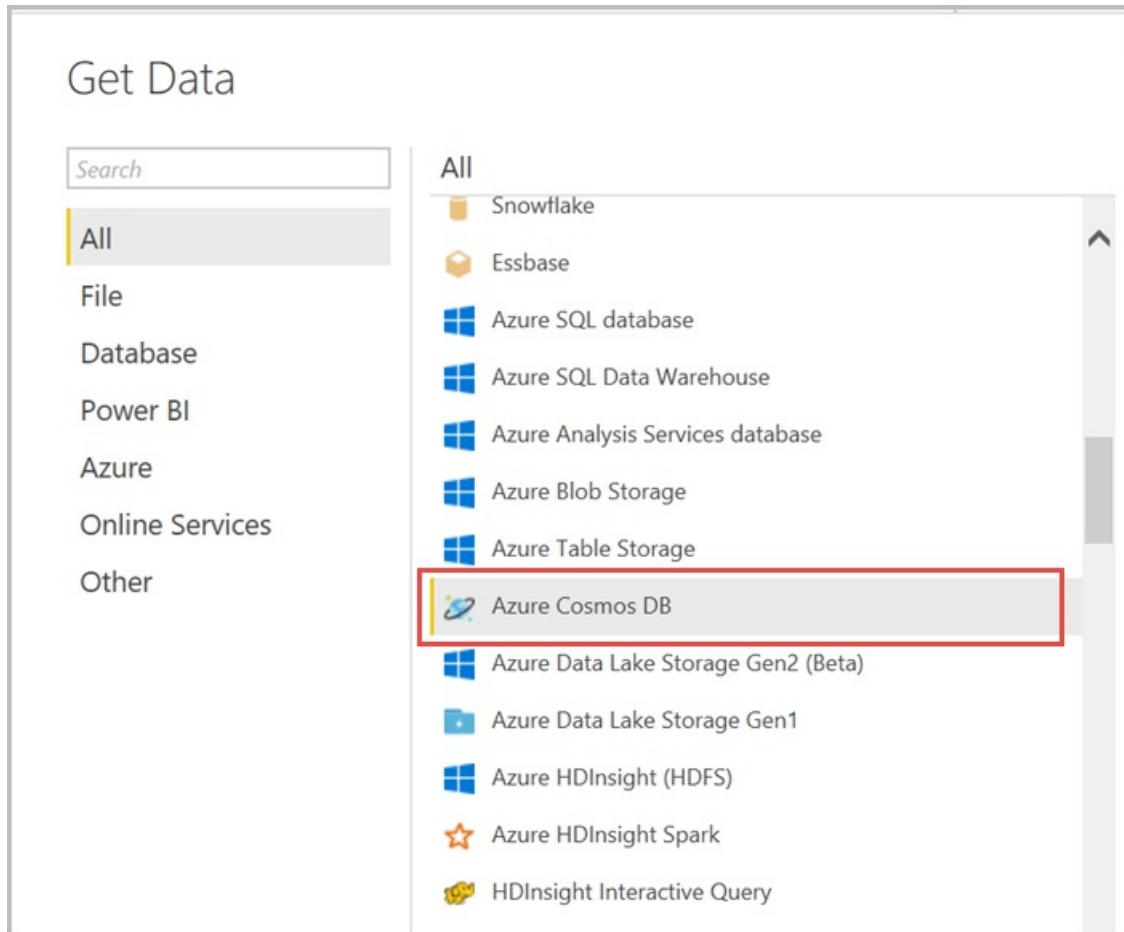
## Power BI incremental refresh

## Ingest weather data into Azure Cosmos DB

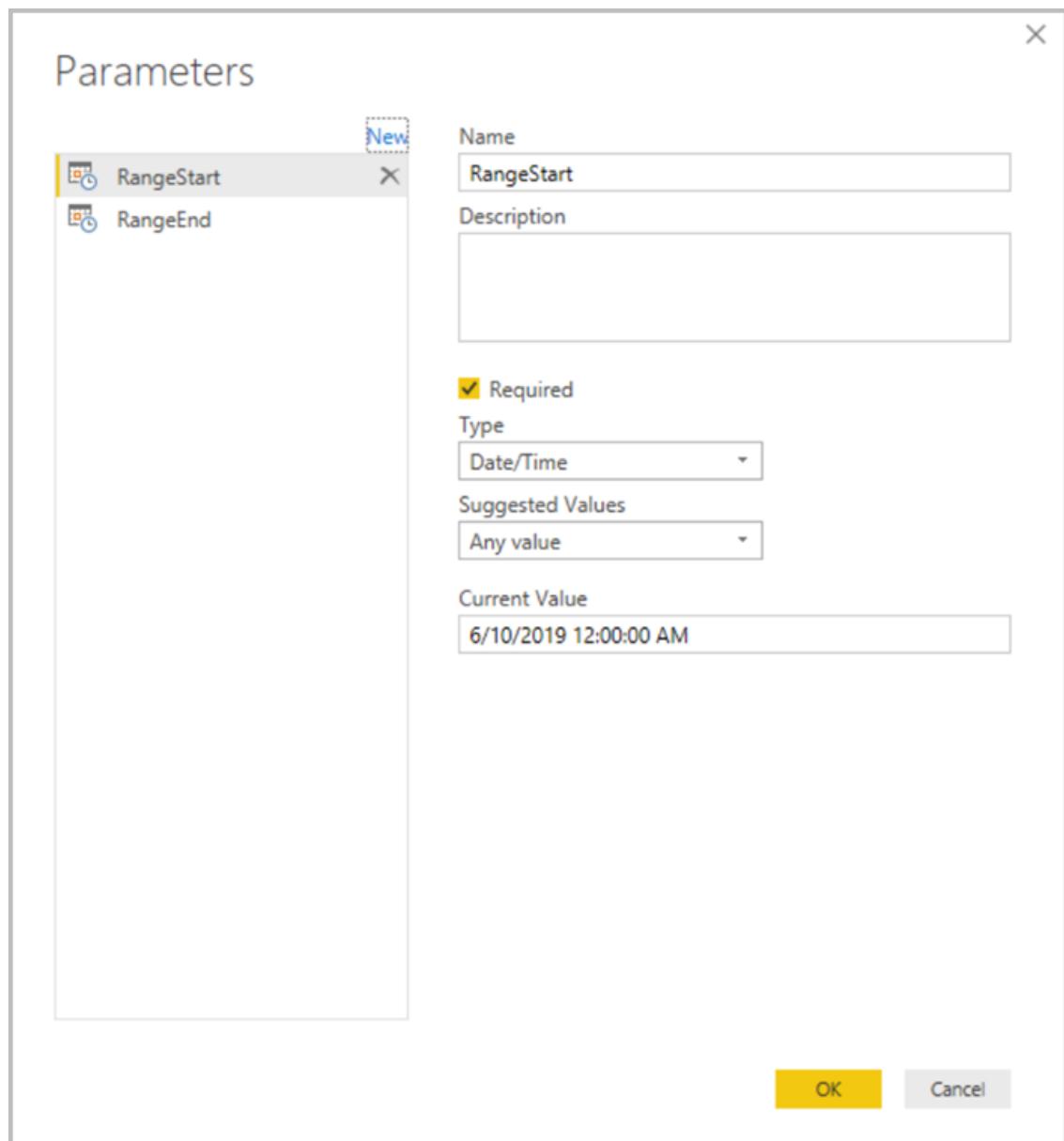
Set up an ingestion pipeline to load [weather data](#) to Azure Cosmos DB. You can set up an [Azure Data Factory \(ADF\)](#) job to periodically load the latest weather data into Azure Cosmos DB using the HTTP Source and Cosmos DB sink.

## 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:

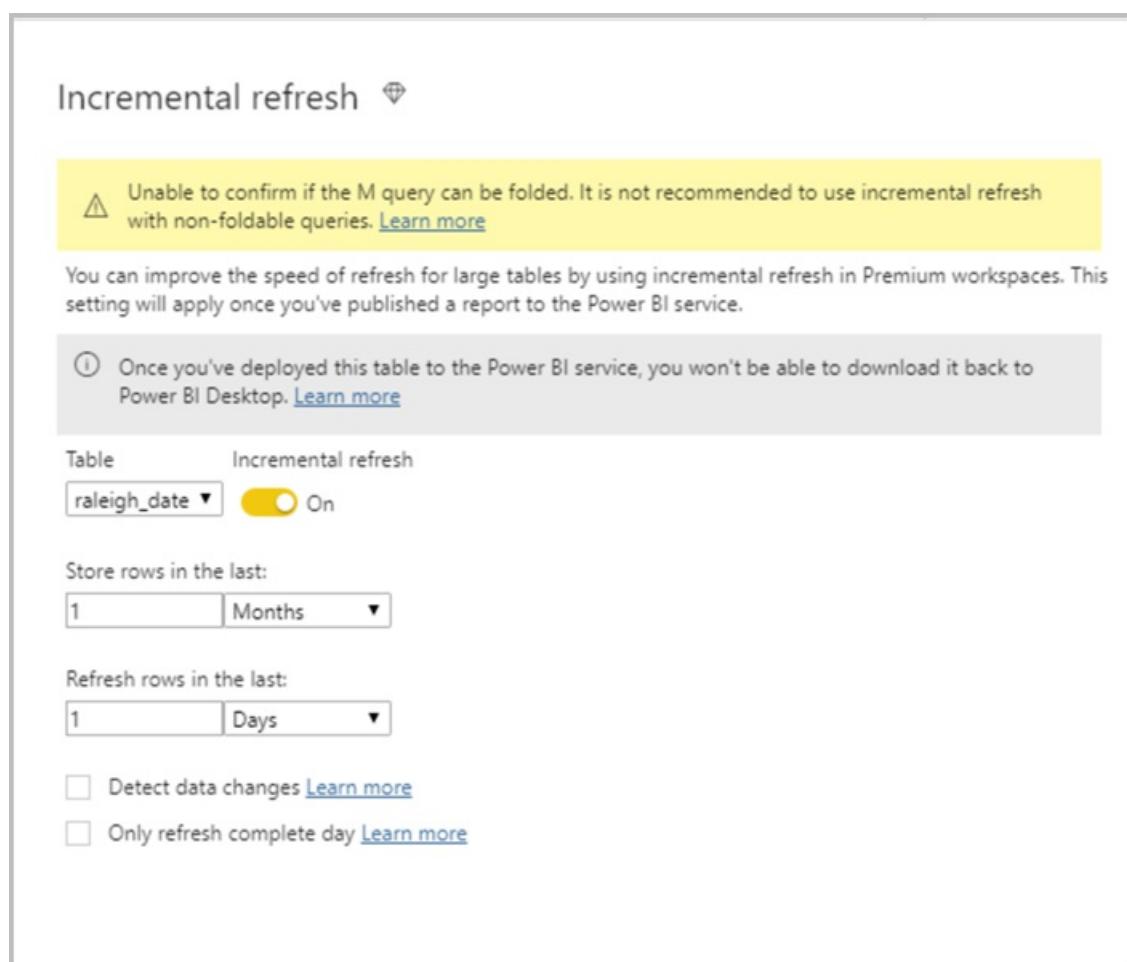
```
#"Filtered Rows" = Table.SelectRows(#"Expanded Document", each [Document.date] >
DateTime.ToDateTime(RangeStart,"yyyy-MM-dd") and [Document.date] < DateTime.ToDateTime(RangeEnd,"yyyy-MM-dd"))
```

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)))

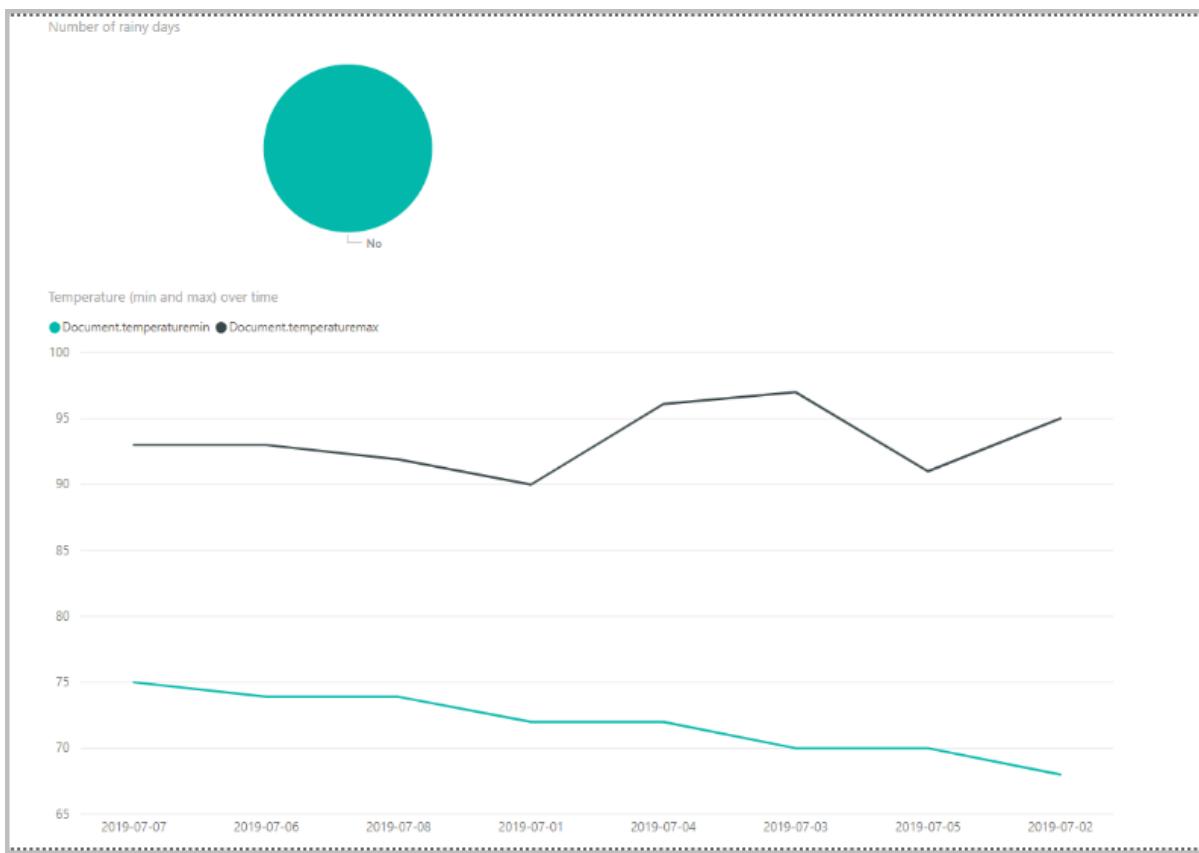
PROPERTY	DATA TYPE	FILTER
Date (for example:- 2019-08-19)	String	[Document.date]> DateTime.ToText(RangeStart,"yyyy-MM-dd") and [Document.date] < DateTime.ToText(RangeEnd,"yyyy-MM-dd")
Date (for example:- 2019-08-11 12:00:00)	String	[Document.date]> DateTime.ToText(RangeStart," yyyy-mm-dd HH:mm:ss") and [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.

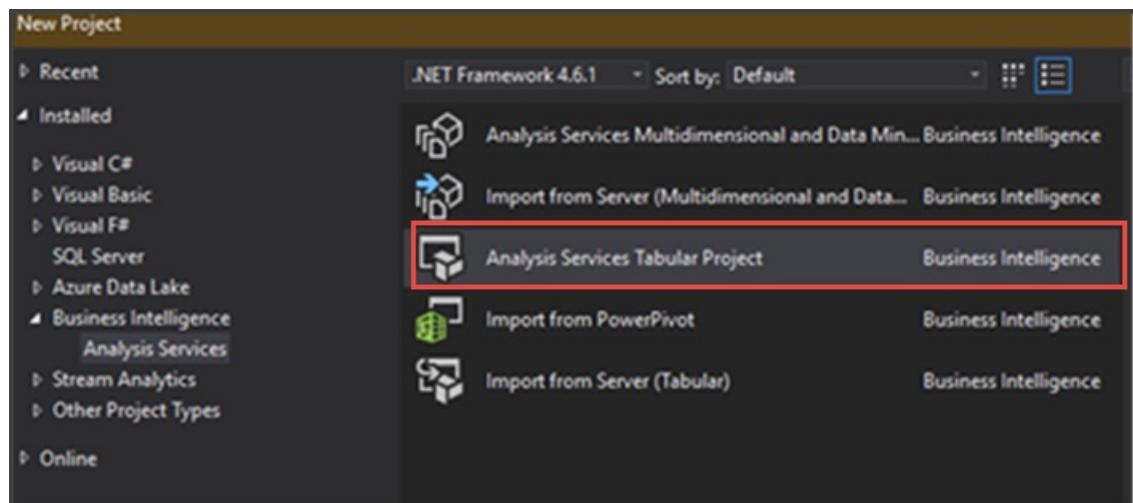
## Power BI Azure Analysis connector + Azure Analysis Services

### Ingest weather data into Azure Cosmos DB

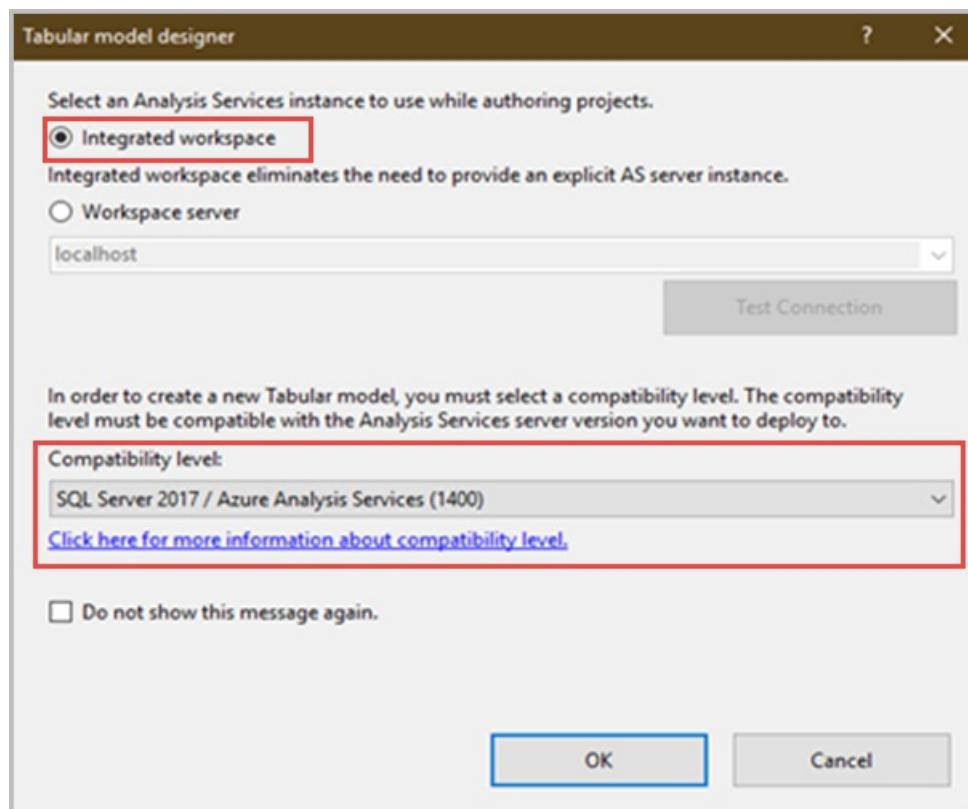
Set up an ingestion pipeline to load [weather data](#) to Azure Cosmos DB. You can set up an Azure Data Factory(ADF) job to periodically load the latest weather data into Azure Cosmos DB using the HTTP Source and Cosmos DB Sink.

### Connect Azure Analysis Services to Azure Cosmos account

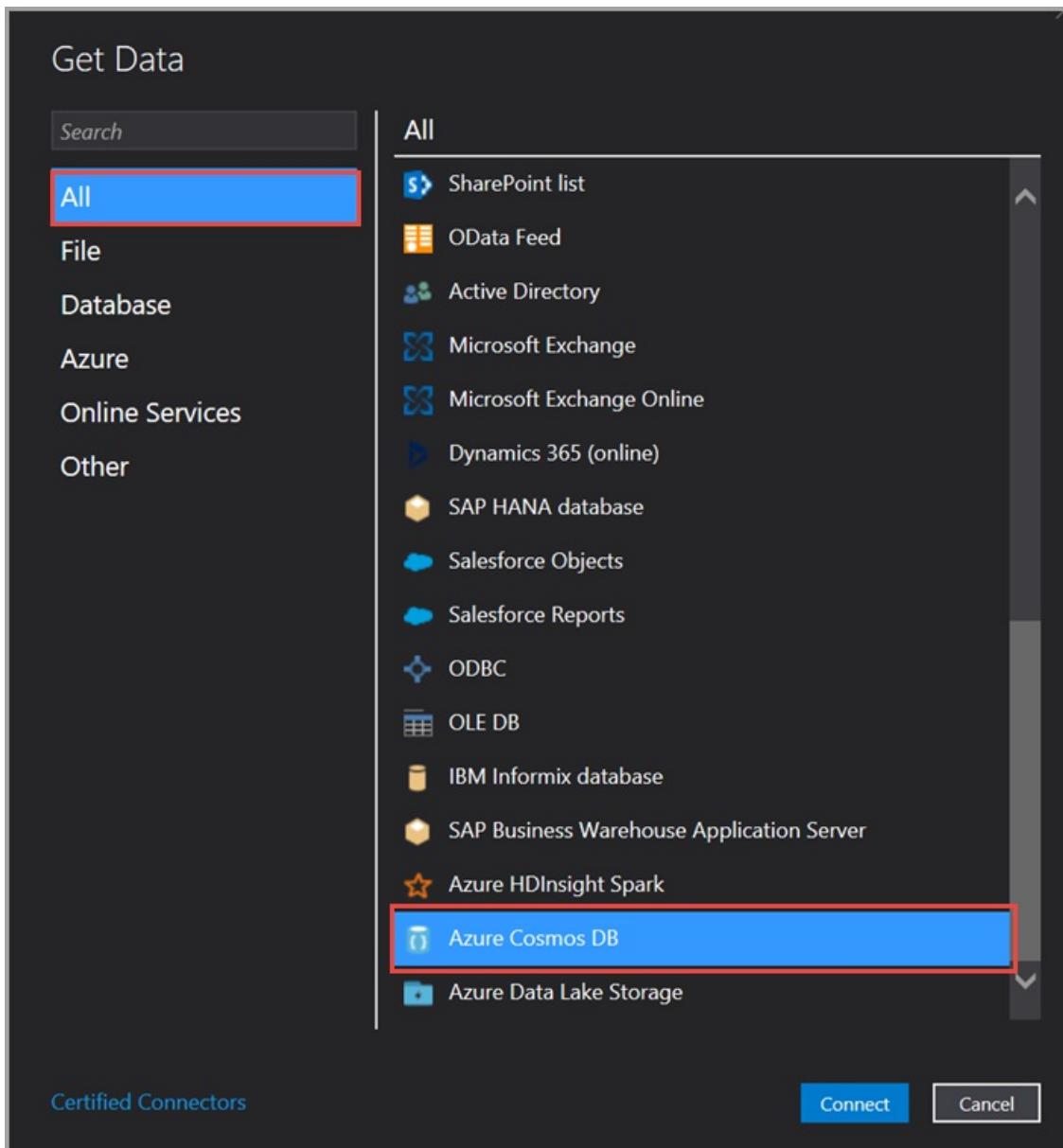
1. **Create a new Azure Analysis Services cluster** - [Create an instance of Azure Analysis services](#) in the same region as the Azure Cosmos account and the Databricks cluster.
2. **Create a new Analysis Services Tabular Project in Visual Studio** - [Install the SQL Server Data Tools \(SSDT\)](#) 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.

	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:

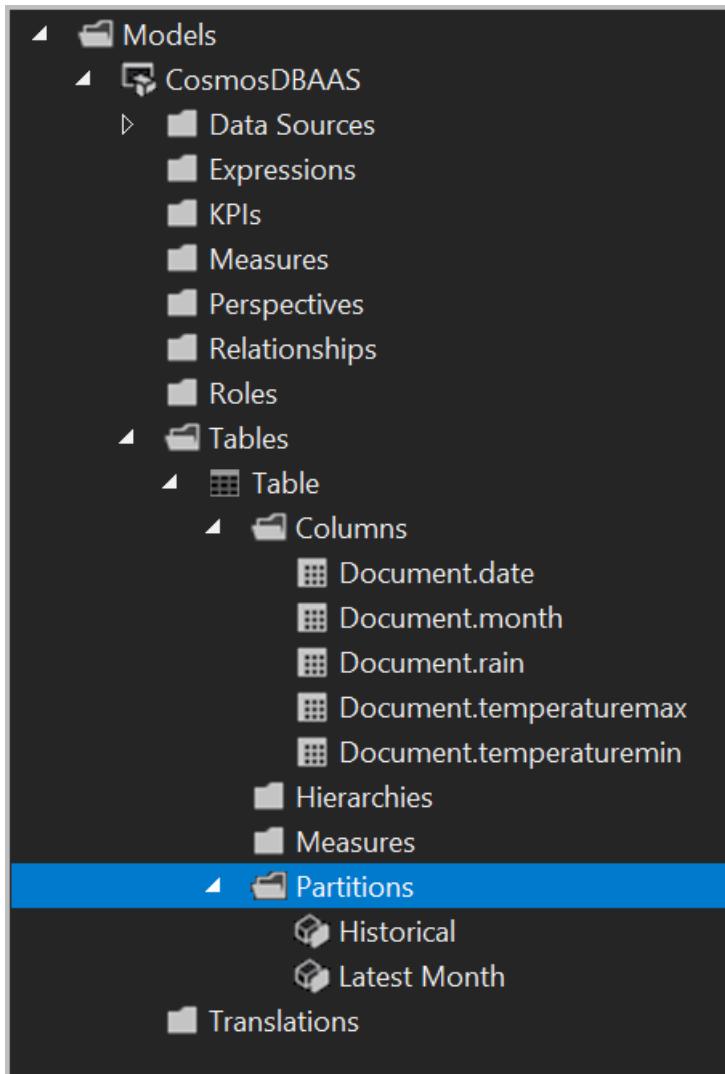
```

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:

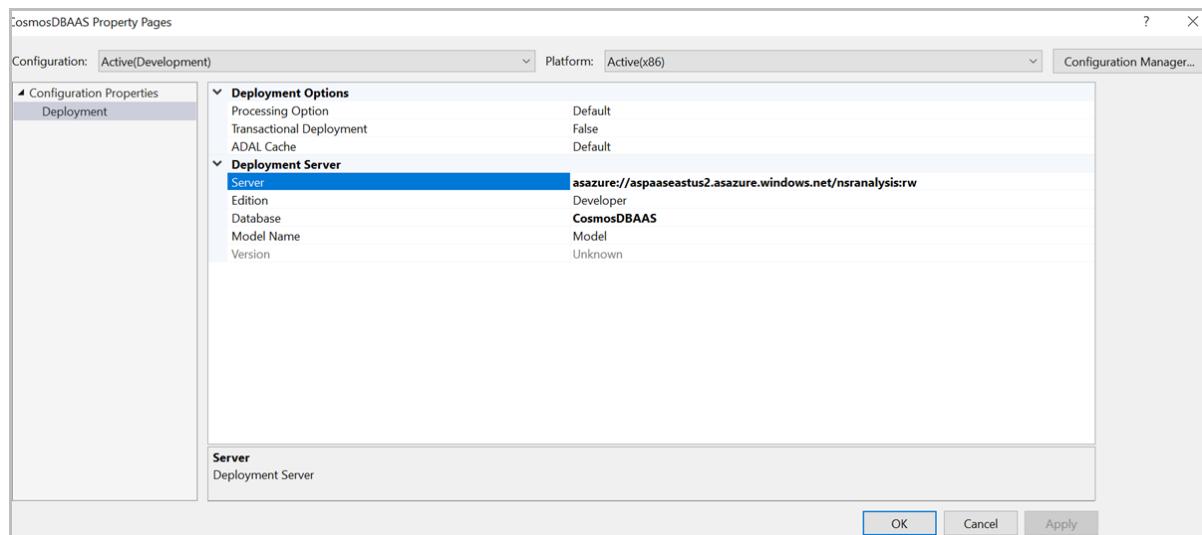
- **Latest Month -**

```
#"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.



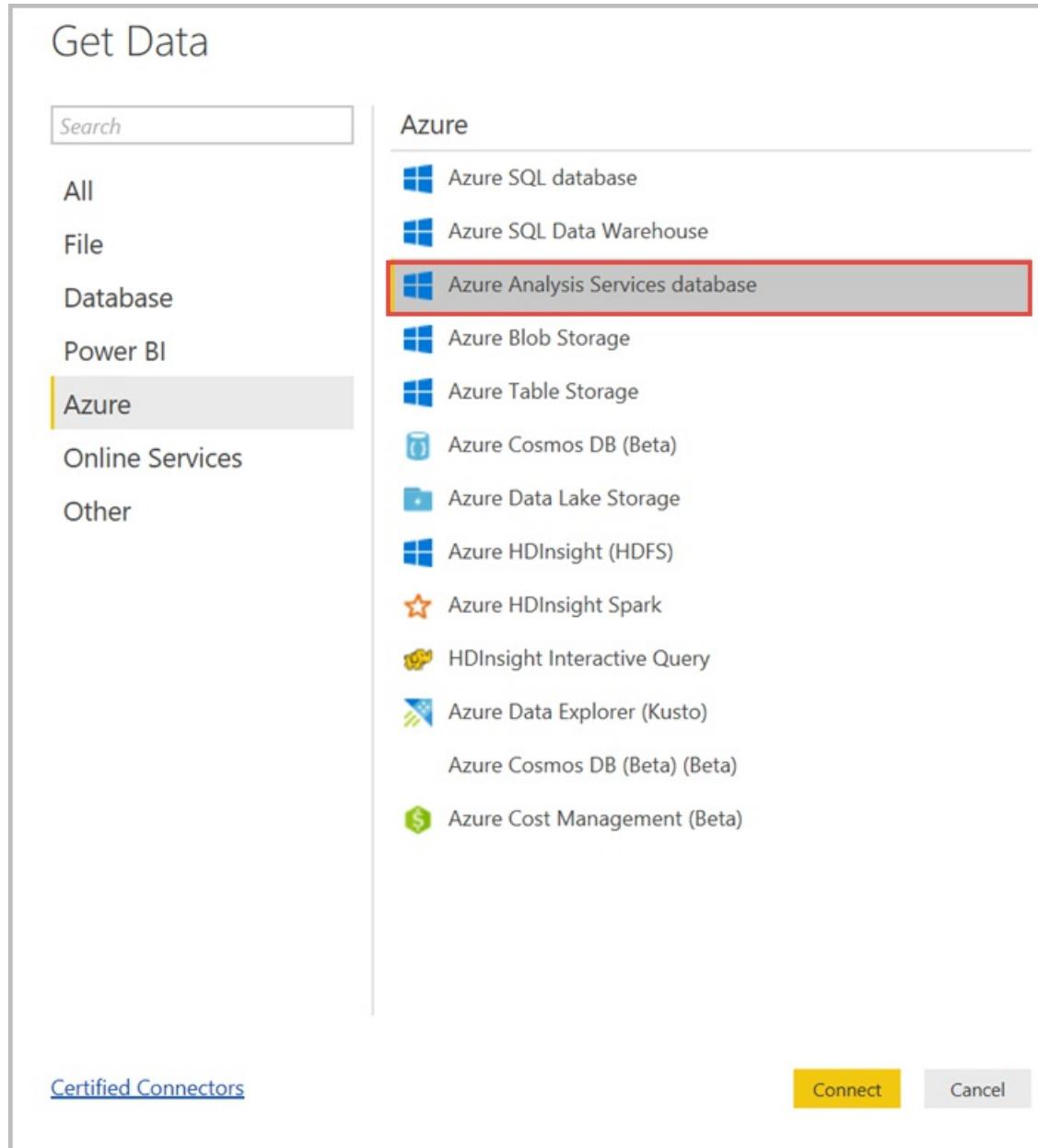
7. **Configure partition refreshes and merges** - Azure Analysis Services allows independent processing of partitions. Since we want the **Latest Month** partition to be constantly updated with the most recent data, set the refresh interval to 5 minutes. It's not required to refresh the data in historical 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

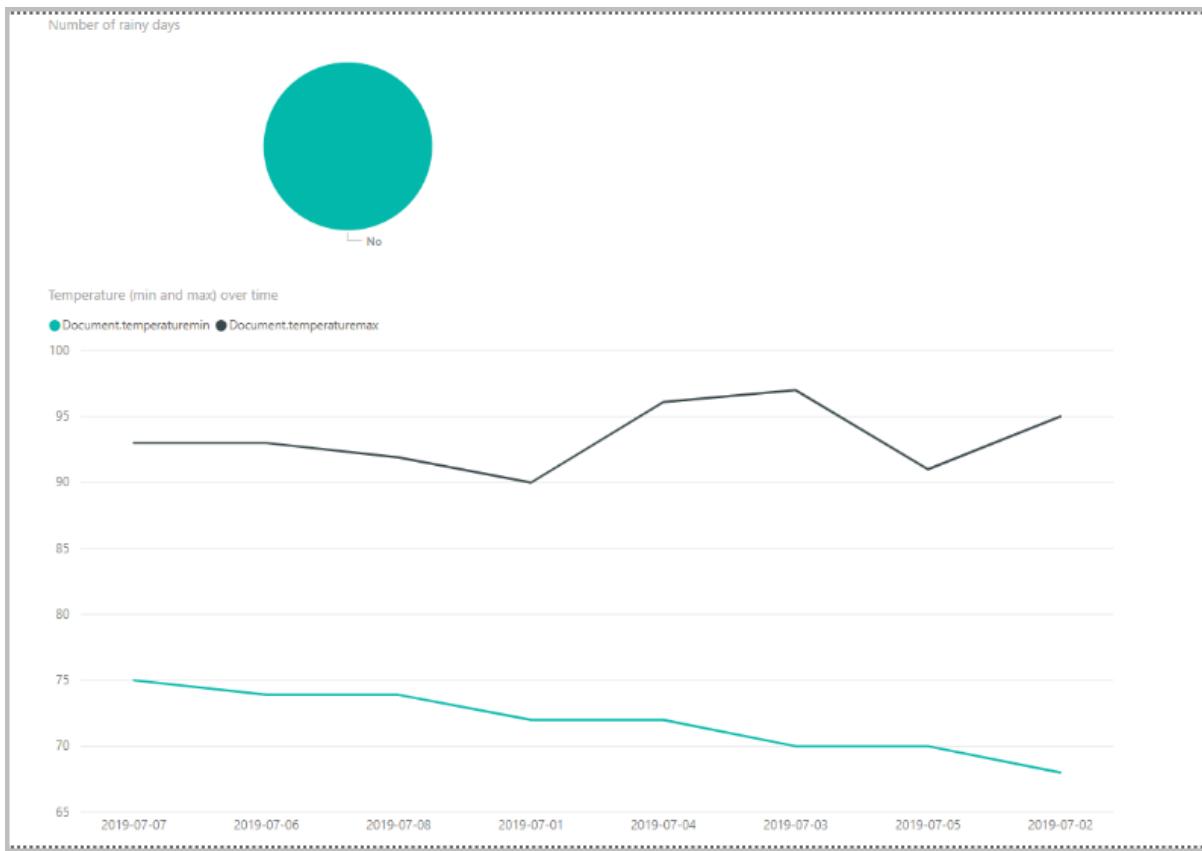
### 1. Connect to the Azure Analysis Server using the Azure Analysis Services database Connector -

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

7/15/2019 • 3 minutes to read • [Edit Online](#)

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.

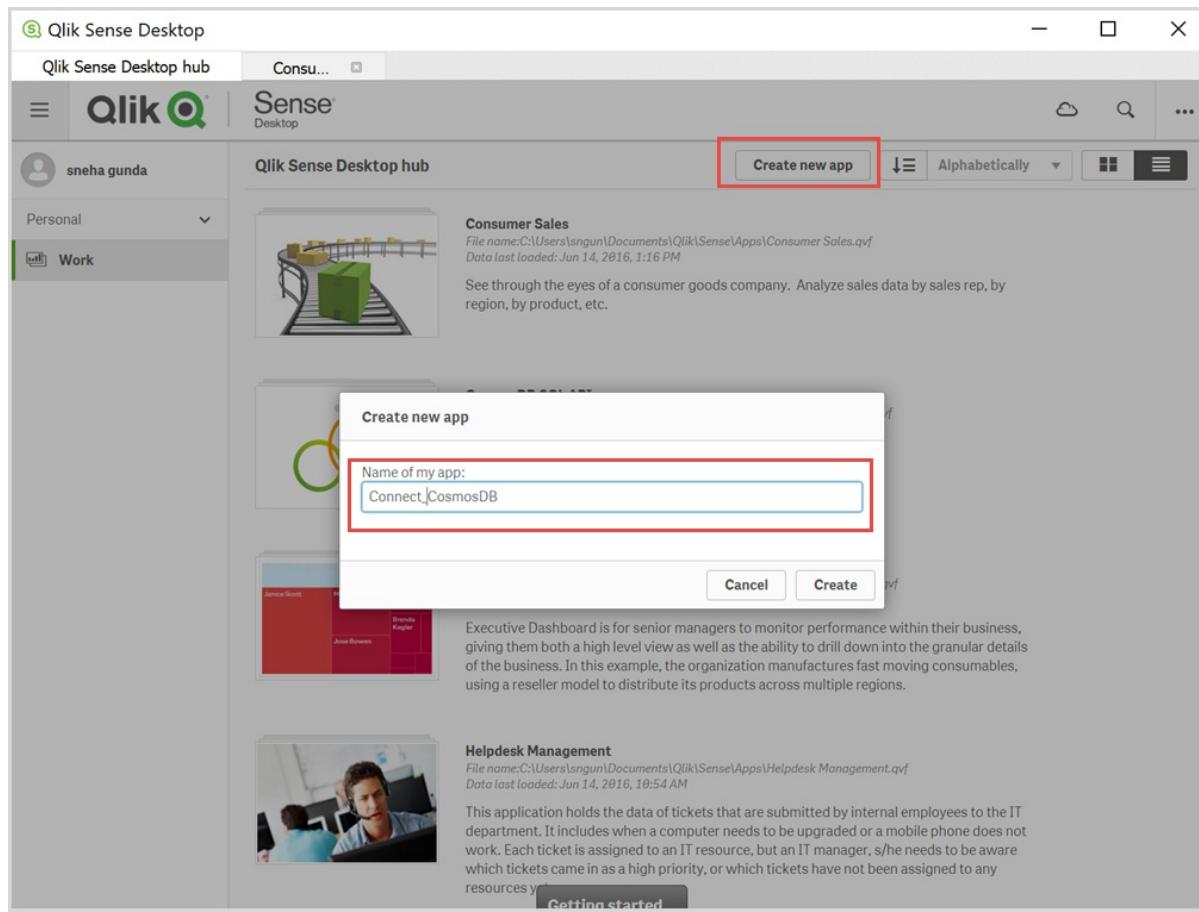
## Prerequisites

Before following the instructions in this article, ensure that you have the following resources ready:

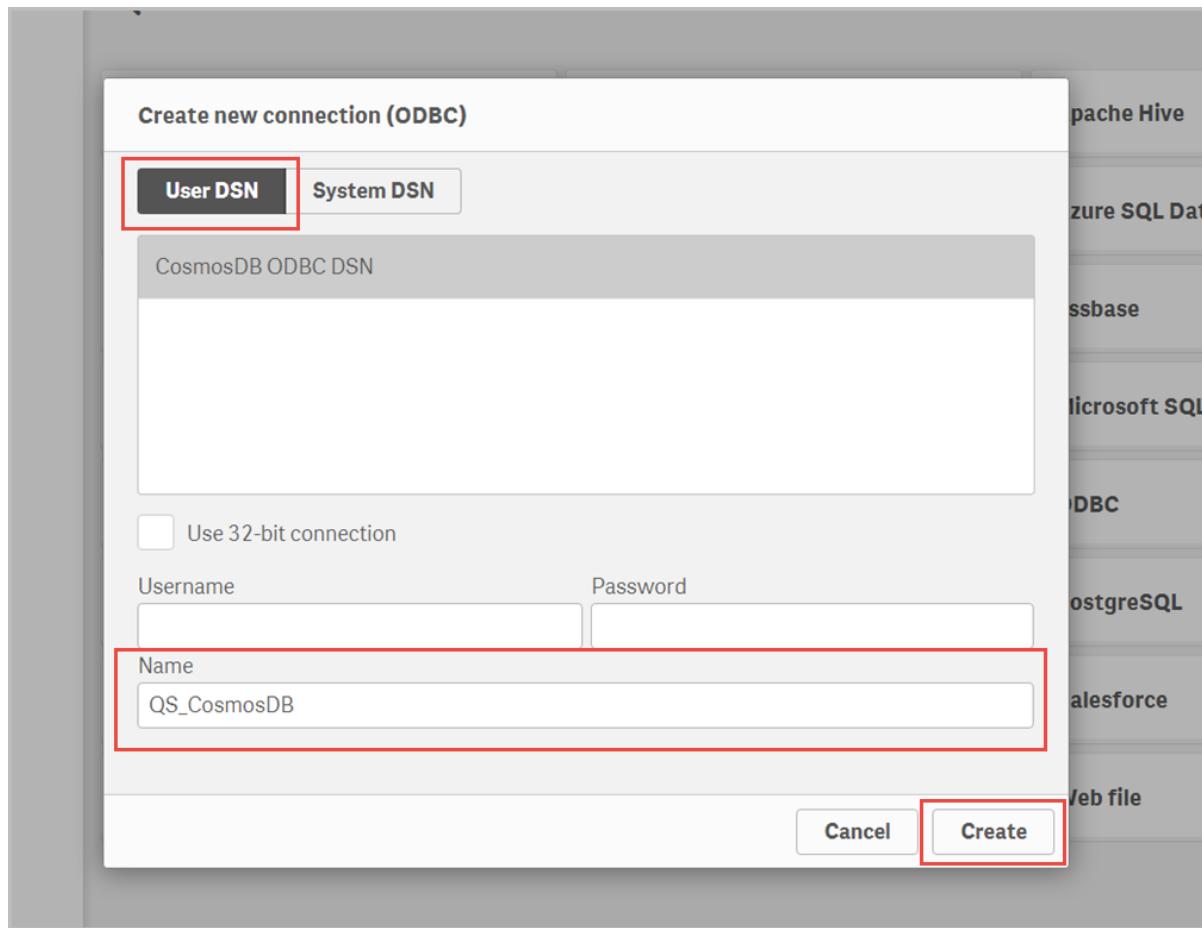
- Download the [Qlik Sense Desktop](#) or set up Qlik Sense in Azure by [Installing the Qlik Sense marketplace item](#).
- Download the [video game data](#), this sample data is in CSV format. You will store this data in a Cosmos DB account and visualize it in Qlik Sense.
- Create an Azure Cosmos DB 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.

EU_S...	Ge...	Global_S...	JP_S...	NA_S...	Name
9.23	Platform	30.01	6.5	11.38	New Super Mario Bros.
8.89	Role-Playing	31.37	10.22	11.27	Pokemon Red/Pokemon Bl
2.26	Puzzle	30.26	4.22	23.2	Tetris
3.58	Platform	40.24	6.81	29.08	Super Mario Bros.
9.2	Misc	29.02	2.93	14.03	Wii Play
12.88	Racing	35.82	3.79	15.85	Mario Kart Wii
29.02	Sports	82.74	3.77	41.49	Wii Sports
0.63	Shooter	28.31	0.28	26.93	Duck Hunt
11.01	Sports	33	3.28	15.75	Wii Sports Resort
7.66	Platform	28.62	4.7	14.59	New Super Mario Bros. Wi
7.57	Racing	23.42	4.13	9.81	Mario Kart DS
6.18	Role-Playing	23.1	7.2	9	Pokemon Gold/Pokemon Si
11	Simulation	24.76	1.93	9.07	Nintendogs
8.59	Sports	22	2.53	9.09	Wii Fit Plus
4.94	Misc	21.82	0.24	14.97	Kinect Adventures!
8.03	Sports	22.72	3.6	8.94	Wii Fit
0.4	Action	20.81	0.41	9.43	Grand Theft Auto: San Andr
9.27	Action	21.4	0.97	7.01	Grand Theft Auto V
9.26	Misc	20.22	4.16	4.75	Brain Age: Train Your Brain i

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](#)

# Use Azure Cosmos DB change feed to visualize real-time data analytics

1/22/2020 • 16 minutes to read • [Edit Online](#)

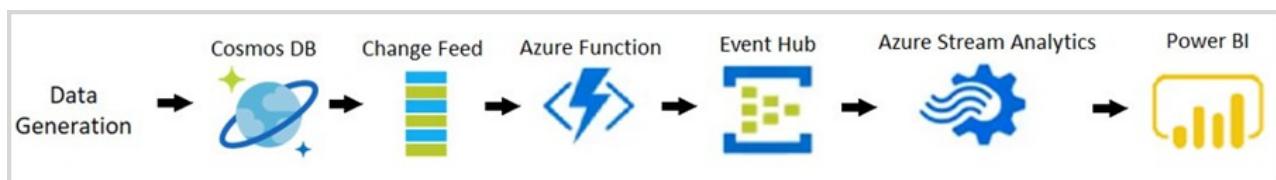
The Azure Cosmos DB change feed is a mechanism to get a continuous and incremental feed of records from an Azure Cosmos container as those records are being created or modified. Change feed support works by listening to container for any changes. It then outputs the sorted list of documents that were 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 as 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.)

```

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://xxxxxxxxxxxxx/" />
8     <add key="authKey" value="xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx==" />
9     <add key="database" value="xxxxxxxxxx" />
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**.

id	/Item
16e02ed0-...	Unisex Sandal...
f2c06b3a-...	Unisex Soc...
7fc7a31b-...	Unisex San...
d775f5b8-...	Women Ea...
26ca2cab-...	Men Black...
1db2449e-...	Men Black...
cd0e4a29-...	Women Ea...
8a785c83-...	Women At...
320da7f5-...	Women Gr...
3d8cc049-...	Women Gr...
228f0543-...	Men Puffy...
2c9efd80-...	Unisex Soc...
207d68bc-...	Women Hi...
93cd77f...	Women At...
08539846-...	Unisex Soc...
39dc1082-...	Women At...
8af02016-f...	Unisex Soc...
77576752-...	Women At...

## 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.

The screenshot shows the Stream Analytics job 'Streamjob1' in the Azure portal. The left sidebar has sections like Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, SETTINGS (Locks), JOB TOPOLOGY (Inputs, Functions, Query, Outputs), and CONFIGURE (Scale). The main area shows a 'Created' card with details: Resource group (change) 'changefeedlab', Status 'Created', Created 'Tuesday, June 19, 2018 4:49:21 PM', Location 'West US', Subscription 'UserVoice', and Subscription ID. On the right, there's a 'Send feedback' link. Below the card, the 'Inputs' section is highlighted with a red box; it shows 0 inputs and an 'Empty' status. The 'Outputs' section shows 0 outputs and an 'Empty' status. To the right, the 'Query' pane displays the following T-SQL code:

```
1 SELECT
2 *
3 INTO
4 [YourOutputAlias]
5 FROM
6 [YourInputAlias]
```

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**.

```

1 SELECT *
2 | *
3 INTO [YourOutputAlias]
4 FROM [YourInputAlias]

```

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.

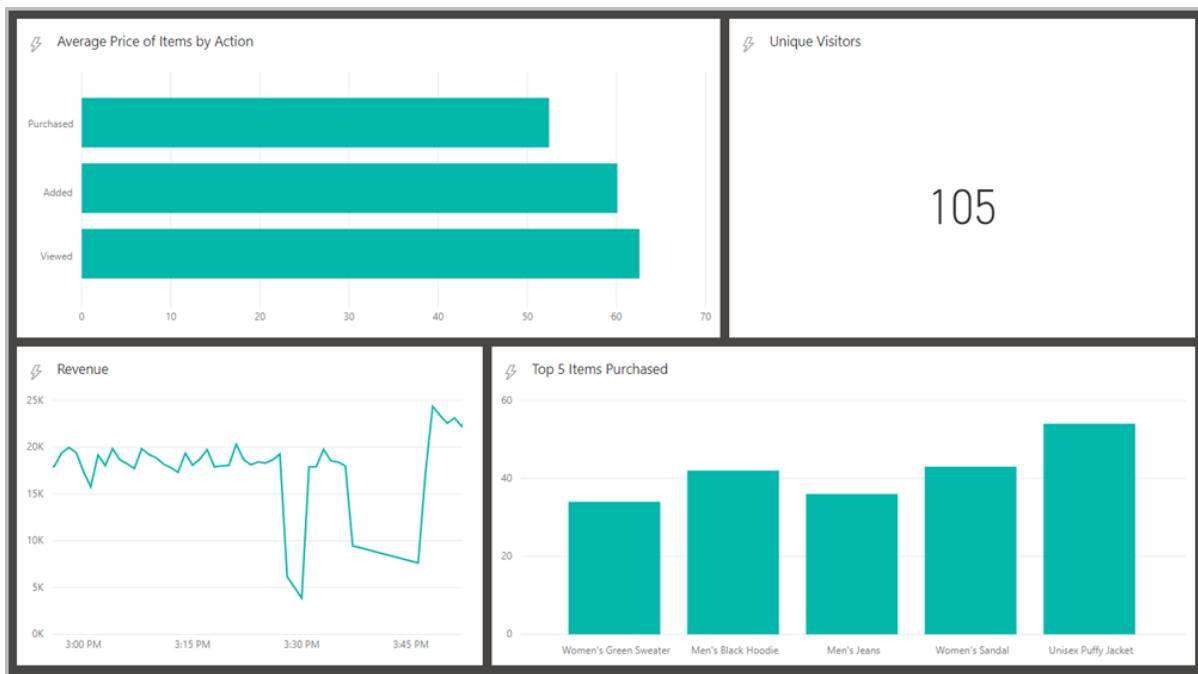
The REVENUE query calculates revenue by summing up the prices of all items purchased each minute. This metric can help e-commerce companies evaluate its financial performance and also understand what times of day contribute to most revenue. This can impact the overall company strategy, marketing in particular.

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.

**topItems**

**Documents**

- Scale & Settings **(highlighted)**
- Stored Procedures
- User Defined Functions
- Triggers

**Storage capacity**  
**Unlimited**

**Throughput (1,000 - 50,000 RU/s)**  
10000 **- +**

**Estimated spend (USD): \$0.80 hourly / \$19.20 daily.**  
[Contact support](#) for more than 50,000 RU/s

**Settings**

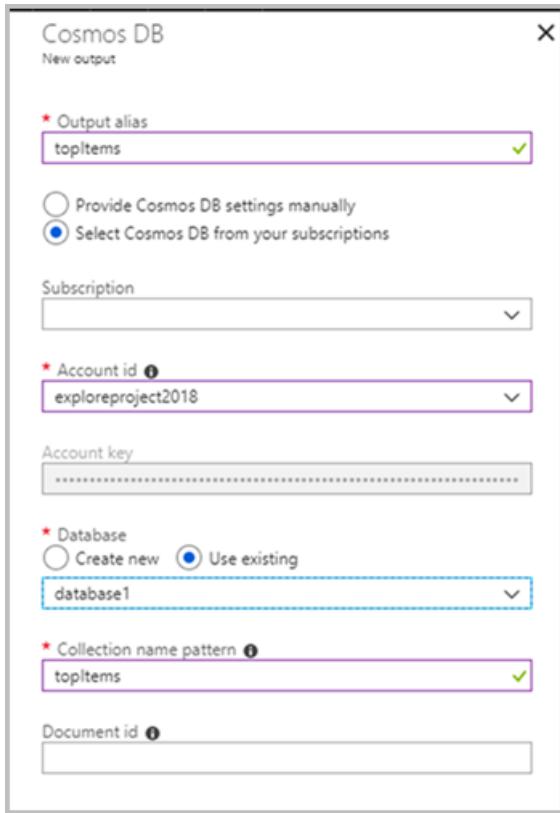
**Time to Live**

Off	On (no default)	<b>On</b>
30	second(s)	

**Partition key**  
**/Item**

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 Web Apps using an Azure Resource Manager Template

1/30/2020 • 6 minutes to read • [Edit Online](#)

This tutorial shows you how to use an Azure Resource Manager template to deploy and integrate [Microsoft Azure Cosmos DB](#), an [Azure App Service](#) web app, and a sample web application.

Using Azure Resource Manager templates, you can easily automate the deployment and configuration of your Azure resources. This tutorial shows how to deploy a web application and automatically configure Azure Cosmos DB account connection information.

After completing this tutorial, you will be able to answer the following questions:

- How can I use an Azure Resource Manager template to deploy and integrate an Azure Cosmos DB account and a web app in Azure App Service?
- How can I use an Azure Resource Manager template to deploy and integrate an Azure Cosmos DB account, a web app in App Service Web Apps, and a Webdeploy application?

## 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.

Before following the instructions in this tutorial, ensure that you have the an Azure subscription. Azure is a subscription-based platform. For more information about obtaining a subscription, see [Purchase Options](#), [Member Offers](#), or [Free Trial](#).

## Step 1: Download the template files

Let's start by downloading the template files that this tutorial requires.

1. Download the [Create an Azure Cosmos DB account, Web Apps, and deploy a demo application sample](#) template to a local folder (for example, C:\Azure Cosmos DBTemplates). This template deploys an Azure Cosmos DB account, an App Service web app, and a web application. It also automatically configures the web application to connect to the Azure Cosmos DB account.
2. Download the [Create an Azure Cosmos DB account and Web Apps sample](#) template to a local folder (for example, C:\Azure Cosmos DBTemplates). This template deploys an Azure Cosmos DB account, an App Service web app, and modifies the site's application settings to easily surface Azure Cosmos DB connection information, but does not include a web application.

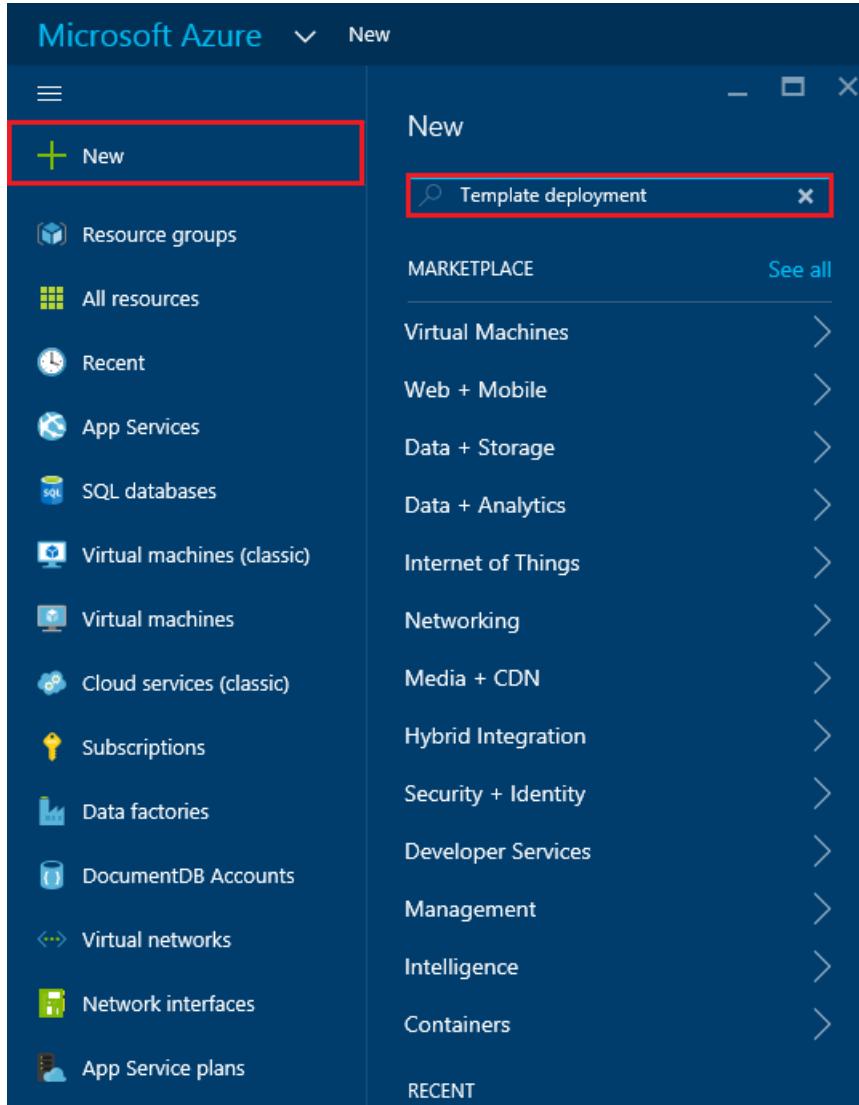
## Step 2: Deploy the Azure Cosmos DB account, App Service web app, and demo application sample

Now let's deploy your first template.

**TIP**

The template does not validate that the web app name and Azure Cosmos DB account name entered in the following template are a) valid and b) available. It is highly recommended that you verify the availability of the names you plan to supply prior to submitting the deployment.

1. Login to the [Azure Portal](#), click New and search for "Template deployment".



2. Select the Template deployment item and click **Create**

3. Click **Edit template**, paste the contents of the DocDBWebsiteTodo.json template file, and click **Save**.

```

1 {
2   "$schema": "http://schema.management.azure.com/schemas/2014-04-01-preview/deploymentTemplate.json#",
3   "contentVersion": "1.0.0.0",
4   "parameters": {
5     "siteName": {
6       "type": "string"
7     },
8     "hostingPlanName": {
9       "type": "string"
10    },
11    "location": {
12      "type": "string",
13      "allowedValues": [
14        "Central US",
15        "East Asia",
16        "East US",
17        "Japan East",
18        "Japan West",
19        "North Europe",
20        "South Central US",
21        "Southeast Asia",
22        "West Europe",
23        "West US"
24      ]
25    },
26    "sku": {
27      "type": "string",
28      "allowedValues": [
29        "Free",
30        "Shared",
31        "Basic",
32        "Standard"
33      ],
34      "defaultValue": "Free"
35    }
36  },
37  "variables": {},
38  "resources": [
39    {
40      "type": "Microsoft.Web/sites",
41      "name": "[parameters('siteName')]",
42      "apiVersion": "2015-08-01",
43      "dependsOn": [
44        "[resourceId('Microsoft.Web/hostingPlans', parameters('hostingPlanName'))]"
45      ],
46      "properties": {
47        "name": "[parameters('siteName')]",
48        "hostingPlan": "[resourceId('Microsoft.Web/hostingPlans', parameters('hostingPlanName'))]",
49        "location": "[parameters('location')]",
50        "scmSiteName": "[concat(parameters('siteName'), '-scm')]"
51      }
52    }
53  ]
54 }

```

4. Click **Edit parameters**, provide values for each of the mandatory parameters, and click **OK**. The parameters are as follows:

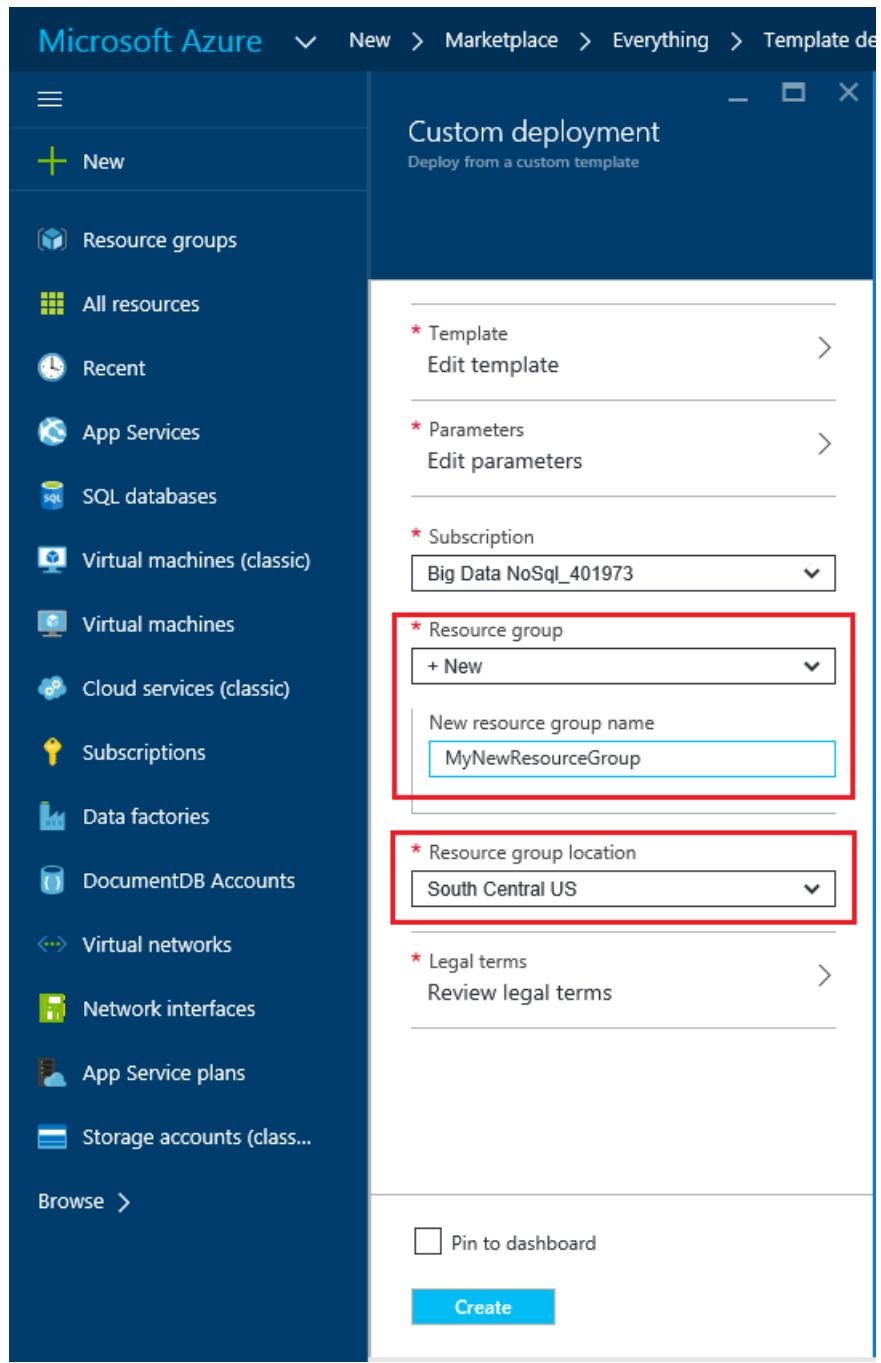
- SITENAME: Specifies the App Service web app name and is used to construct the URL that you use to access the web app (for example, if you specify "mydemodocdbwebapp", then the URL by which you access the web app is `mydemodocdbwebapp.azurewebsites.net` ).
- HOSTINGPLANNAME: Specifies the name of App Service hosting plan to create.
- LOCATION: Specifies the Azure location in which to create the Azure Cosmos DB and web app resources.

d. DATABASEACCOUNTNAME: Specifies the name of the Azure Cosmos DB account to create.

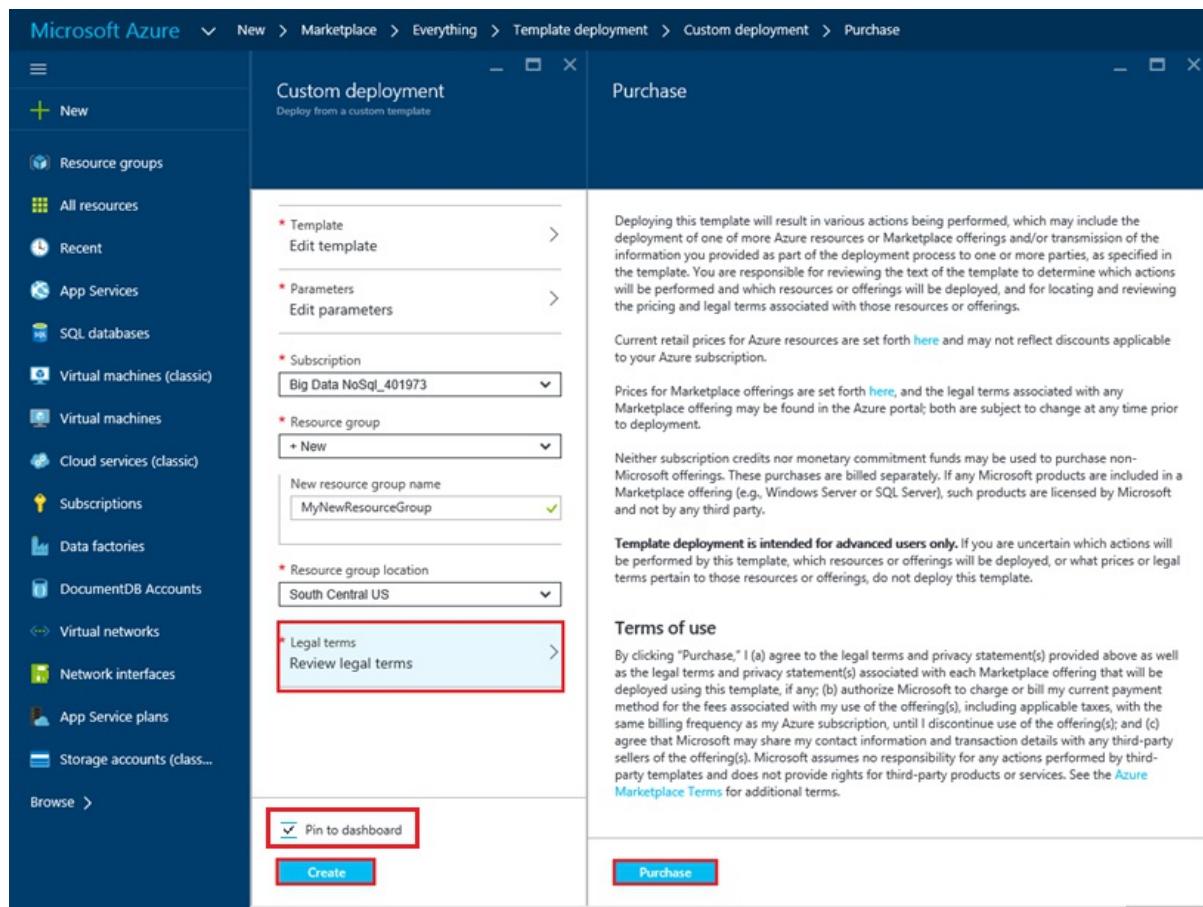
The screenshot shows the Microsoft Azure portal interface with the following details:

- Left Sidebar:** Shows various Azure service categories like Resource groups, All resources, App Services, SQL databases, etc.
- Central Panel - Custom deployment:**
  - Template:** Edit template
  - Parameters:** Edit parameters (highlighted with a red box)
  - Subscription:** Big Data NoSql\_401973
  - Resource group:** + New (highlighted with a red box)
  - New resource group name:** (empty input field)
  - Resource group location:** South Central US
  - Legal terms:** Review legal terms
- Right Panel - Parameters:**
  - SITENAME (string):** mydemotodoappsite
  - HOSTINGPLANNENAME (string):** mydemotodoplans
  - LOCATION (string):** Central US
  - DATABASEACCOUNTNAME (string):** mydemotododocdb
- Bottom Buttons:** Pin to dashboard (checkbox), Create, OK (highlighted with a red box).

5. Choose an existing Resource group or provide a name to make a new resource group, and choose a location for the resource group.



6. Click **Review legal terms**, **Purchase**, and then click **Create** to begin the deployment. Select **Pin to dashboard** so the resulting deployment is easily visible on your Azure portal home page.



7. When the deployment finishes, the Resource group pane opens.

8. To use the application, navigate to the web app URL (in the example above, the URL would be <http://mydemodocdbwebapp.azurewebsites.net>). You'll see the following web application:

## Index

[Create New](#)

Name	Description	Completed

© 2015 - My ASP.NET Application

9. Go ahead and create a couple of tasks in the web app and then return to the Resource group pane in the Azure portal. Click the Azure Cosmos DB account resource in the Resources list and then click **Data Explorer**.
10. Run the default query, "SELECT \* FROM c" and inspect the results. Notice that the query has retrieved the JSON representation of the todo items you created in step 7 above. Feel free to experiment with queries; for example, try running SELECT \* FROM c WHERE c.isComplete = true to return all todo items that have been marked as complete.
11. Feel free to explore the Azure Cosmos DB portal experience or modify the sample Todo application. When you're ready, let's deploy another template.

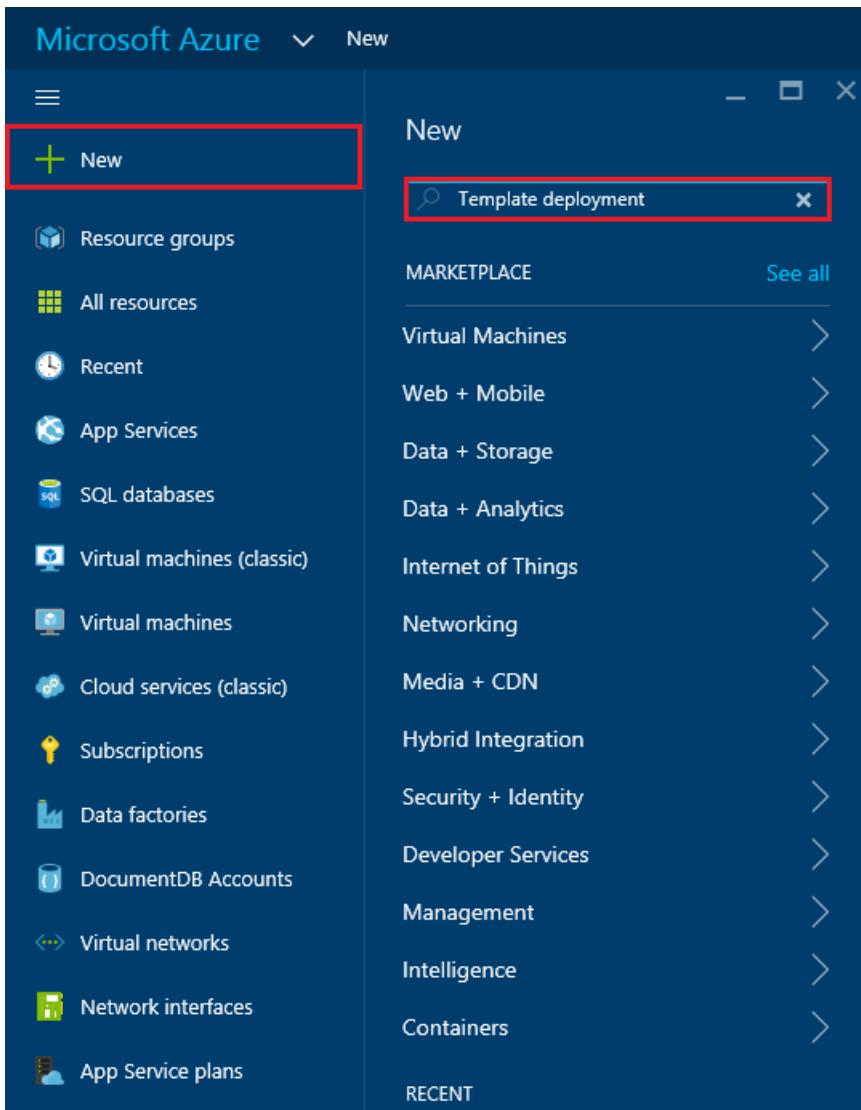
## Step 3: Deploy the Document account and web app sample

Now let's deploy your second template. This template is useful to show how you can inject Azure Cosmos DB connection information such as account endpoint and master key into a web app as application settings or as a custom connection string. For example, perhaps you have your own web application that you would like to deploy with an Azure Cosmos DB account and have the connection information automatically populated during deployment.

**TIP**

The template does not validate that the web app name and Azure Cosmos DB account name entered below are a) valid and b) available. It is highly recommended that you verify the availability of the names you plan to supply prior to submitting the deployment.

1. In the [Azure Portal](#), click New and search for "Template deployment".



2. Select the Template deployment item and click **Create**

NAME	PUBLISHER	CATEGORY
<b>Template deployment</b>	Microsoft	Virtual Machines
VM-Series (BYOL) Solution Template	Palo Alto Networks, Inc.	Virtual Machines
Web App	Microsoft	Web + Mobile
Web App + MySQL	Microsoft	Web + Mobile
Advanced Supply Chain Software	I.B.I.S., Inc. A Sonata Software Company	Virtual Machines
Parse Server on managed Azure services	Microsoft	Web + Mobile
Web App + SQL	Microsoft	Web + Mobile
Website + SQL + Redis Cache	Microsoft	
PrestaShop 1.6 (Starter offer)	PrestaShop SA	Virtual Machines
Food AXcelerator	Edgewater Fullscope	Virtual Machines

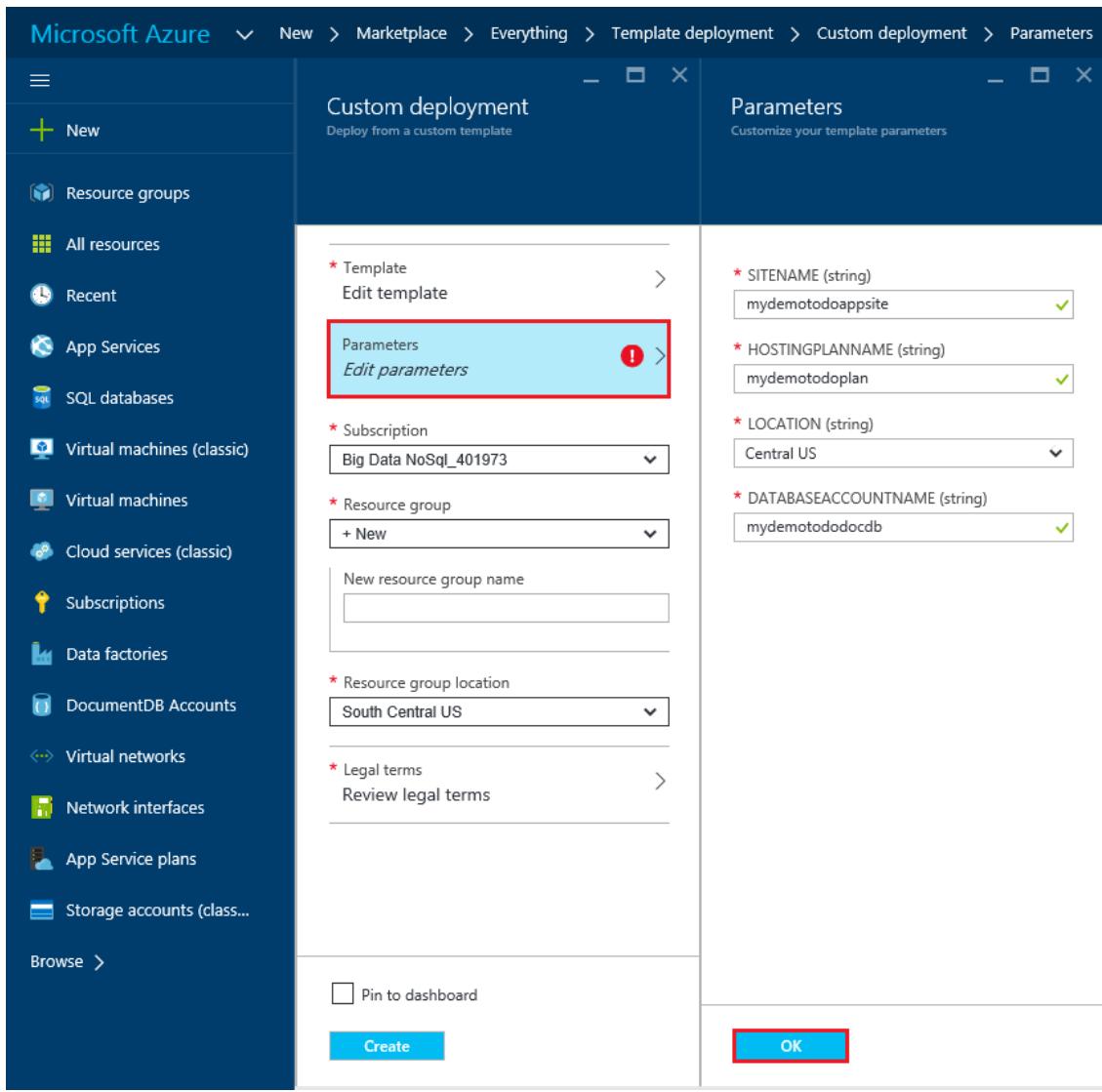
3. Click **Edit template**, paste the contents of the DocDBWebSite.json template file, and click **Save**.

```

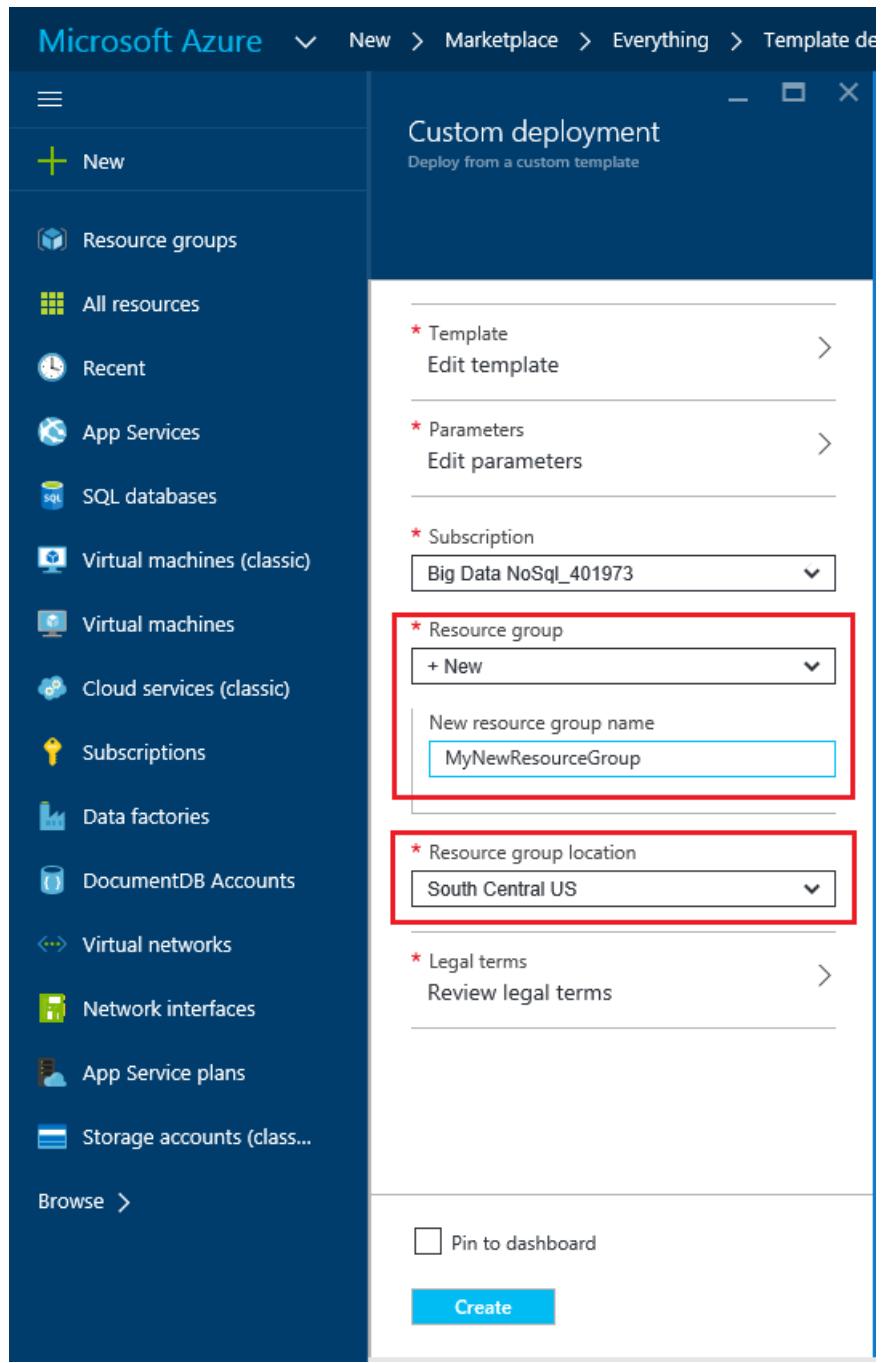
1 {
2   "$schema": "http://schema.management.azure.com/schemas/2014-04-01-preview/deploymentTemplate.json#",
3   "contentVersion": "1.0.0.0",
4   "parameters": {
5     "siteName": {
6       "type": "string"
7     },
8     "hostingPlanName": {
9       "type": "string"
10    },
11    "location": {
12      "type": "string",
13      "allowedValues": [
14        "Central US",
15        "East Asia",
16        "East US",
17        "Japan East",
18        "Japan West",
19        "North Europe",
20        "South Central US",
21        "Southeast Asia",
22        "West Europe",
23        "West US"
24      ]
25    }
26  },
27  "sku": {
28    "type": "string",
29    "allowedValues": [
30      "Free",
31      "Shared",
32      "Basic",
33      "Standard"
34    ],
35    "defaultValue": "Free"
36  },
37  "workerSize": {
38    "type": "string",
39    "allowedValues": [
40      "0",
41      "1",
42      "2"
43    ],
44    "defaultValue": "0"
45  }
46}

```

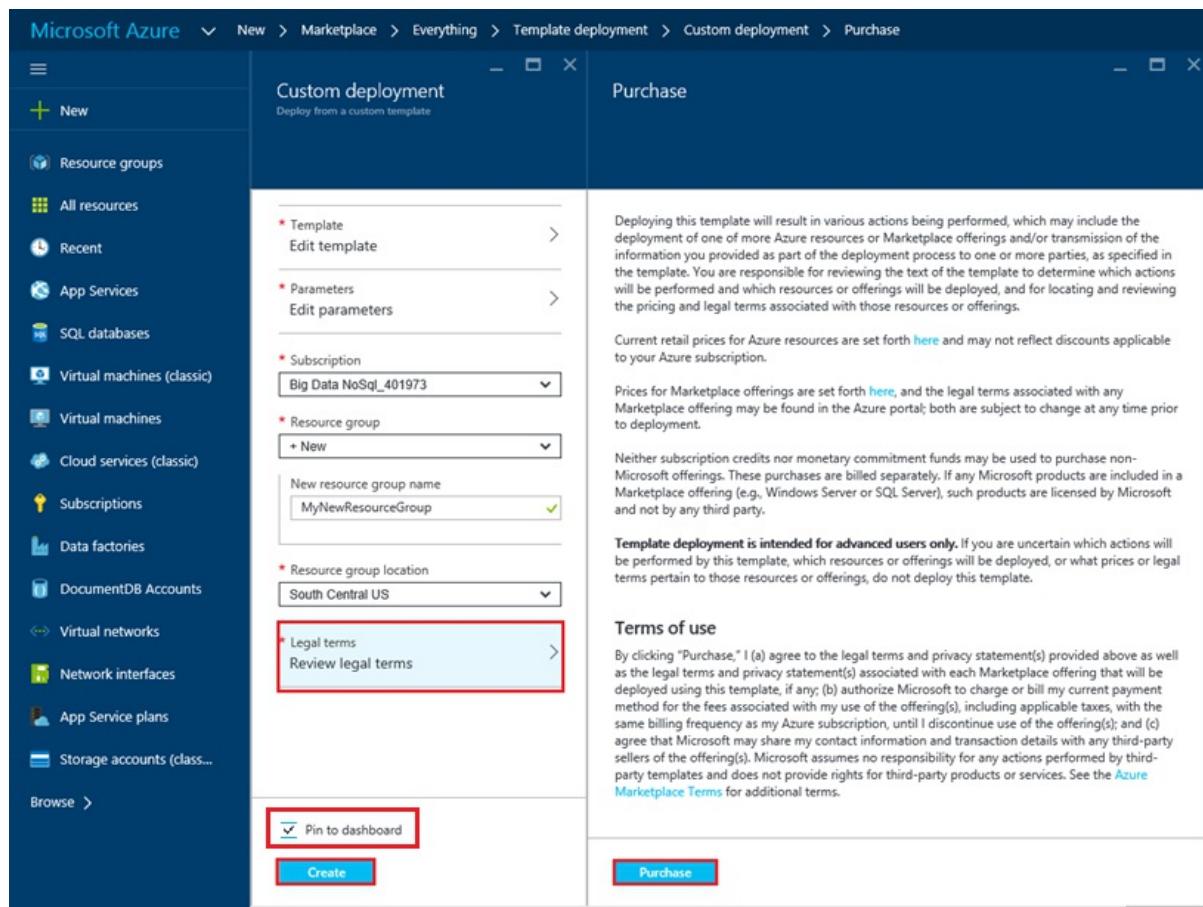
4. Click **Edit parameters**, provide values for each of the mandatory parameters, and click **OK**. The parameters are as follows:
  - a. SITENAME: Specifies the App Service web app name and is used to construct the URL that you will use to access the web app (for example, if you specify "mydemodocdbwebapp", then the URL by which you access the web app is mydemodocdbwebapp.azurewebsites.net).
  - b. HOSTINGPLANNNAME: Specifies the name of App Service hosting plan to create.
  - c. LOCATION: Specifies the Azure location in which to create the Azure Cosmos DB and web app resources.
  - d. DATABASEACCOUNTNAME: Specifies the name of the Azure Cosmos DB account to create.



5. Choose an existing Resource group or provide a name to make a new resource group, and choose a location for the resource group.



6. Click **Review legal terms**, **Purchase**, and then click **Create** to begin the deployment. Select **Pin to dashboard** so the resulting deployment is easily visible on your Azure portal home page.



- When the deployment finishes, the Resource group pane opens.

The screenshot shows the 'Resource group' settings page for 'MyNewResourceGroup'. The left pane is titled 'Essentials' and shows basic information like Subscription name (Big Data), Last deployment (4/29/2016 (Succeeded)), and a list of resources: mydemotododocdb, mydemotodoplan, and mydemotodoappsite. The right pane is titled 'Settings' and includes sections for SUPPORT + TROUBLESHOOTING (Audit logs), GENERAL (Properties, Resources, Resource costs, Deployments, Alerts, Export template), and RESOURCE MANAGEMENT (Users, Tags).

- Click the Web App resource in the Resources list and then click **Application settings**

The screenshot shows the Azure portal interface. On the left, the 'Resource group' blade for 'MyDocDBRG' is visible, displaying basic information such as Subscription name, Last deployment, and Location. In the center, the 'mydocdbwebapp' blade is open, showing its configuration. The 'Essentials' section includes details like Resource group, Status (Running), Location (North Europe), and URLs. Below this is the 'Monitoring' section, which displays a chart of Requests and errors over time. On the right, the 'Settings' blade is open, providing options for troubleshooting, audit logs, resource health, site metrics, app service plan metrics, mitigation, live HTTP traffic, and new support requests. The 'Application settings' section is specifically highlighted with a red box.

9. Note how there are application settings present for the Azure Cosmos DB endpoint and each of the Azure Cosmos DB master keys.

 Application settings  
mydocdbwebapp

Save Discard

General settings

The 64-bit and Always On options can be enabled in Basic plans and higher. AutoSwap can only be enabled on Standard plans.

.NET Framework version ⓘ v4.6

PHP version ⓘ 5.4

Java version ⓘ Off

Python version ⓘ Off

Platform ⓘ 32-bit 64-bit

Web sockets ⓘ Off On

Always On ⓘ Off On

Managed Pipeline Version Integrated Classic

Auto swap destinations cannot be configured from production slot

Auto Swap Off On

Auto Swap Slot

Debugging

Remote debugging Off On

Remote Visual Studio version 2012 2013 2015

App settings

endpoint https://mydocdbwebaccct.d...  Slot setting ...

authKey I4cUDBeTw2rZsSg7mY4fXV...  Slot setting ...

Key Value  Slot setting ...

Connection strings

The connection string values are hidden [Show connection string values](#)

DocDBConnection < Hidden for Securi... Custom  Slot setting ...

Name Value SQL Database  Slot setting ...

10. Feel free to continue exploring the Azure Portal, or follow one of our Azure Cosmos DB [samples](#) to create your own Azure Cosmos DB application.

## Next steps

Congratulations! You've deployed Azure Cosmos DB, App Service web app and a sample web application using Azure Resource Manager templates.

- To learn more about Azure Cosmos DB, click [here](#).
- To learn more about Azure App Service Web apps, click [here](#).
- To learn more about Azure Resource Manager templates, click [here](#).

## What's changed

- For a guide to the change from Websites to App Service see: [Azure App Service and Its Impact on Existing Azure Services](#)

### NOTE

If you want to get started with Azure App Service before signing up for an Azure account, go to [Try App Service](#), where you can immediately create a short-lived starter web app in App Service. No credit cards required; no commitments.

# How to index Cosmos DB data using an indexer in Azure Cognitive Search

2/20/2020 • 14 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](#). The REST API version 2019-05-06-Preview provides preview 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 2019-05-06-Preview](#) to create the data source and indexer.
- For [Cassandra API \(preview\)](#) and [Gremlin API \(preview\)](#), you can only use the [REST API version 2019-05-06-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 the primary or secondary connection string from Cosmos DB, with an `AccountEndpoint` and an `AccountKey`. For MongoDB collections, add `ApiKind=MongoDb` to the end of the connection string and separate it from the connection string with a semicolon. For the Gremlin API and Cassandra API, use the instructions for the [REST API](#).
- **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.

**Import data**

[Connect to your data](#)   [Add cognitive search \(Optional\)](#)   [Customize target index](#)   [...](#)

Create and load a search index using data from an existing Azure data source in your current subscription. Azure Search crawls the data structure you provide, extracts searchable content, optionally enriches it with cognitive skills, and loads it into an index. [Learn more](#)

Data Source: Cosmos DB

\* Name: my-cosmosdb-datasource-name

\* Cosmos DB account: AccountEndpoint=https://westus-cosmosdb.documents.azure.com:4... [Choose an existing connection](#)

\* Database: hotel-demo-db

\* Collection: hotel-demo-coll

Query:

```
SELECT * FROM c WHERE c._ts >= @HighWaterMark ORDER BY c._ts
```

#### 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.

**Import data**

\* Key [?](#)  
HotelID

Suggerster name  Search mode [?](#)

[Delete](#)

FIELD NAME	TYPE	RETRIEVABLE	FILTERABLE	SORTABLE	FACEABLE	SEARCHABLE	ANALYZER	SUGGESTER
HotelID	Edm.String	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	Standard - Lucene <input type="button" value="▼"/>	<a href="#">...</a>
HotelName	Edm.String	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	Standard - Lucene <input type="button" value="▼"/>	<a href="#">...</a>
Description	Edm.String	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	English - Microsoft <input type="button" value="▼"/>	<a href="#">...</a>
Description_fr	Edm.String	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	French - Microsoft <input type="button" value="▼"/>	<a href="#">...</a>
Category	Edm.String	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	Standard - Lucene <input type="button" value="▼"/>	<a href="#">...</a>
Tags	Edm.String	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	Standard - Lucene <input type="button" value="▼"/>	<a href="#">...</a>
ParkingIncluded	Edm.Int64	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>			<a href="#">...</a>
SmokingAllowed	Edm.Int64	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>			<a href="#">...</a>
LastRenovationDate	Edm.DateTi...	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>			<a href="#">...</a>
Rating	Edm.Double	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>			<a href="#">...</a>

[Previous: Add cognitive search \(Optional\)](#) [Next: Create an indexer](#)

## 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.

## Import data

Connect to your data Add cognitive search (Optional) Customize target index [Create an indexer](#)

### Indexer

\* Name

my-cosmosdb-indexer 

Schedule 

Once

Hourly

Daily

Custom



Change tracking automatically configured with a high watermark policy.

Track deletions 



Description

(optional)

▼ Advanced options

[Previous: Customize target index](#)

**Submit**

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 2019-05-06-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](#) 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=2019-05-06
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=myCosmosDbDatabaseId",
        "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
<b>name</b>	Required. Choose any name to represent your data source object.
<b>type</b>	Required. Must be <code>cosmosdb</code> .
<b>credentials</b>	<p>Required. Must be a Cosmos DB connection string. For SQL collections, connection strings are in this format:</p> <div style="border: 1px solid #ccc; padding: 5px; width: fit-content;"> AccountEndpoint=&lt;Cosmos DB endpoint url&gt;;AccountKey=&lt;Cosmos DB auth key&gt;;Database=&lt;Cosmos DB database id&gt; </div> <p>For MongoDB collections, add <b>ApiKind=MongoDb</b> to the connection string:</p> <div style="border: 1px solid #ccc; padding: 5px; width: fit-content;"> AccountEndpoint=&lt;Cosmos DB endpoint url&gt;;AccountKey=&lt;Cosmos DB auth key&gt;;Database=&lt;Cosmos DB database id&gt;;ApiKind=MongoDb </div> <p>For Gremlin graphs and Cassandra tables, sign up for the <a href="#">gated indexer preview</a> to get access to the preview and information about how to format the credentials.</p> <p>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.</p>
<b>container</b>	<p>Contains the following elements:</p> <p><b>name:</b> Required. Specify the ID of the database collection to be indexed.</p> <p><b>query:</b> Optional. You can specify a query to flatten an arbitrary JSON document into a flat schema that Azure Cognitive Search can index.</p> <p>For the MongoDB API, Gremlin API, and Cassandra API, queries are not supported.</p>
<b>dataChangeDetectionPolicy</b>	Recommended. See <a href="#">Indexing Changed Documents</a> section.
<b>dataDeletionDetectionPolicy</b>	Optional. See <a href="#">Indexing Deleted Documents</a> 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=2019-05-06
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=2019-05-06
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.

- [microsoft.azure.search.models.datasource](#)
- [microsoft.azure.search.models.datasourcetype](#)
- [microsoft.azure.search.models.index](#)
- [microsoft.azure.search.models.indexer](#)

## 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=2019-05-06  
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=myCosmosDbDatabaseId"  
    },  
    "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

2/18/2020 • 9 minutes to read • [Edit Online](#)

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

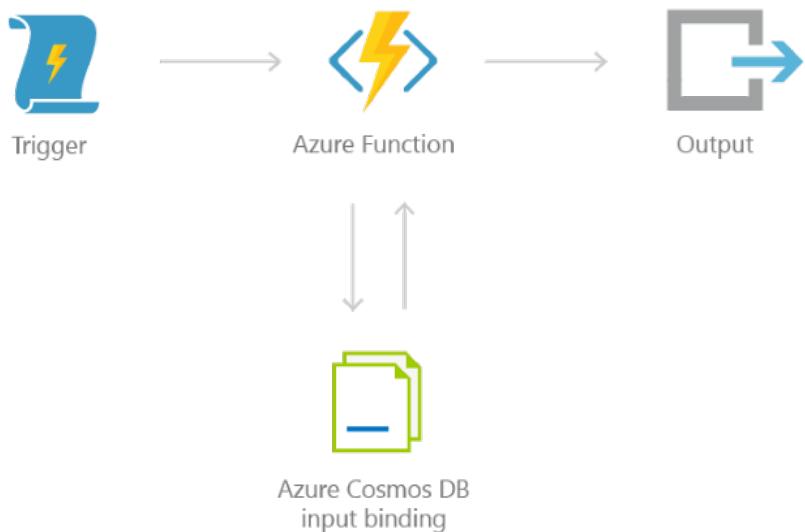
Currently, Azure Functions trigger, input bindings, and output bindings for Cosmos DB are supported for use with the SQL API only. For all other Azure Cosmos DB APIs, you should access the database from your function by using the static client for your API.

The following diagram illustrates each of these three integrations:

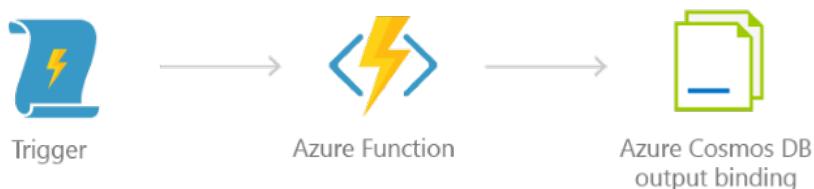
Modified items



### Use an Azure Cosmos DB trigger to invoke an Azure Function



### Use an input binding to get data from Azure Cosmos DB



### Use an output binding to write data to Azure Cosmos DB

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.

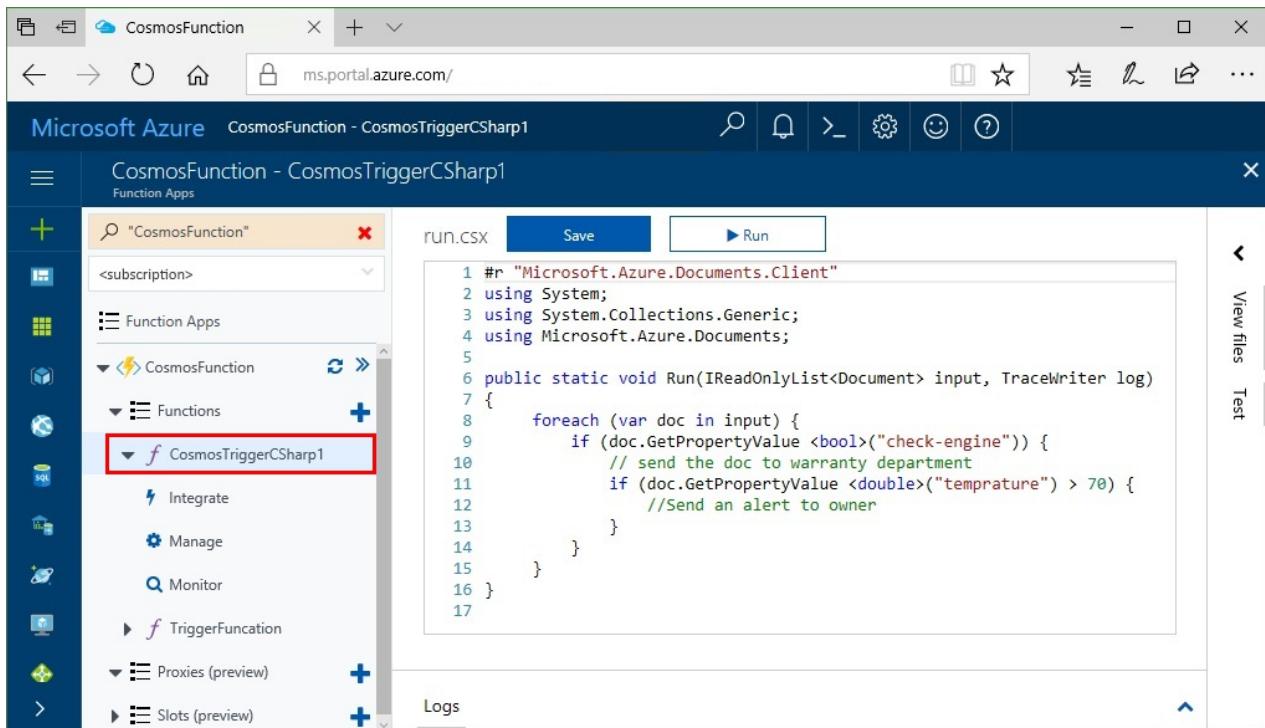
### IoT use case - Azure Functions trigger and output binding for Cosmos DB

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.



The screenshot shows the Azure portal interface for a function named "CosmosFunction - CosmosTriggerCSharp1". The left sidebar shows the function app structure with a "Functions" section containing "CosmosTriggerCSharp1". The main area displays the "run.csx" code:

```
#r "Microsoft.Azure.Documents.Client"
using System;
using System.Collections.Generic;
using Microsoft.Azure.Documents;
using Microsoft.Azure.Documents.Client;
public static void Run(IReadOnlyList<Document> input, TraceWriter log)
{
    foreach (var doc in input)
    {
        if (doc.GetPropertyValue <bool>("check-engine"))
        {
            // send the doc to warranty department
            if (doc.GetPropertyValue <double>("temprature") > 70)
            {
                //Send an alert to owner
            }
        }
    }
}
```

### 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
1 module.exports = function (context, myTimer) {
2     var timeStamp = new Date().toISOString();
3     context.log('JavaScript timer trigger function ran!', timeStamp);
4
5     var accounts = context.bindings.accounts;
6     var lowBalanceThreshold = 100;
7
8     for (var i = 0; i < accounts.length; i++) {
9         var account = accounts[i];
10        var accountNumber = account['$v']['number']['$v'];
11        var accountBalance = account['$v']['balance']['$v'];
12
13        context.log('The account #' + accountNumber + ' has a balance of: $' + accountBalance);
14
15        if (accountBalance < lowBalanceThreshold){
16            // Notify account owners about low balance.
17        }
18    }
19    context.done();
20 };

```

```

run.csx Save ▶ Save and run
1 #r "Microsoft.Azure.Documents.Client"
2 using System;
3 using System.Collections.Generic;
4 using Microsoft.Azure.Documents;
5
6 public static void Run(IReadOnlyList<Document> documents, TraceWriter log)
7 {
8     double lowBalanceThreshold = 100;
9     foreach (var doc in documents) {
10         if (doc.GetProperty<double>("balance") < lowBalanceThreshold) {
11             //Send the mail to user
12             log.Verbose ("Balance low");
13         }
14     }
15 }
16

```

### 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 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

1/16/2020 • 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	<a href="#">Install the package</a>
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
<b>type</b>	n/a	Must be set to <code>cosmosDBTrigger</code> .
<b>direction</b>	n/a	Must be set to <code>in</code> . This parameter is set automatically when you create the trigger in the Azure portal.
<b>name</b>	n/a	The variable name used in function code that represents the list of documents with changes.
<b>connectionStringSetting</b>	<b>ConnectionStringSetting</b>	The name of an app setting that contains the connection string used to connect to the Azure Cosmos DB account being monitored.
<b>databaseName</b>	<b>DatabaseName</b>	The name of the Azure Cosmos DB database with the collection being monitored.
<b>collectionName</b>	<b>CollectionName</b>	The name of the collection being monitored.
<b>leaseConnectionStringSetting</b>	<b>LeaseConnectionStringSetting</b>	(Optional) The name of an app setting that contains the connection string to the service which holds the lease collection. When not set, the <code>connectionStringSetting</code> 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.
<b>leaseDatabaseName</b>	<b>LeaseDatabaseName</b>	(Optional) The name of the database that holds the collection used to store leases. When not set, the value of the <code>databaseName</code> setting is used. This parameter is automatically set when the binding is created in the portal.
<b>leaseCollectionName</b>	<b>LeaseCollectionName</b>	(Optional) The name of the collection used to store leases. When not set, the value <code>leases</code> is used.
<b>createLeaseCollectionIfNotExists</b>	<b>CreateLeaseCollectionIfNotExists</b>	(Optional) When set to <code>true</code> , the leases collection is automatically created when it doesn't already exist. The default value is <code>false</code> .

FUNCTION.JSON PROPERTY	ATTRIBUTE PROPERTY	DESCRIPTION
<b>leasesCollectionThroughput</b>	<b>LeasesCollectionThroughput</b>	(Optional) Defines the amount of Request Units to assign when the leases collection is created. This setting is only used When <code>createLeaseCollectionIfNotExists</code> is set to <code>true</code> . This parameter is automatically set when the binding is created using the portal.
<b>leaseCollectionPrefix</b>	<b>LeaseCollectionPrefix</b>	(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.
<b>feedPollDelay</b>	<b>FeedPollDelay</b>	(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).
<b>leaseAcquireInterval</b>	<b>LeaseAcquireInterval</b>	(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).
<b>leaseExpirationInterval</b>	<b>LeaseExpirationInterval</b>	(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).
<b>leaseRenewInterval</b>	<b>LeaseRenewInterval</b>	(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).
<b>checkpointFrequency</b>	<b>CheckpointFrequency</b>	(Optional) When set, it defines, in milliseconds, the interval between lease checkpoints. Default is always after each Function call.
<b>maxItemsPerInvocation</b>	<b>MaxItemsPerInvocation</b>	(Optional) When set, it customizes the maximum amount of items received per Function call.

FUNCTION.JSON PROPERTY	ATTRIBUTE PROPERTY	DESCRIPTION
<b>startFromBeginning</b>	<b>StartFromBeginning</b>	(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 <code>true</code> 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.

- [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 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 toItem in ToDoItems)
            {
                log.Info(toItem.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 toItem in ToDoItems)
            {
                log.Info(toItem.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
<b>type</b>	n/a	Must be set to <code>documentdb</code> .
<b>direction</b>	n/a	Must be set to <code>in</code> .
<b>name</b>	n/a	Name of the binding parameter that represents the document in the function.
<b>databaseName</b>	<b>DatabaseName</b>	The database containing the document.
<b>collectionName</b>	<b>CollectionName</b>	The name of the collection that contains the document.
<b>id</b>	<b>Id</b>	The ID of the document to retrieve. This property supports <a href="#">binding expressions</a> . Don't set both the <b>id</b> and <b>sqlQuery</b> properties. If you don't set either one, the entire collection is retrieved.
<b>sqlQuery</b>	<b>SqlQuery</b>	An Azure Cosmos DB SQL query used for retrieving multiple documents. The property supports runtime bindings, as in this example: <code>SELECT * FROM c WHERE c.departmentId = {departmentId}</code> . Don't set both the <b>id</b> and <b>sqlQuery</b> properties. If you don't set either one, the entire collection is retrieved.
<b>connection</b>	<b>ConnectionStringSetting</b>	The name of the app setting containing your Azure Cosmos DB connection string.
<b>partitionKey</b>	<b>PartitionKey</b>	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.

- [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
<b>type</b>	n/a	Must be set to <code>documentdb</code> .
<b>direction</b>	n/a	Must be set to <code>out</code> .
<b>name</b>	n/a	Name of the binding parameter that represents the document in the function.
<b>databaseName</b>	<b>DatabaseName</b>	The database containing the collection where the document is created.
<b>collectionName</b>	<b>CollectionName</b>	The name of the collection where the document is created.
<b>createIfNotExists</b>	<b>CreateIfNotExists</b>	A boolean value to indicate whether the collection is created when it doesn't exist. The default is <i>false</i> because new collections are created with reserved throughput, which has cost implications. For more information, see the <a href="#">pricing page</a> .
<b>partitionKey</b>	<b>PartitionKey</b>	When <code>CreateIfNotExists</code> is true, defines the partition key path for the created collection.
<b>collectionThroughput</b>	<b>CollectionThroughput</b>	When <code>CreateIfNotExists</code> is true, defines the <a href="#">throughput</a> of the created collection.
<b>connection</b>	<b>ConnectionStringSetting</b>	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	<a href="#">CosmosDB Error Codes</a>

## 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

2/18/2020 • 10 minutes to read • [Edit Online](#)

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

You can use one of the following tools or SDKs to use the copy activity with a pipeline. Select a link for step-by-step instructions:

- [Copy data tool](#)
- [Azure portal](#)
- [.NET SDK](#)
- [Python SDK](#)
- [Azure PowerShell](#)
- [REST API](#)

- 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 <b>type</b> property must be set to <b>CosmosDb</b> .	Yes
connectionString	<p>Specify information that's required to connect to the Azure Cosmos DB database.</p> <p><b>Note:</b> 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 <code>accountKey</code> configuration out of the connection string. Refer to the following samples and <a href="#">Store credentials in Azure Key Vault</a> article with more details.</p>	Yes
connectVia	The <a href="#">Integration Runtime</a> 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 <b>type</b> property of the dataset must be set to <b>CosmosDbSqlApiCollection</b> .	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 <b>type</b> property of the copy activity source must be set to <b>CosmosDbSqlApiSource</b> .	Yes
query	Specify the Azure Cosmos DB query to read data.  Example: <pre>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 &gt; \\"2009-01-01T00:00:00\\"</pre>	No  If not specified, this SQL statement is executed: <pre>select &lt;columns defined in structure&gt; from mycollection</pre>
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 **source** section:

PROPERTY	DESCRIPTION	REQUIRED
type	The <b>type</b> property of the Copy Activity sink must be set to <b>CosmosDbSqlApiSink</b> .	Yes

PROPERTY	DESCRIPTION	REQUIRED
writeBehavior	<p>Describes how to write data to Azure Cosmos DB. Allowed values: <b>insert</b> and <b>upsert</b>.</p> <p>The behavior of <b>upsert</b> is to replace the document if a document with the same ID already exists; otherwise, insert the document.</p> <p><b>Note:</b> 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 <b>upsert</b> to work as expected, your document has an ID.</p>	No (the default is <b>insert</b> )
writeBatchSize	Data Factory uses the <a href="#">Azure Cosmos DB bulk executor library</a> to write data to Azure Cosmos DB. The <b>writeBatchSize</b> property controls the size of documents that ADF provides to the library. You can try increasing the value for <b>writeBatchSize</b> to improve performance and decreasing the value if your document size being large - see below tips.	No (the default is <b>10,000</b> )
disableMetricsCollection	Data Factory collects metrics such as Cosmos DB RUs for copy performance optimization and recommendations. If you are concerned with this behavior, specify <code>true</code> to turn it off.	No (default is <code>false</code> )

#### 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

2/4/2020 • 7 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.

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 is limited to 10 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.

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.

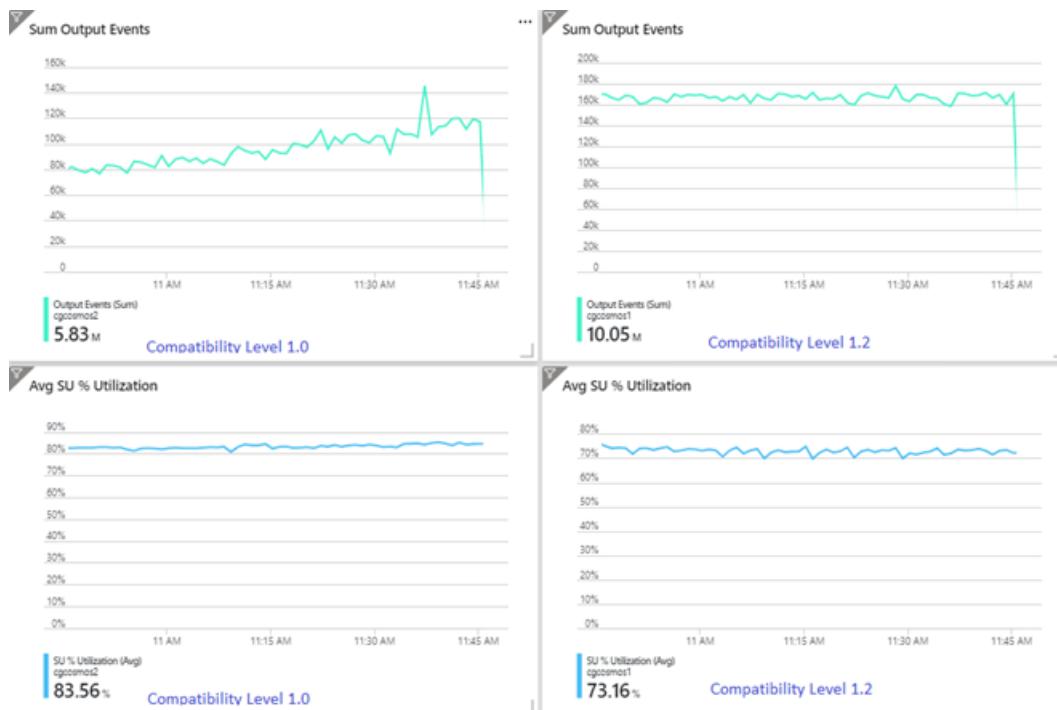
## Improved throughput with compatibility level 1.2

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.

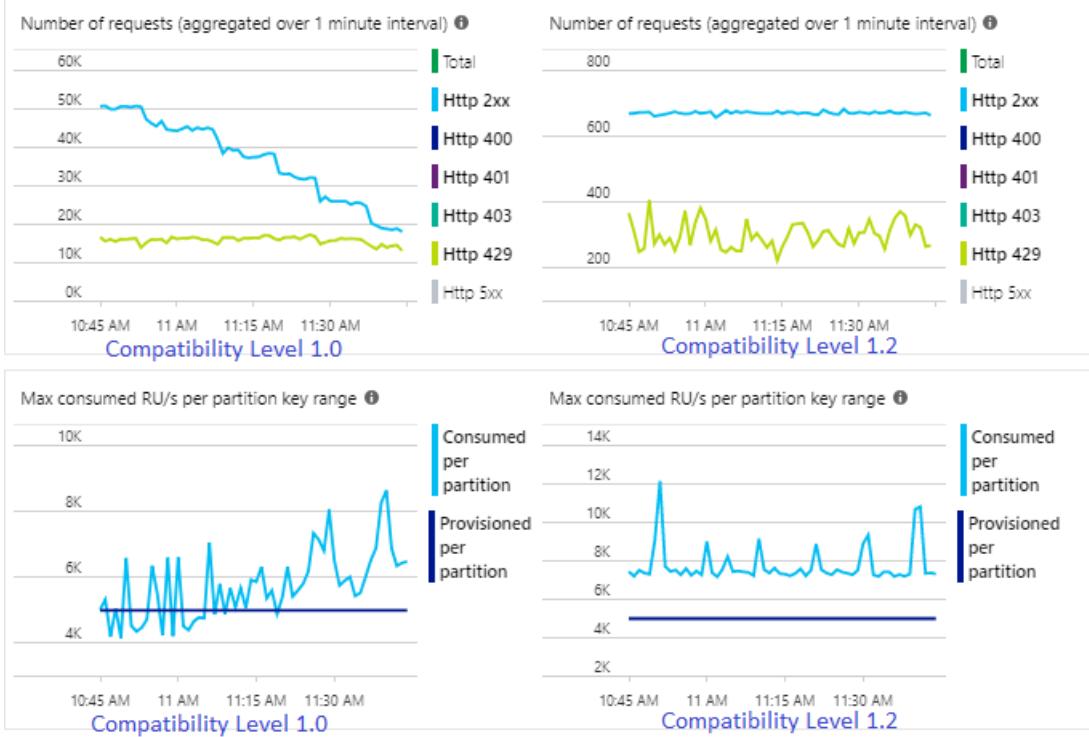
The improved writing mechanism is available under a new compatibility level because of a difference in upsert behavior. With levels before 1.2, the upsert behavior is to insert or merge the document. With 1.2, upsert behavior is modified to insert or replace the document.

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.

**Cosmos DB**

New output

\* Output alias

Provide Cosmos DB settings manually  
 Select Cosmos DB from your subscriptions

Subscription

\* Account id

Account key

\* Database

Create new  
 Use existing

\* Collection name pattern

Document id

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 <code>MyContainer</code> . One container named <code>MyContainer</code> 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)

# Connect to Azure Cosmos DB using BI analytics tools with the ODBC driver

1/23/2020 • 14 minutes to read • [Edit Online](#)

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.

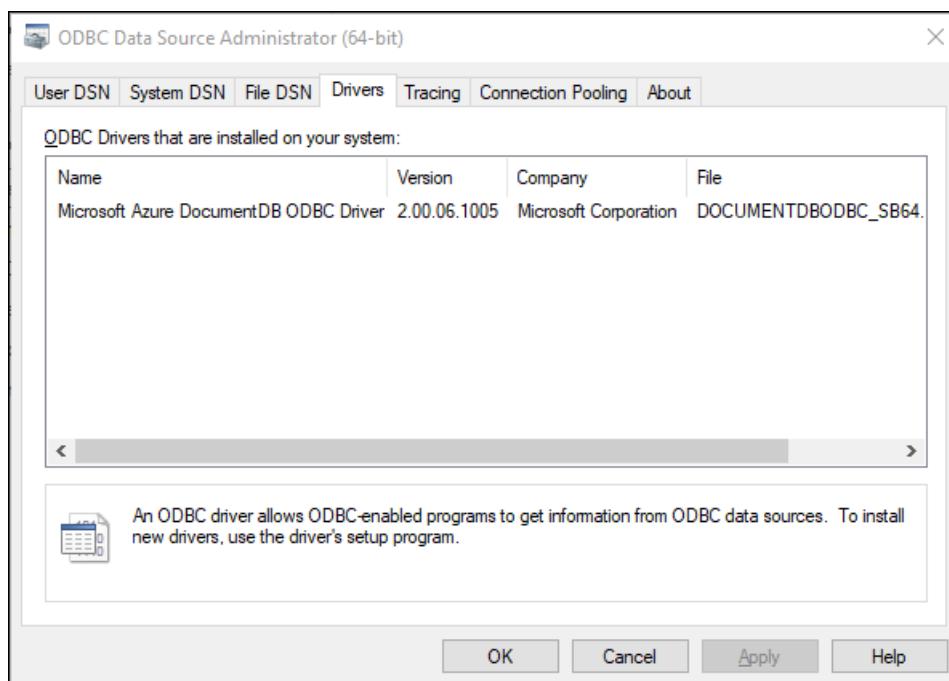
## Step 1: Install the Azure Cosmos DB ODBC driver

1. Download the drivers for your environment:

INSTALLER	SUPPORTED OPERATING SYSTEMS
<a href="#">Microsoft Azure Cosmos DB ODBC 64-bit.msi</a> 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.
<a href="#">Microsoft Azure Cosmos DB ODBC 32x64-bit.msi</a> 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.
<a href="#">Microsoft Azure Cosmos DB ODBC 32-bit.msi</a> for 32-bit Windows	32-bit versions of Windows 8.1 or later, Windows 8, Windows 7, Windows XP, and Windows Vista.

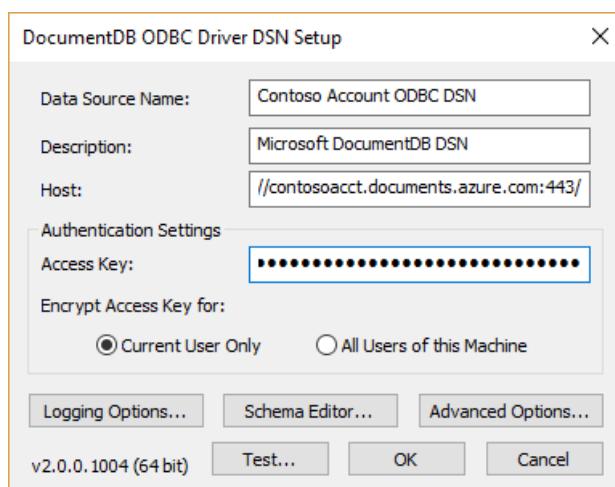
Run the msi file locally, which starts the **Microsoft Azure Cosmos DB ODBC Driver Installation Wizard**.

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.

The screenshot shows the 'contoso-account Keys' page in the Azure portal. On the left, there's a sidebar with options like Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Quick start, Notifications, Data Explorer, Settings, and Replicate data globally. The main area has tabs for 'Read-write Keys' and 'Read-only Keys'. Under 'Read-only Keys', there are fields for 'URI', 'PRIMARY KEY', and 'SECONDARY KEY'. Below these are sections for 'PRIMARY CONNECTION STRING' and 'SECONDARY CONNECTION STRING'. A red box highlights the 'Host' field under 'PRIMARY KEY' with the text 'Click to copy'. Another red box highlights the 'Access Key' field with the same 'Click to copy' text.

- **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**

The screenshot shows the Windows Registry Editor. The left pane shows a tree view of registry keys under 'Computer\HKEY\_LOCAL\_MACHINE\SOFTWARE\ODBC\ODBC.INI\Contoso Account ODBC DSN'. The right pane is a table with columns: Name, Type, and Data. It shows two entries: '(Default)' with Type REG\_SZ and Data '(value not set)', and 'IgnoreSessionToken' with Type REG\_SZ and Data '1'. A red box highlights the 'IgnoreSessionToken' entry.

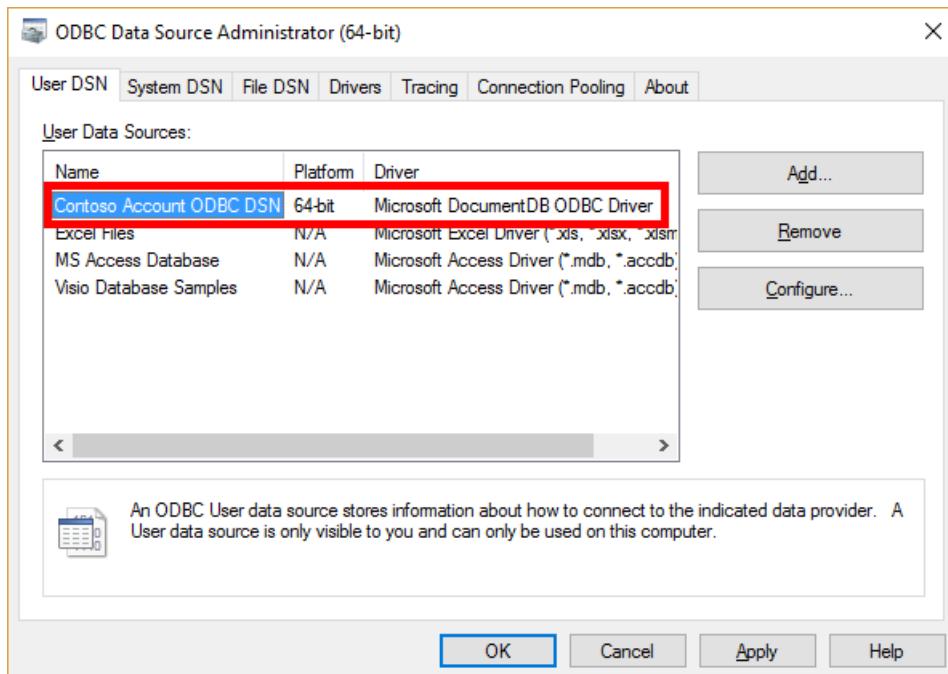
- **Query Consistency:** Select the [consistency level](#) for your operations. The default is Session.

- **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.

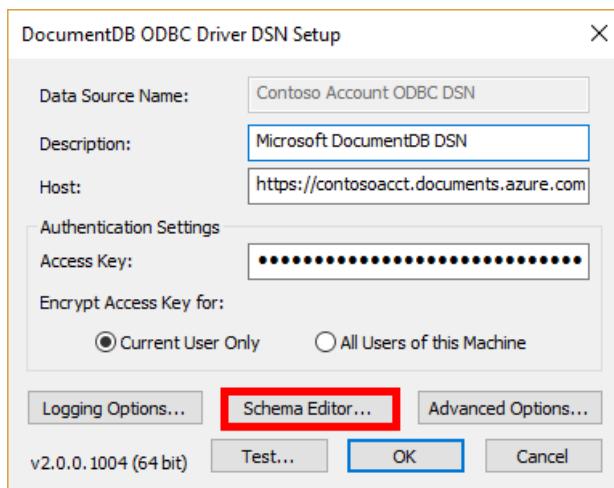
6. Once you complete and close the **Azure Cosmos DB ODBC Driver DSN Setup** window, the new User DSN is added to the User DSN tab.



## 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.

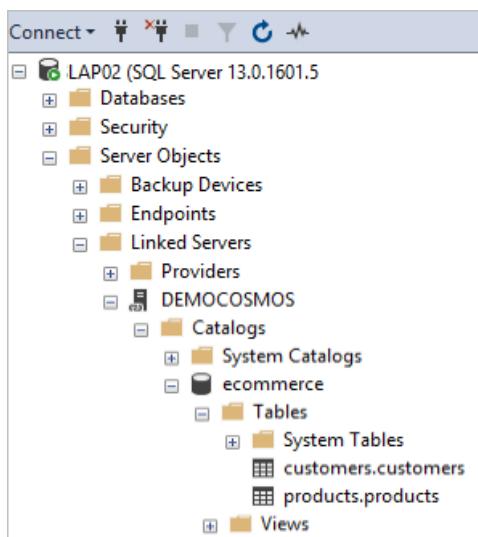
```
USE [master]
GO

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:

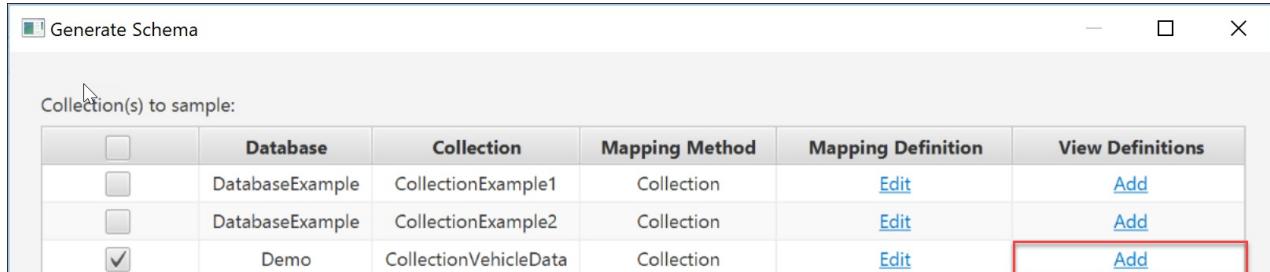
```
Msg 7312, Level 16, State 1, Line 44
```

```
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

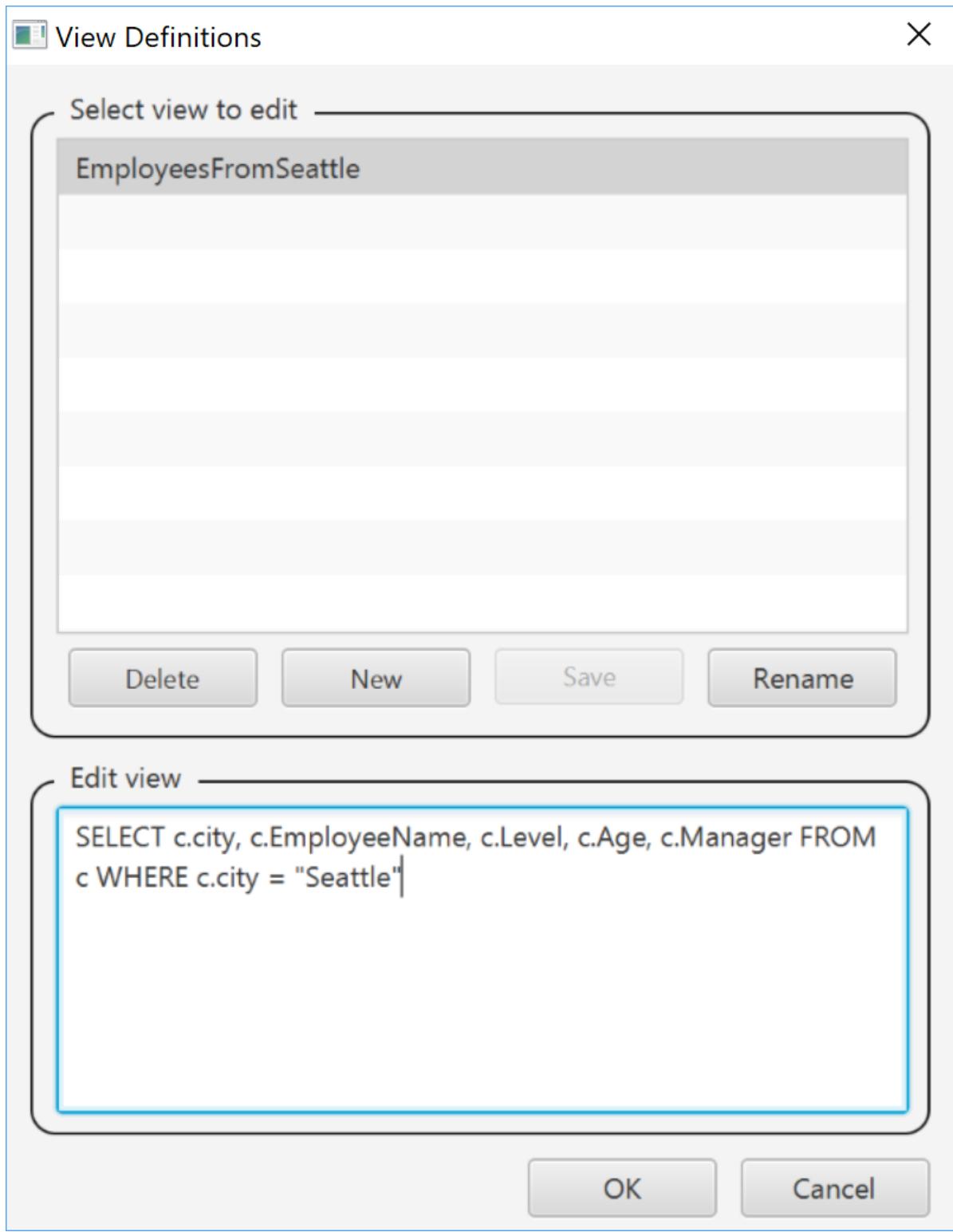
You can define and create views as part of the sampling process. These views are equivalent to SQL views. They are read-only and are scope the selections and projections of the Azure Cosmos DB SQL query defined.

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**.

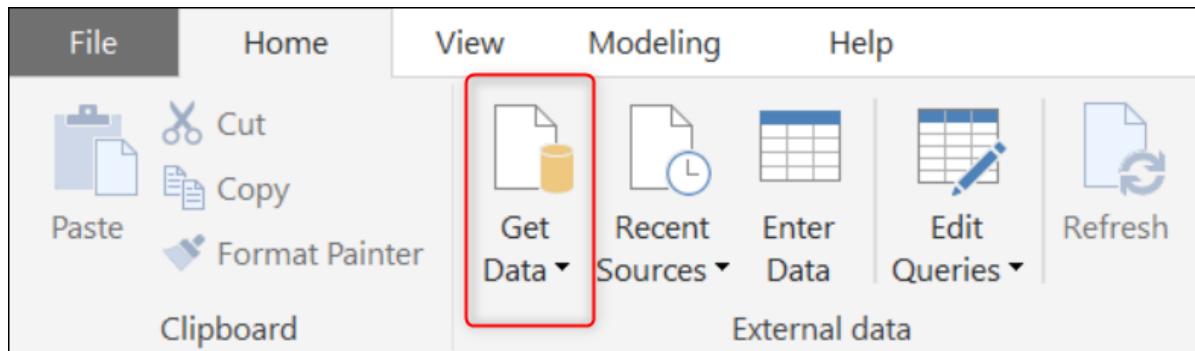


You can create as many views as you like. Once you are done defining the views, you can then sample the data.

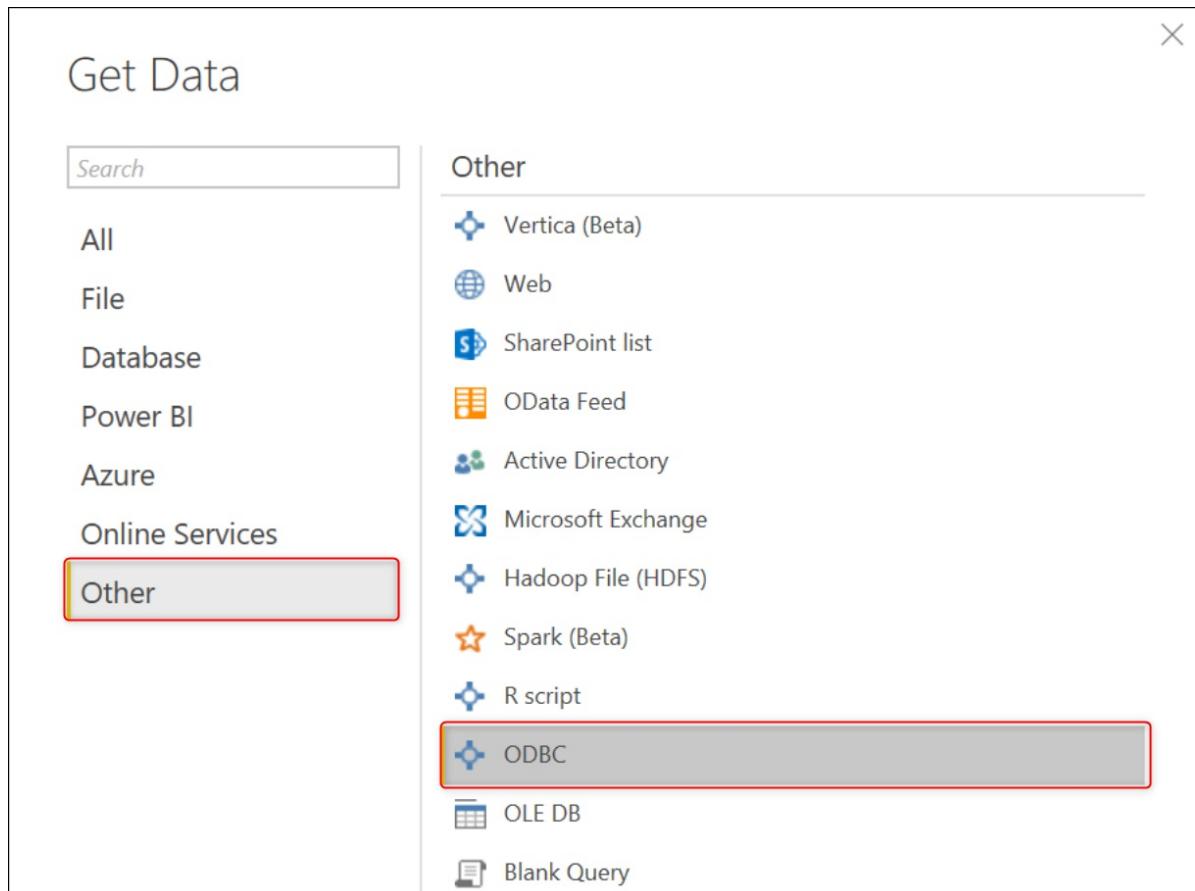
## Step 5: View your data in BI tools such as Power BI Desktop

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.

The screenshot shows the 'Navigator' window in Power BI desktop. On the left, there's a tree view of data sources. Under 'Demo [3]', there's a folder named 'CollectionVehicleData [1]'. Inside this folder, a table named 'CollectionVehicleData' is selected, indicated by a red border around its entry in the tree view. The right side of the window shows the name 'CollectionVehicleData'.

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](#).

# Options to migrate your on-premises or cloud data to Azure Cosmos DB

10/24/2019 • 5 minutes to read • [Edit Online](#)

You can load data from various data sources to Azure Cosmos DB. Additionally, since Azure Cosmos DB supports multiple APIs, the targets can be any of the existing APIs. 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 Cassandra, Azure SQL Server, 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	CONSIDERATIONS
Offline	<a href="#">Data Migration Tool</a>	<ul style="list-style-type: none"><li>• Easy to set up and supports multiple sources</li><li>• Not suitable for large datasets</li></ul>

MIGRATION TYPE	SOLUTION	CONSIDERATIONS
Offline	<a href="#">Azure Data Factory</a>	<ul style="list-style-type: none"> <li>• Easy to set up and supports multiple sources</li> <li>• Makes use of the Azure Cosmos DB bulk executor library</li> <li>• Suitable for large datasets</li> <li>• Lack of checkpointing - It means that if an issue occurs during the course of migration, you need to restart the whole migration process</li> <li>• Lack of a dead letter queue - It means that a few erroneous files can stop the entire migration process.</li> </ul>
Offline	<a href="#">Azure Cosmos DB Spark connector</a>	<ul style="list-style-type: none"> <li>• Makes use of the Azure Cosmos DB bulk executor library</li> <li>• Suitable for large datasets</li> <li>• Needs a custom Spark setup</li> <li>• Spark is sensitive to schema inconsistencies and this can be a problem during migration</li> </ul>
Offline	<a href="#">Custom tool with Cosmos DB bulk executor library</a>	<ul style="list-style-type: none"> <li>• Provides checkpointing, dead-lettering capabilities which increases migration resiliency</li> <li>• Suitable for very large datasets (10 TB+)</li> <li>• Requires custom setup of this tool running as an App Service</li> </ul>
Online	<a href="#">Cosmos DB Functions + ChangeFeed API</a>	<ul style="list-style-type: none"> <li>• Easy to set up</li> <li>• Works only if the source is an Azure Cosmos DB container</li> <li>• Not suitable for large datasets</li> <li>• Does not capture deletes from the source container</li> </ul>
Online	<a href="#">Custom Migration Service using ChangeFeed</a>	<ul style="list-style-type: none"> <li>• Provides progress tracking</li> <li>• Works only if the source is an Azure Cosmos DB container</li> <li>• Works for larger datasets as well</li> <li>• Requires the user to set up an App Service to host the Change feed processor</li> <li>• Does not capture deletes from the source container</li> </ul>
Online	<a href="#">Striim</a>	<ul style="list-style-type: none"> <li>• Works with a large variety of sources like Oracle, DB2, SQL Server</li> <li>• Easy to build ETL pipelines and provides a dashboard for monitoring</li> <li>• Supports larger datasets</li> <li>• Since this is a third-party tool, it needs to be purchased from the marketplace and installed in the user's environment</li> </ul>

## Azure Cosmos DB Mongo API

MIGRATION TYPE	SOLUTION	CONSIDERATIONS
Offline	<a href="#">Data Migration Tool</a>	<ul style="list-style-type: none"> <li>• Easy to set up and supports multiple sources</li> <li>• Not suitable for large datasets</li> </ul>
Offline	<a href="#">Azure Data Factory</a>	<ul style="list-style-type: none"> <li>• Easy to set up and supports multiple sources</li> <li>• Makes use of the Azure Cosmos DB bulk executor library</li> <li>• Suitable for large datasets</li> <li>• Lack of checkpointing means that any issue during the course of migration would require a restart of the whole migration process</li> <li>• Lack of a dead letter queue would mean that a few erroneous files could stop the entire migration process</li> <li>• Needs custom code to increase read throughput for certain data sources</li> </ul>
Offline	<a href="#">Existing Mongo Tools (mongodump, mongorestore, Studio3T)</a>	<ul style="list-style-type: none"> <li>• Easy to set up and integration</li> <li>• Needs custom handling for throttles</li> </ul>
Online	<a href="#">Azure Database Migration Service</a>	<ul style="list-style-type: none"> <li>• Makes use of the Azure Cosmos DB bulk executor library</li> <li>• Suitable for large datasets and takes care of replicating live changes</li> <li>• Works only with other MongoDB sources</li> </ul>

## Azure Cosmos DB Cassandra API

MIGRATION TYPE	SOLUTION	CONSIDERATIONS
Offline	<a href="#">cqlsh COPY command</a>	<ul style="list-style-type: none"> <li>• Easy to set up</li> <li>• Not suitable for large datasets</li> <li>• Works only when the source is a Cassandra table</li> </ul>
Offline	<a href="#">Copy table with Spark</a>	<ul style="list-style-type: none"> <li>• Can make use of Spark capabilities to parallelize transformation and ingestion</li> <li>• Needs configuration with a custom retry policy to handle throttles</li> </ul>
Online	<a href="#">Striim (from Oracle DB/Apache Cassandra)</a>	<ul style="list-style-type: none"> <li>• Works with a large variety of sources like Oracle, DB2, SQL Server</li> <li>• Easy to build ETL pipelines and provides a dashboard for monitoring</li> <li>• Supports larger datasets</li> <li>• Since this is a third-party tool, it needs to be purchased from the marketplace and installed in the user's environment</li> </ul>

MIGRATION TYPE	SOLUTION	CONSIDERATIONS
Online	<a href="#">Blitz (from Oracle DB/Apache Cassandra)</a>	<ul style="list-style-type: none"> <li>Supports larger datasets</li> <li>Since this is a third-party tool, it needs to be purchased from the marketplace and installed in the user's environment</li> </ul>

## 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 data to Azure Cosmos DB SQL API account using Striim

9/15/2019 • 8 minutes to read • [Edit Online](#)

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.

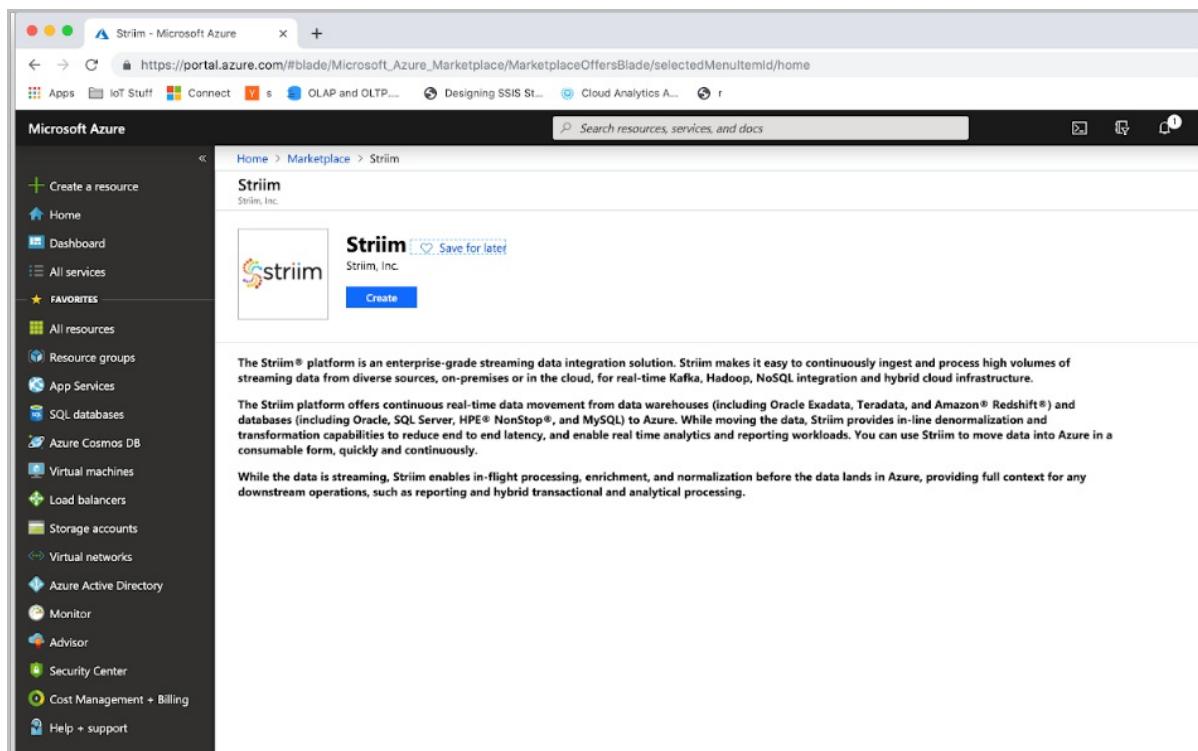
This article shows how to use Striim to migrate data from an **Oracle database** to an **Azure Cosmos DB SQL API account**.

## Prerequisites

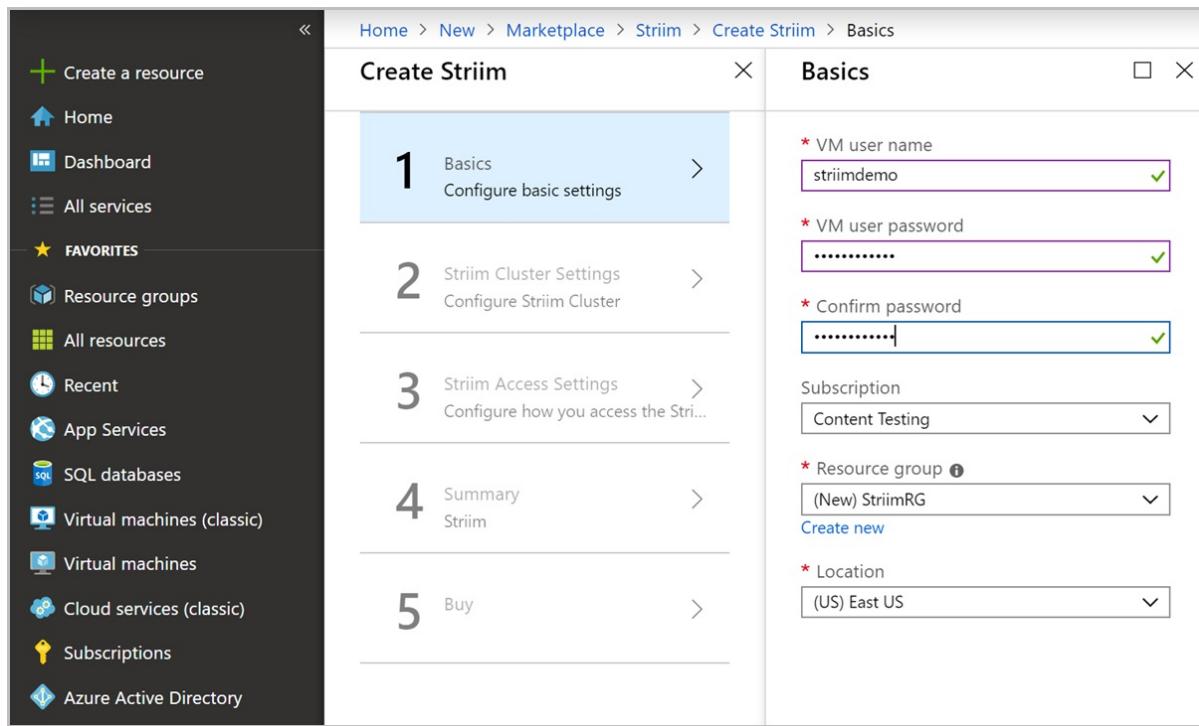
- If you don't have an [Azure subscription](#), create a [free account](#) before you begin.
- An Oracle database running on-premises with some data in it.

## 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 <b>Standalone</b> or <b>Cluster</b> 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.

The screenshot shows the Azure portal interface for creating a Striim instance. On the left, a sidebar lists various services like Home, Dashboard, All services, Favorites, and App Services. The main area displays a five-step wizard titled 'Create Striim'. Step 1: Basics (Done) has a green checkmark. Step 2: Striim Cluster Settings (Done) has a green checkmark. Step 3: Striim Access Settings (Configure how you access the Striim...) is currently selected and highlighted in blue. Step 4: Summary (Striim) and Step 5: Buy both have arrows pointing right. To the right of the wizard, a separate window titled 'Striim Access Settings' is open, showing configuration options:

- \* Public IP address: (new) striimMasterNodeIP
- \* Domain name for Striim: striimdemo1 eastus.cloudapp.azure.com
- \* Striim admin password: (redacted)
- \* Confirm admin password: (redacted)
- \* Virtual network: (new) striimVNET
- \* Subnet: Review subnet configuration

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

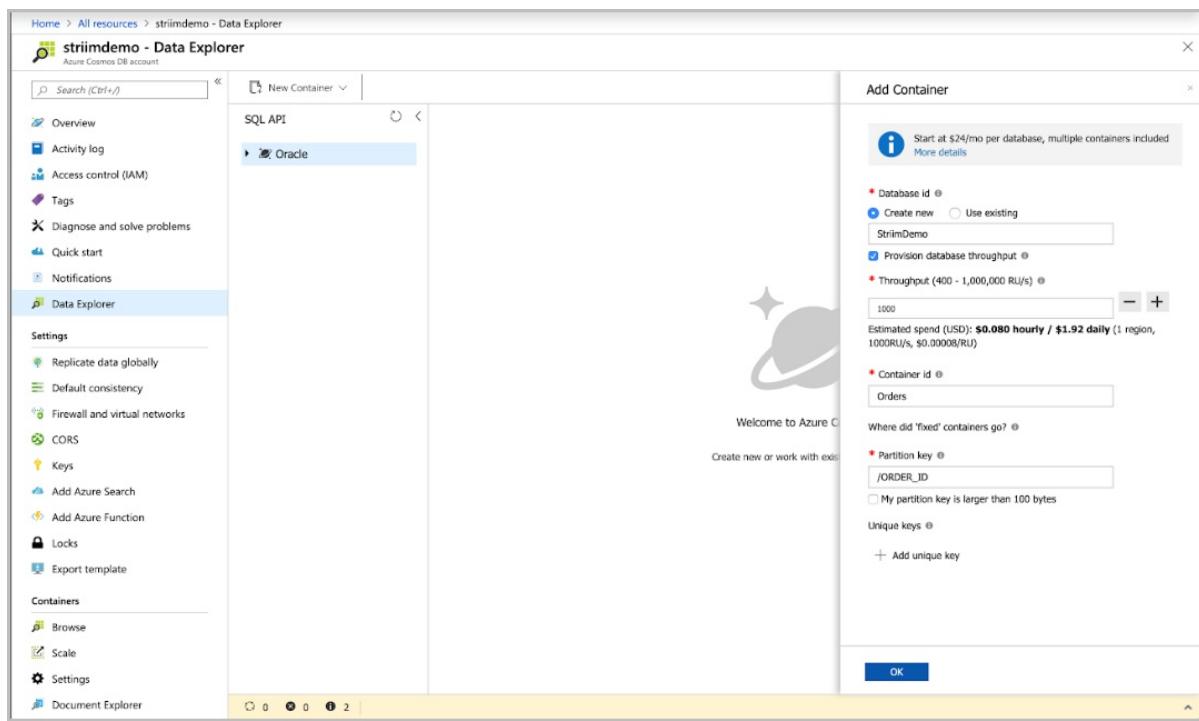
In this section, you configure the Oracle database as the source for data movement. You'll need the [Oracle JDBC driver](#) to connect to Oracle. To read changes from your source Oracle database, you can either use the [LogMiner](#) or the [XStream APIs](#). The Oracle JDBC driver must be present in 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 — striimd@striimdemo-masternode:~ — ssh striimd@striimdemo.westus.cloudapp.azure.com
[Striim-Edwards-MacBook-Pro:Striim edwardbell$ ssh striimd@striimdemo.westus.cloudapp.azure.com
>Password:
Last login: Tue Jul 16 21:04:40 2019 from 208.89.164.26
[striimd@striimdemo-masternode ~]$
```

- 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
hsperldata_root hsperldata_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 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
```

- Now, navigate back to Azure and copy the Public IP address of your Striim VM.

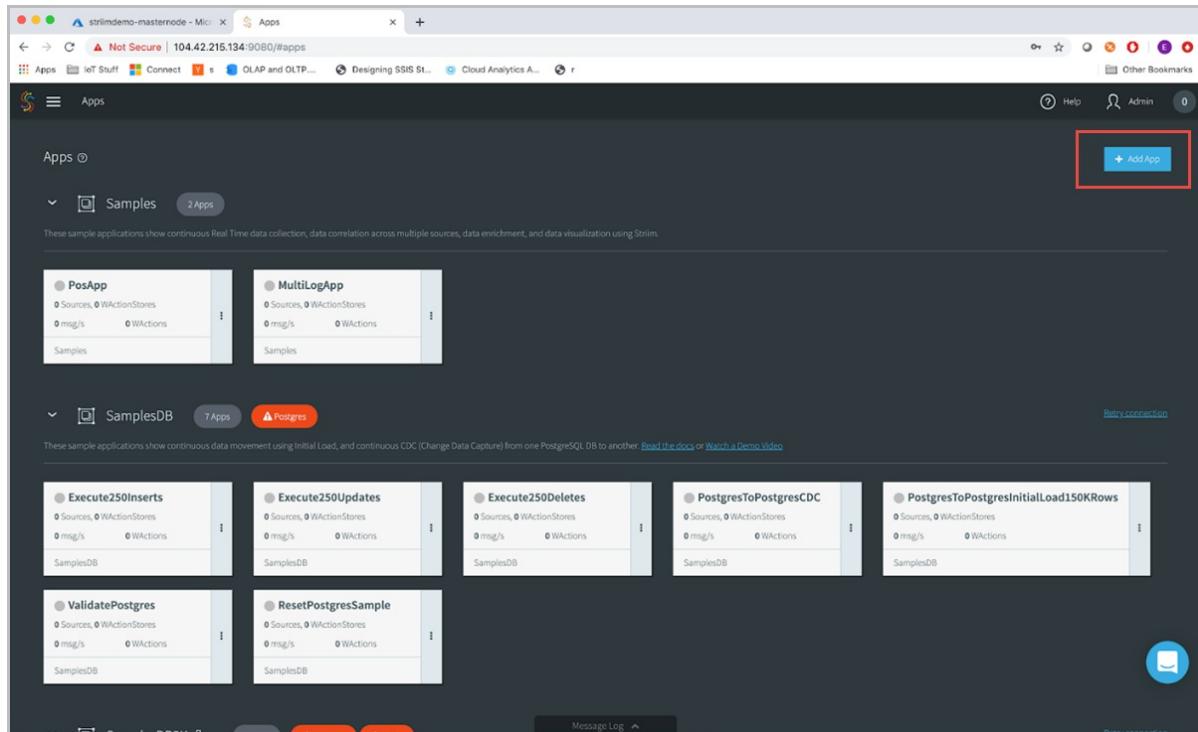
The screenshot shows the Azure portal interface for the 'striimdemo-masternode' virtual machine. On the left, there's a navigation menu with options like Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Settings, and Operations. The main pane displays the VM's details: Resource group (striimazuredemo), Status (Running), Location (West US), Subscription (Visual Studio Enterprise: BizSpark), and Subscription ID. It also shows the Computer name (striimdemo-masternode), Operating system (Linux (centos 7.3.1611)), Size (Standard F4s (4 vCPUs, 8 GB memory)), Ephemeral OS disk (N/A), and Public IP address (104.42.215.134). A 'Copy to clipboard' button is next to the Public IP address. Below this, Private IP address (10.1.0.4), Virtual network/subnet (striimVNET/Subnet-1), and DNS name (striimdemo.westus.cloudapp.azure.com) are listed. At the bottom, there are four performance charts: CPU (average), Network (total), Disk bytes (total), and Disk operations/sec (average). The CPU chart shows a spike at 2:15 PM with a value of 1.58%. The Network chart shows Network In Total (Sum) at 57.5 MB and Network Out Total (Sum) at 11.9 MB. The Disk bytes chart shows Disk Read Bytes (Sum) at 5.86 MB and Disk Write Bytes (Sum) at 453.06 MB. The Disk operations/sec chart shows an average of 8/s.

- 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.

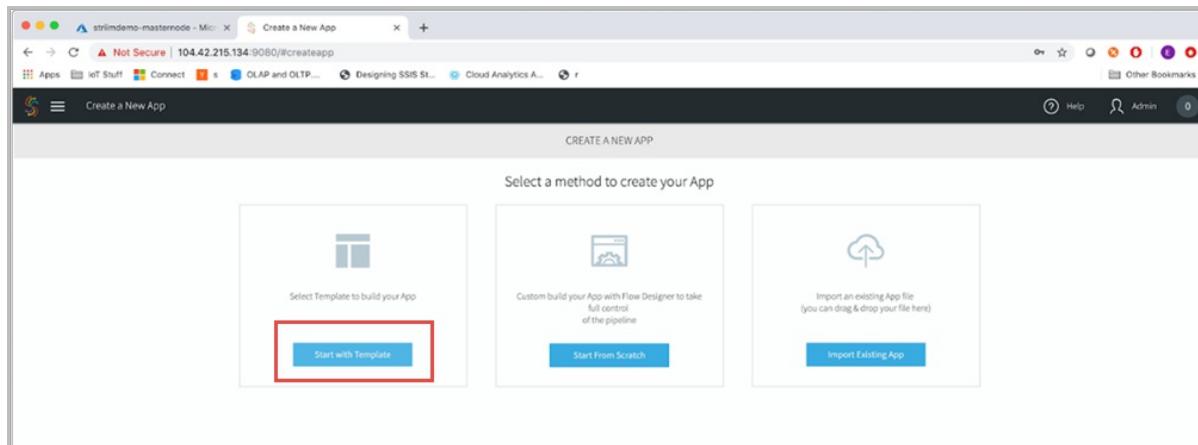
The screenshot shows a web browser window with the URL '104.42.215.134:9080'. The page title is 'striim - Microsoft Edge'. The content is the Striim login screen, featuring the Striim logo at the top. Below it is a login form with two input fields: 'admin' in the username field and 'Password' in the password field. A blue 'LOG IN' button is at the bottom of the form.

- 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.
- Select the **Apps** pane, we'll focus on this pane for now. There are a variety of sample apps that you can use

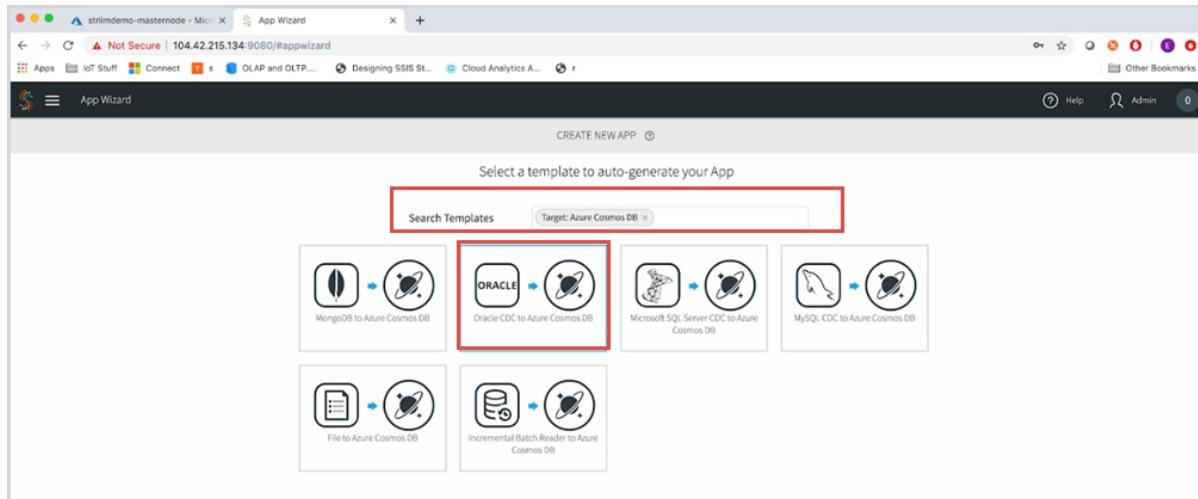
to learn about Striim, however in this article you will create our own. Select the **Add App** button in the top right-hand corner.



12. There are a few different ways to create Striim applications. Select **Start with Template** to start with an existing template.

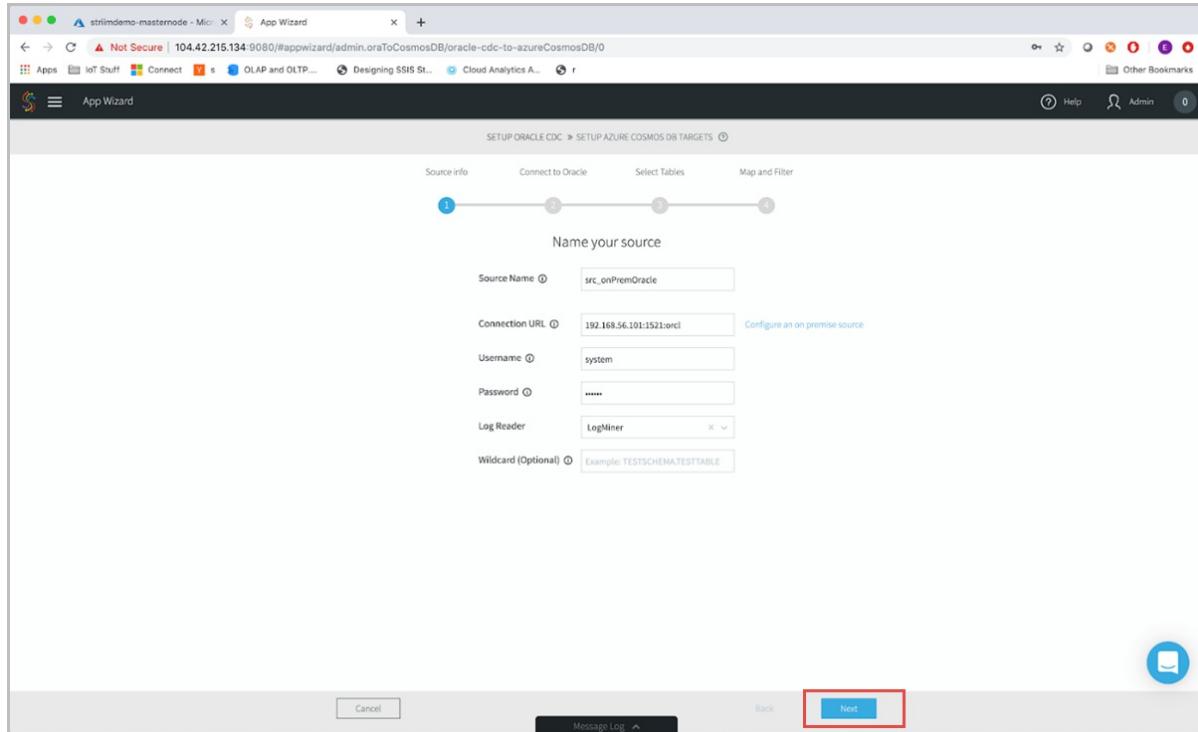


13. In the **Search templates** field, type "Cosmos" and select **Target: Azure Cosmos DB** and then select **Oracle CDC to Azure Cosmos DB**.

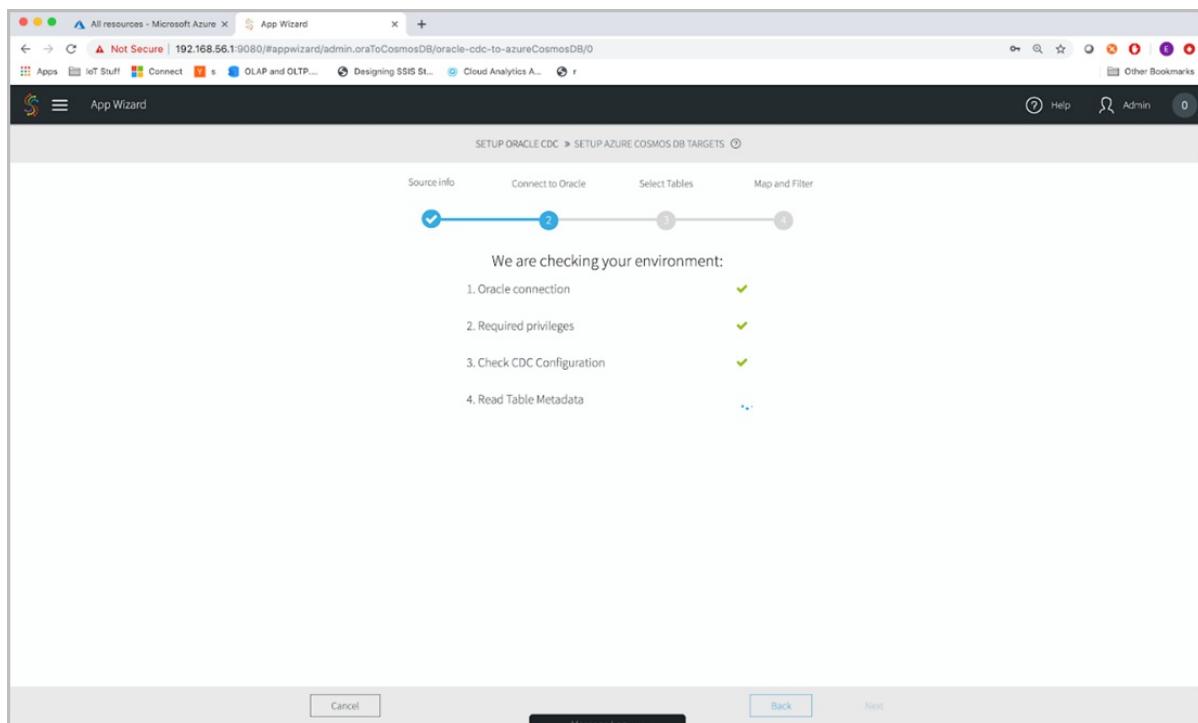


14. In the next page, name your application. You can provide a name such as **oraToCosmosDB** and then select **Save**.

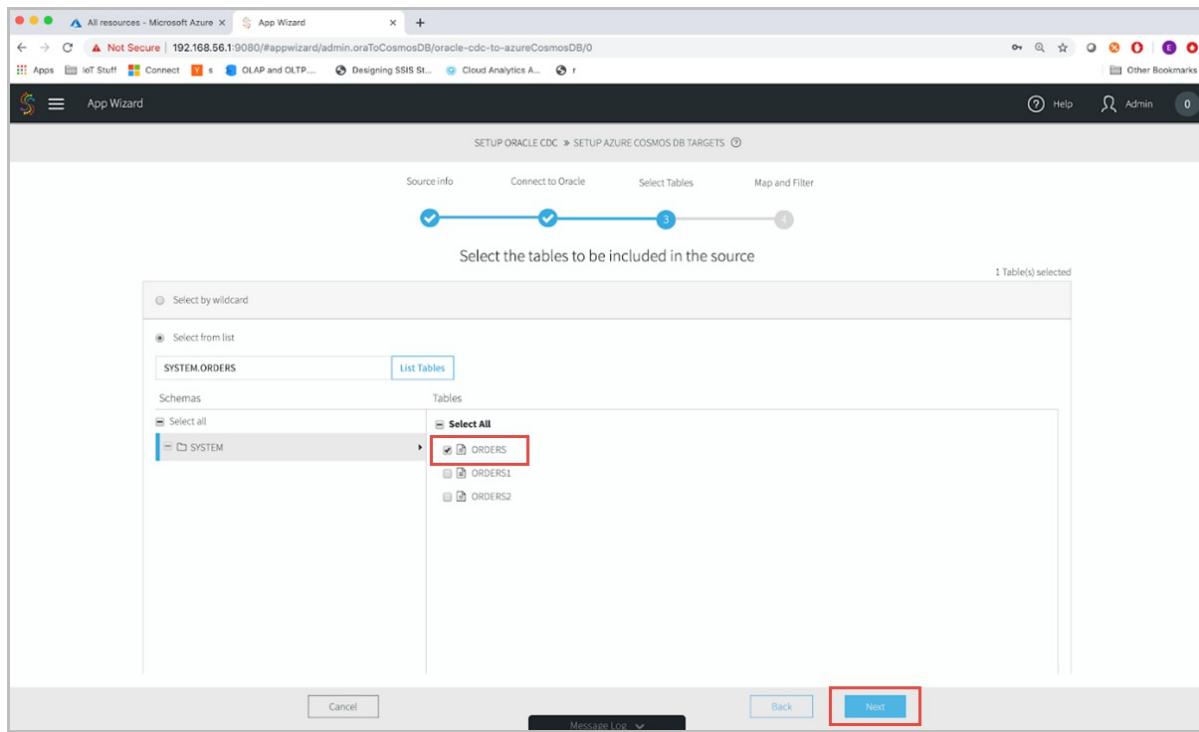
15. Next, enter the source configuration of your source Oracle instance. Enter a value for the **Source Name**. The source name is just a naming convention for the Striim application, you can use something like **src\_onPremOracle**. Enter values for rest of the source parameters **URL**, **Username**, **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**.

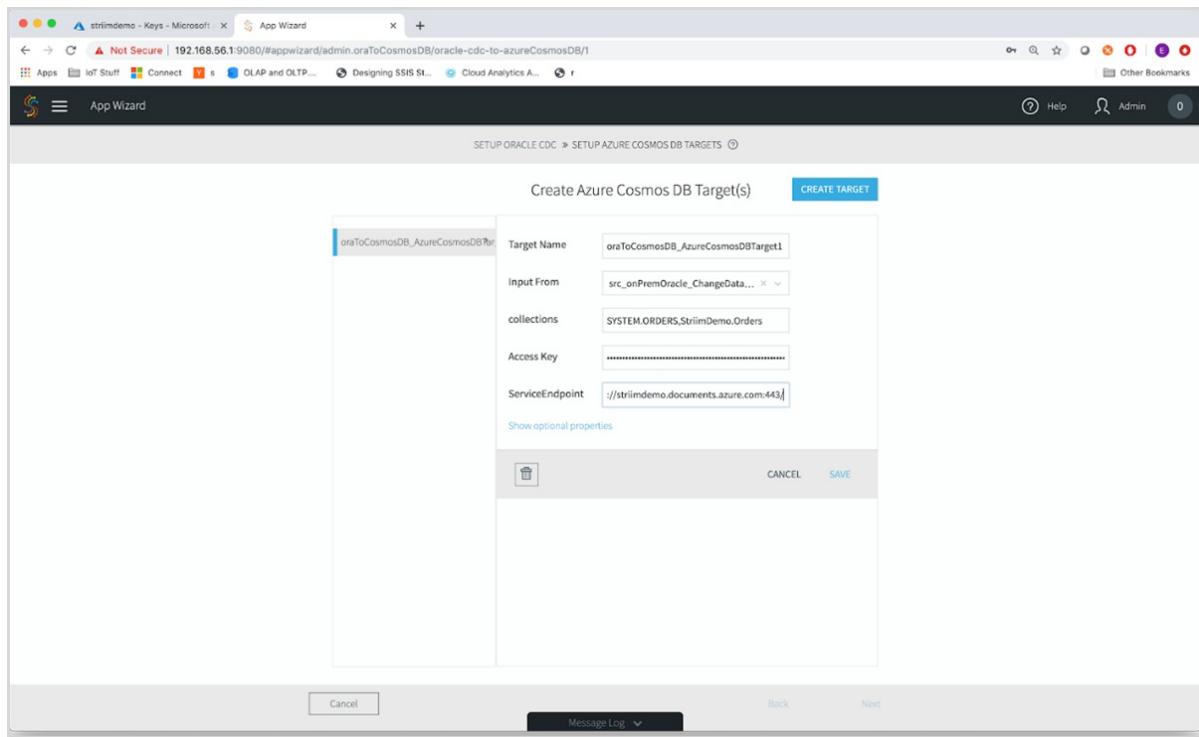


18. After selecting the source table, you can do more complicated operations such as mapping and filtering. In this case, you will just create a replica of your source table in Azure Cosmos DB. So, select **Next** to configure the target

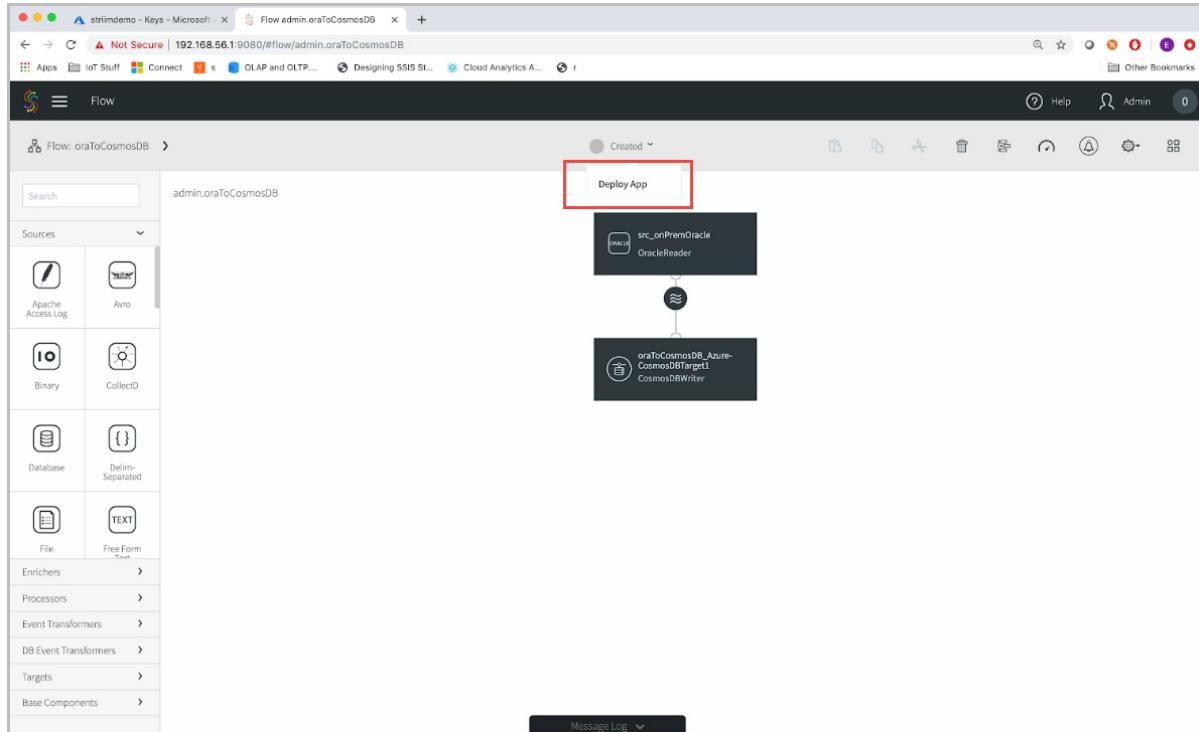
19. Now, let's configure the target:

- **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.

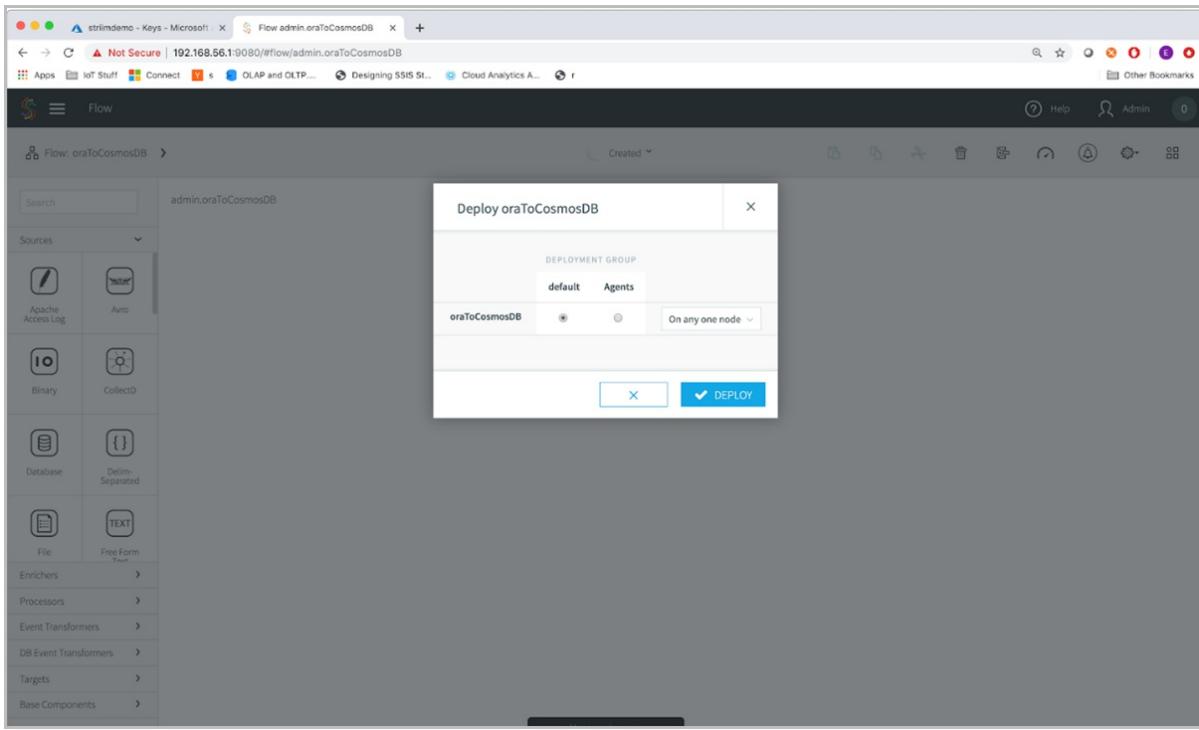
Select **Save** and **Next**.



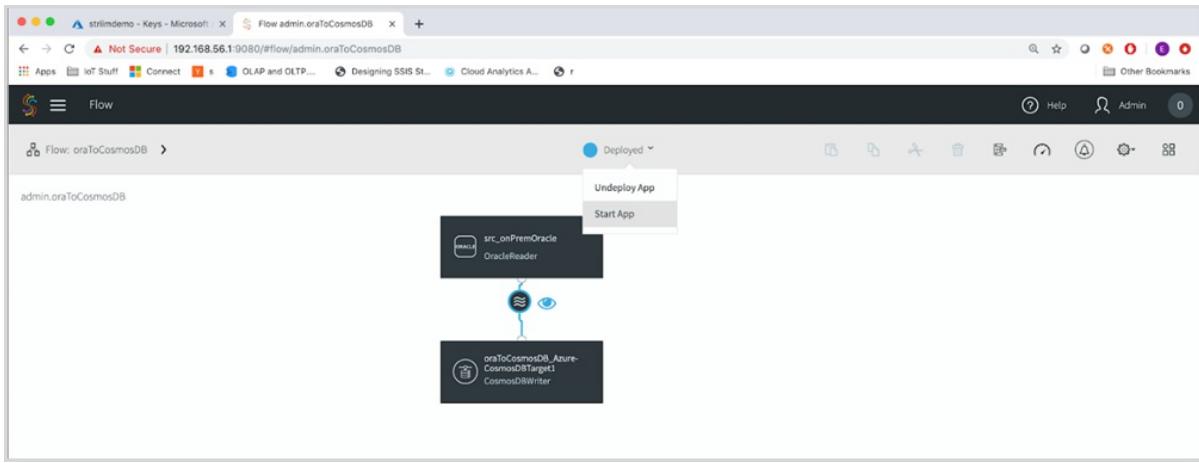
20. Next, you'll arrive at the flow designer, where you can drag and drop out of the box connectors to create your streaming applications. You will not make any modifications to the flow at this point, so go ahead and deploy the application by selecting the **Deploy App** button.



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.

```

SELECT * FROM c
    WHERE id = 1012
    /OR...
1  {
2      "id": "1012-156345372817",
3      "ORDER_ID": "1012",
4      "CUSTOMER_ID": "1012",
5      "ADDRESS": "1012",
6      "REP_ID": "1012",
7      "ORDER_MODE": "Online",
8      "ORDER_STATUS": "1",
9      "ORDER_TOTAL": "31943.94",
10     "PROMOTION_ID": "34411",
11     "id": "1012-156345372817:Online:1012:1:31943.94:176:34411",
12     "_rid": "ANBwAM-a2cBAAAAAAAA==",
13     "_self": "dbs/ANBwAM/colls/ANBwAM-a2c/docs/ANBwAM-a2cBAAAAAAAA==",
14     "_etag": "\\"73087797-0000-0000-5d393cd0000\\",
15     "_attachments": "attachments/",
16     "_ts": 1563464654
}

```

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 to Azure Cosmos DB Cassandra API account using Striim

9/25/2019 • 8 minutes to read • [Edit Online](#)

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.

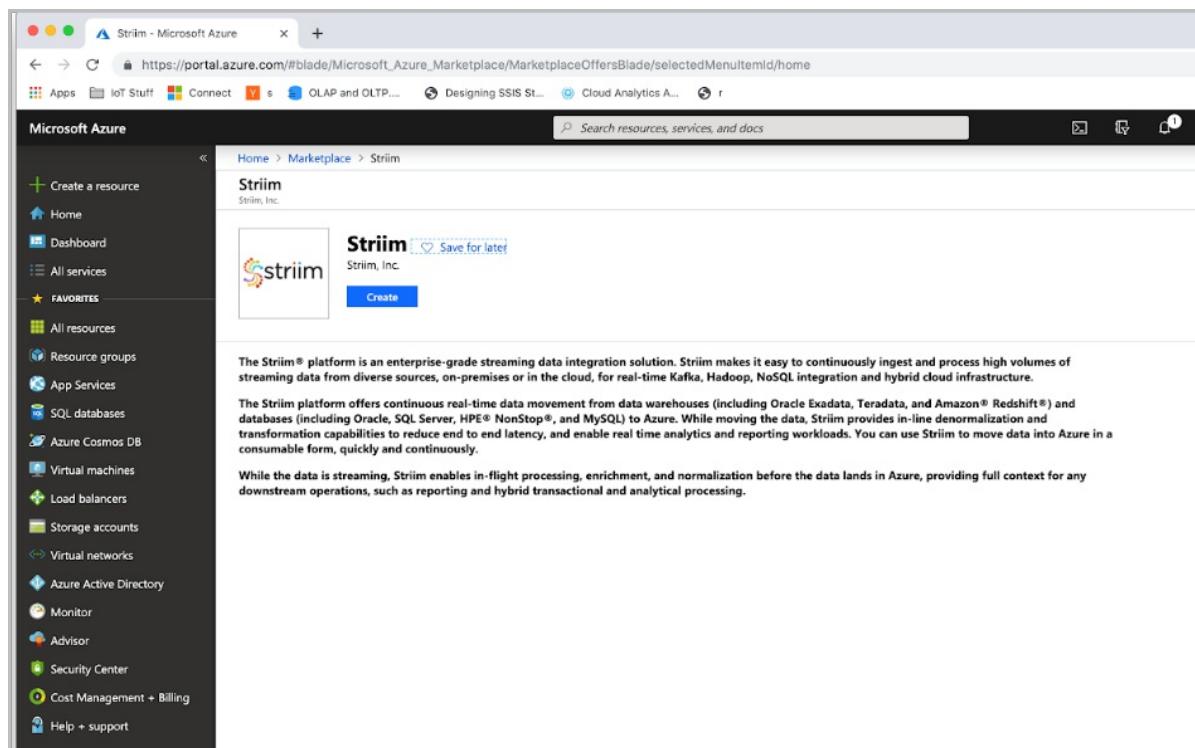
This article shows how to use Striim to migrate data from an **Oracle database** to an **Azure Cosmos DB Cassandra API account**.

## Prerequisites

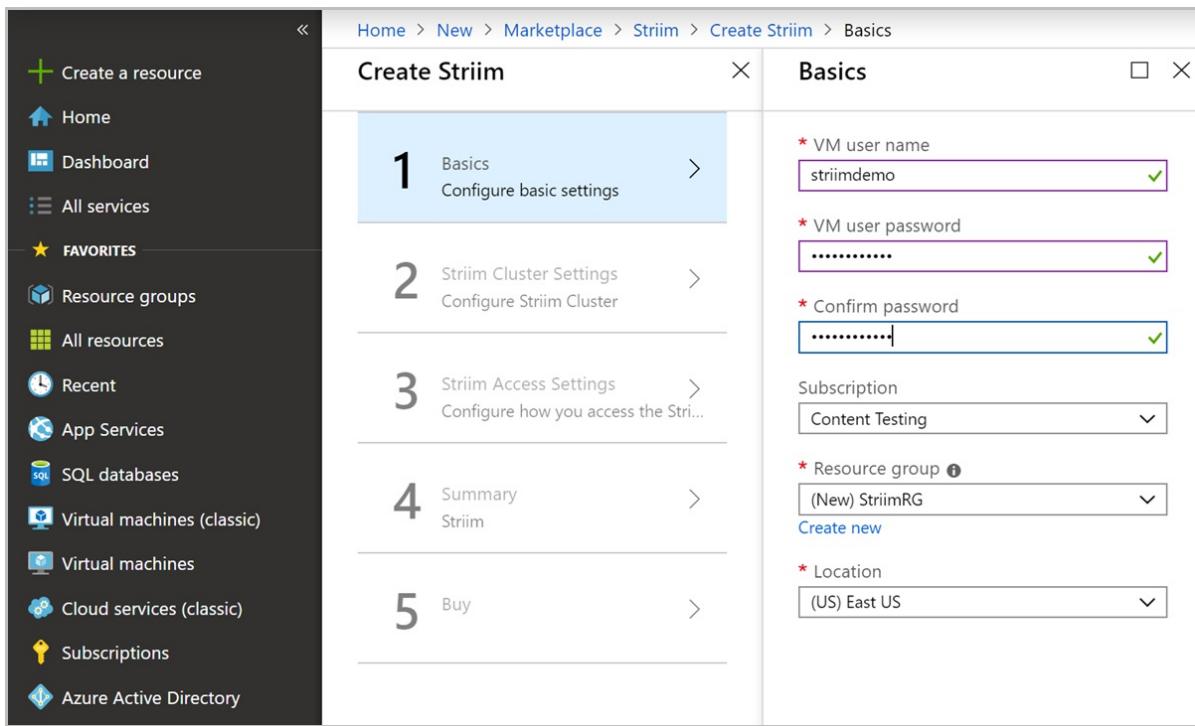
- If you don't have an [Azure subscription](#), create a [free account](#) before you begin.
- An Oracle database running on-premises with some data in it.

## 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 <b>Standalone</b> or <b>Cluster</b> 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.

The screenshot shows the Azure portal interface for creating a Striim instance. On the left, a sidebar lists various services like Home, Dashboard, All services, Favorites, and App Services. The main area displays a step-by-step wizard titled 'Create Striim'. Step 1: Basics (Done) is completed. Step 2: Striim Cluster Settings (Done) is completed. Step 3: Striim Access Settings (Configure how you access the Striim instance) is currently selected. Step 4: Summary (Striim) and Step 5: Buy are shown below. To the right of the wizard, a detailed configuration pane titled 'Striim Access Settings' is open. It includes fields for Public IP address, Domain name for Striim, Striim admin password, Confirm admin password, Virtual network, and Subnet. Most fields have green checkmarks indicating they are valid.

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.

The screenshot shows the Azure Cosmos DB Data Explorer interface. On the left, there's a sidebar with various options like Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Quick start, Notifications, and Data Explorer (which is selected). The main area is titled 'CASSANDRA API' and shows a large button labeled 'Welcome to Azure Cosmos DB'. To the right, a modal window titled 'Add Table' is open, prompting for a Keyspace name ('Create new') and a table definition. The table definition is:

```
CREATE TABLE Oracle.Orders
(
    ORDER_ID int,
    CUSTOMER_ID int,
    ORDER_DATE text,
    ORDER_MODE text,
    ORDER_STATUS int,
    ORDER_TOTAL float,
    PROMOTION_ID int,
    SALES_REP_ID int,
    PRIMARY KEY (ORDER_ID)
)
```

Below the table definition, it says 'Throughput (400 - 1,000,000 RU/s)' set to 1000, and 'Estimated spend (USD): \$0.080 hourly / \$1.92 daily (1 region, 1000RU/s, \$0.00008/RU)'. At the bottom of the modal is an 'OK' button.

## Configure Oracle to Azure Cosmos DB data flow

- Now, let's get back to Striim. Before interacting with Striim, install the Oracle JDBC driver that you downloaded earlier.
- 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.

The screenshot shows the Azure portal's 'All resources' view with a selected virtual machine named 'strimdemo-masternode'. The 'Overview' section is highlighted with a red box. On the right, a modal window titled 'Connect to virtual machine' is open for the 'SSH' tab. It contains fields for 'IP address' (set to 'DNS name strimdemo.westus.cloudapp.azure.com') and 'Port number' (set to '22'). Below these fields is a 'Login using VM local account' field containing the URL 'ssh strimdemo@strimdemo.westus.cloudapp.azure.com'. At the bottom of the modal, there's a link 'Having trouble connecting to this VM?' followed by three bullet points: 'Diagnose and solve problems', 'Troubleshoot connection', and 'Serial console'.

- 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 — striimd@striimdemo-masternode:~ — ssh striimd@striimdemo.westus.cloudapp.azure.com
[Striim-Edwards-MacBook-Pro:Striim edwardbell$ ssh striimd@striimdemo.westus.cloudapp.azure.com
>Password:
Last login: Tue Jul 16 21:04:40 2019 from 208.89.164.26
[striimd@striimdemo-masternode ~]$ ]$
```

- 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
```

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  
mvojdbc8.jar /opt/striim/lib  
chmod +xojdbc8.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
```

- Now, navigate back to Azure and copy the Public IP address of your Striim VM.

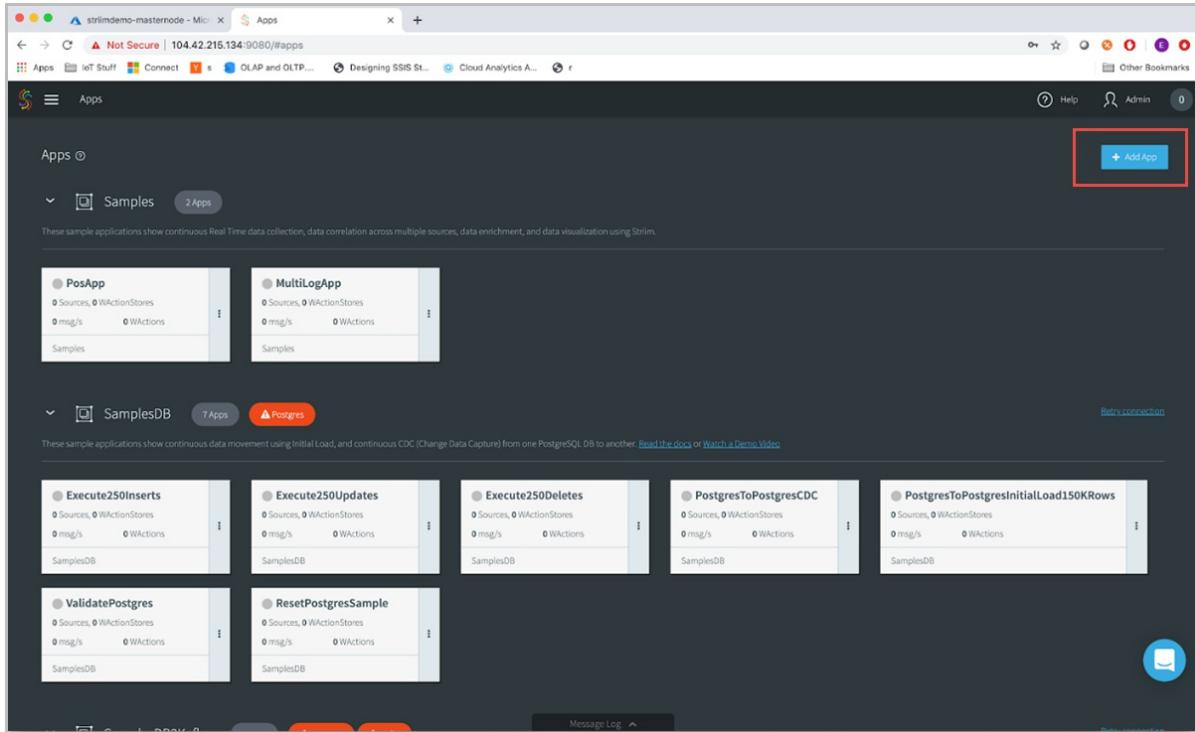
The screenshot shows the Azure portal's 'All resources' view for the 'striimdemo-masternode' virtual machine. On the right, the 'Overview' tab is selected, displaying basic information like Resource group (striimazuredemo), Status (Running), Location (West US), and Subscription (Visual Studio Enterprise: BizSpark). The 'Public IP address' field is highlighted with a red box and contains the value '104.42.215.134'. Below the main info, there are four performance charts: CPU (average), Network (total), Disk bytes (total), and Disk operations/sec (average). The CPU chart shows a peak of 1.58% at 2:15 PM. The Network chart shows Network In Total (Sum) at 57.5 ms and Network Out Total (Sum) at 11.5 ms. The Disk bytes chart shows Disk Read Bytes (Sum) at 5.86 ms and Disk Write Bytes (Sum) at 453.06 MB.

- 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.

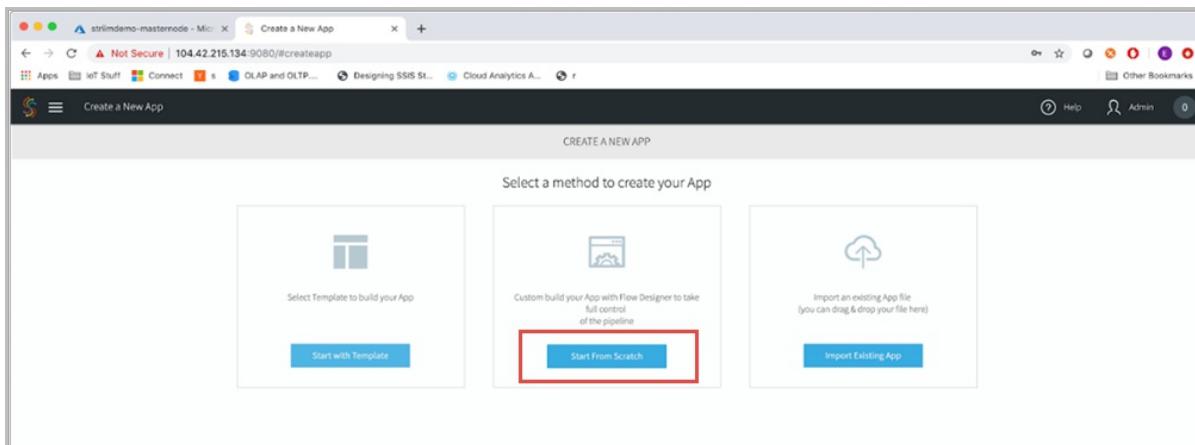
The screenshot shows a web browser window with the URL '104.42.215.134:9080'. The page displays the Striim logo and a login form with fields for 'admin' (username) and 'Password' (password). A blue 'LOG IN' button is at the bottom of the form.

- 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.
- 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

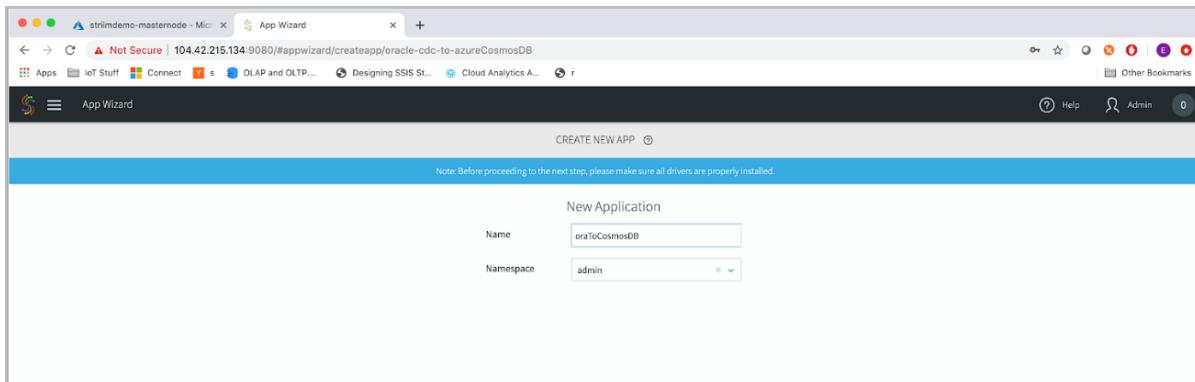
right-hand corner.



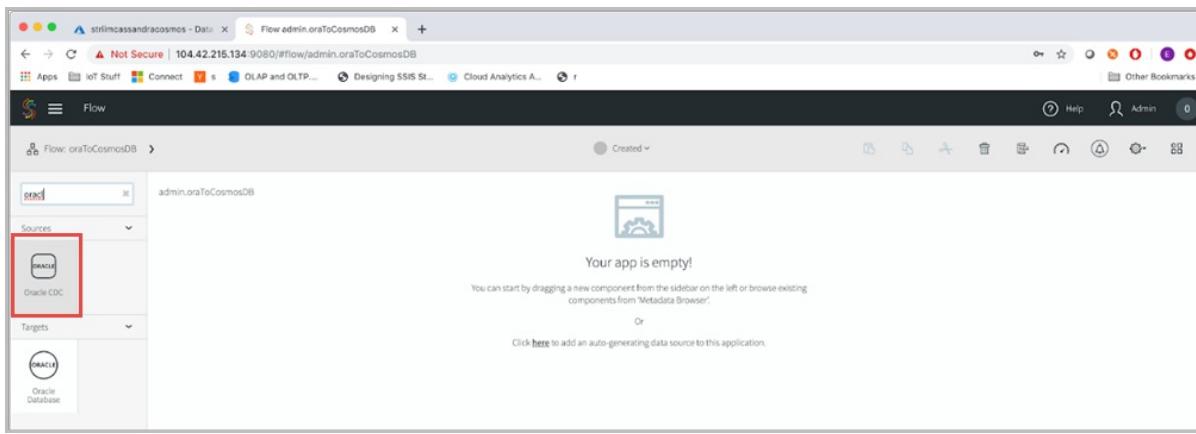
12. There are a few different ways to create Striim applications. Select **Start from Scratch** for this scenario.



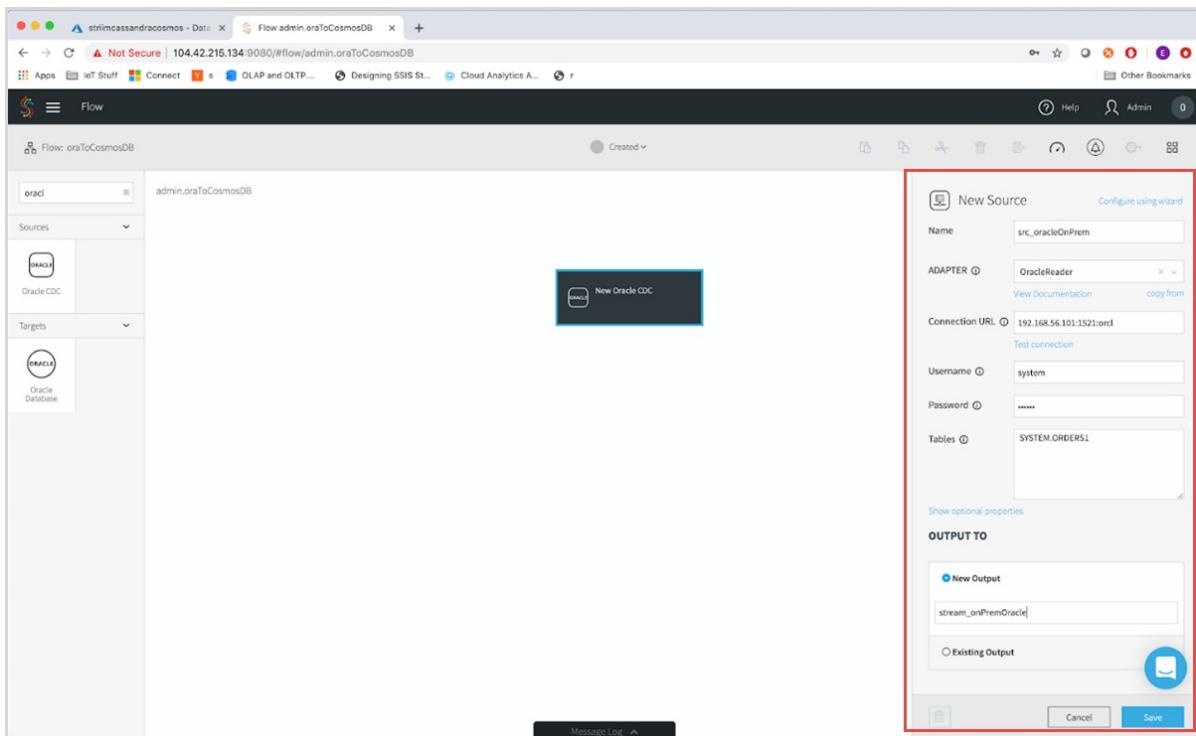
13. Give a friendly name for your application, something like **oraToCosmosDB** and select **Save**.



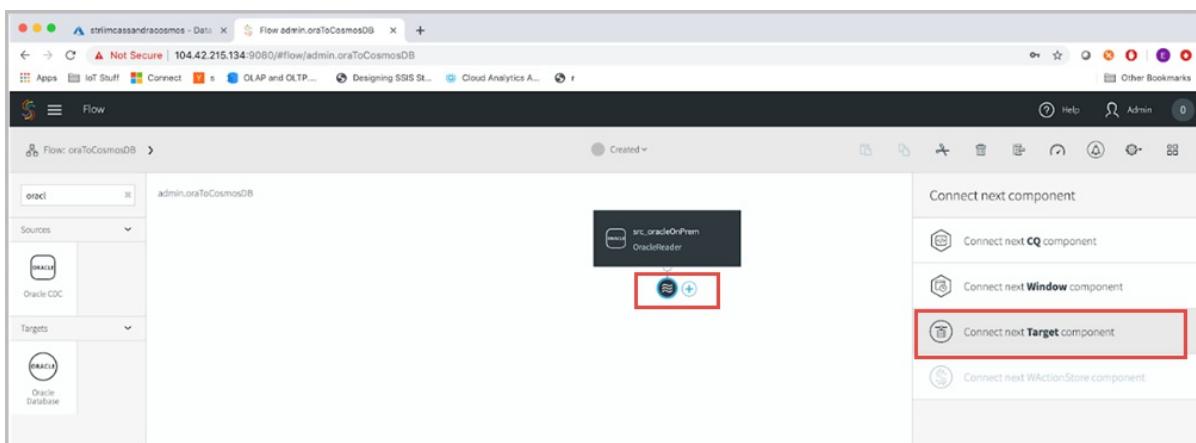
14. You'll arrive at the Flow Designer, where you can drag and drop out of the box connectors to create your streaming applications. Type **Oracle** in the search bar, drag and drop the **Oracle CDC** source onto the app canvas.



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.



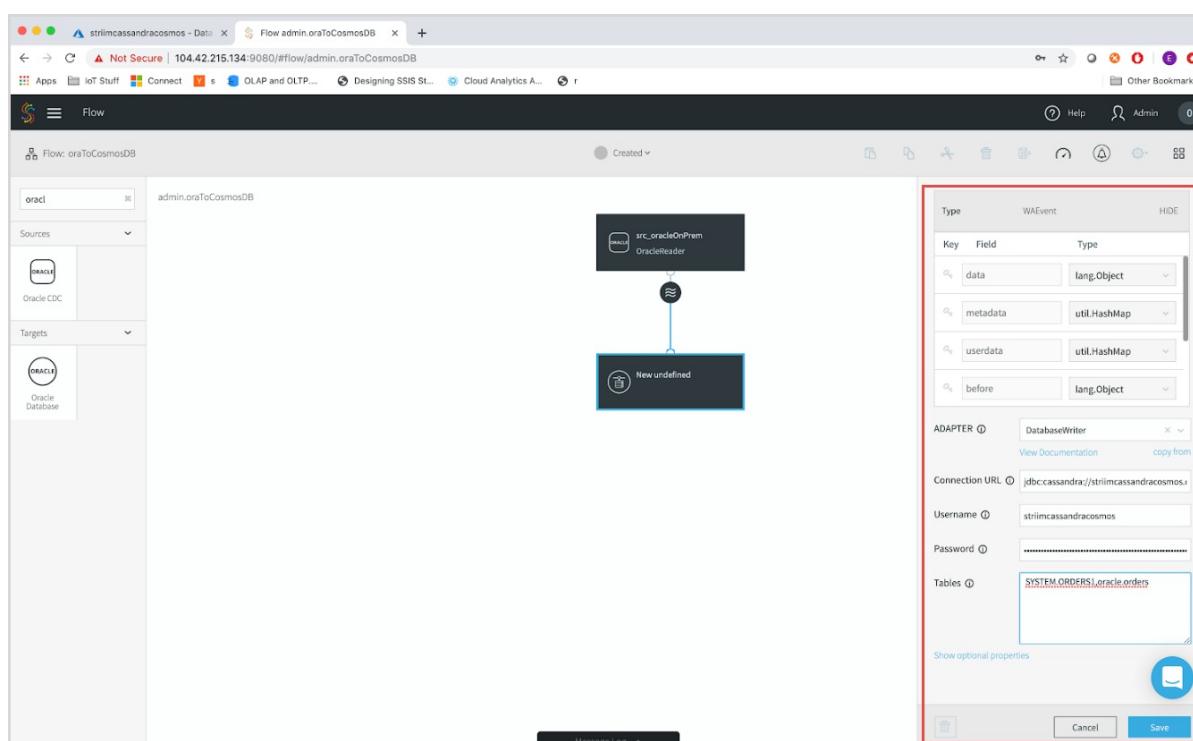
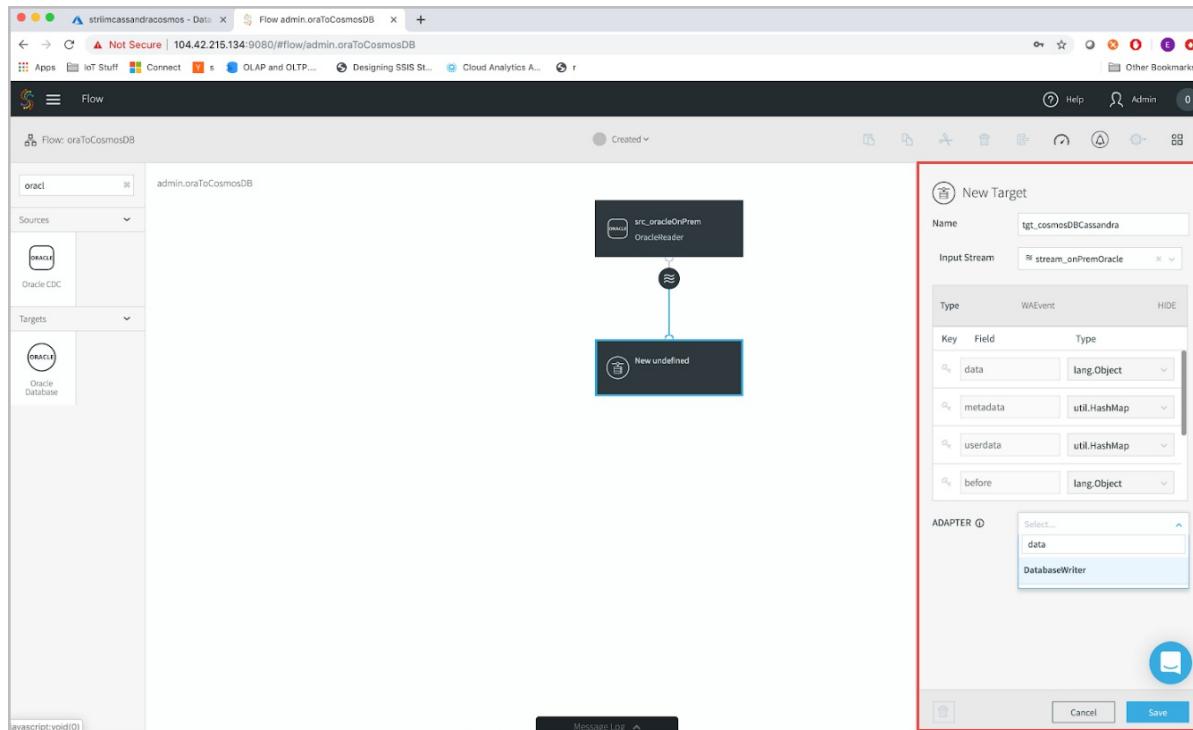
16. Now, click the wave icon of the stream to connect the target Azure Cosmos DB instance.



17. Before configuring the target, make sure you have added a [Baltimore root certificate to Striim's Java environment](#).

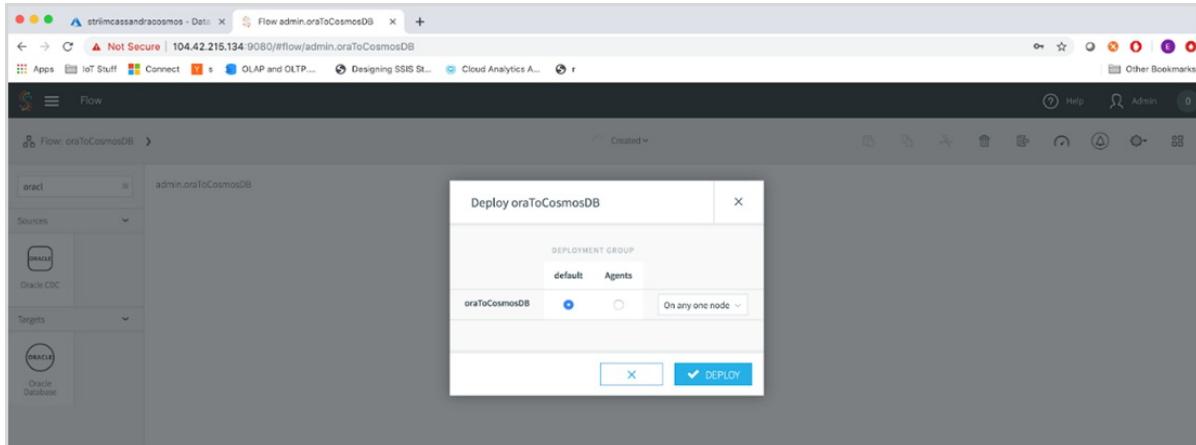
18. Enter the configuration properties of your target Azure Cosmos DB instance and select **Save** to continue. Here are the key parameters to note:

- **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.

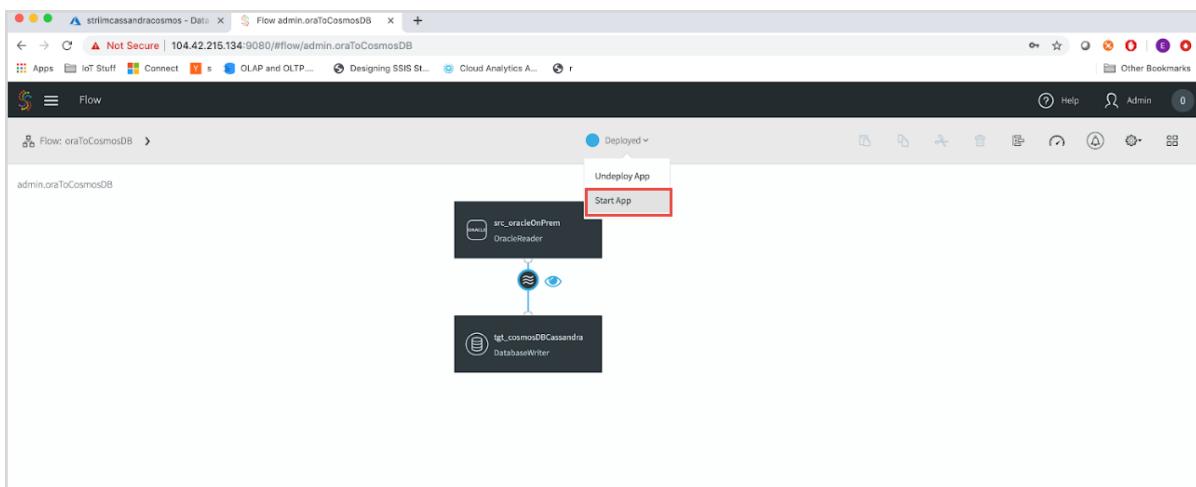


19. Now, we'll go ahead and run the Striim application. In the upper menu bar select **Created**, and then **Deploy App**. In the deployment window you can specify if you want to run certain parts of your application on specific parts of your deployment topology. Since we're running in a simple deployment topology

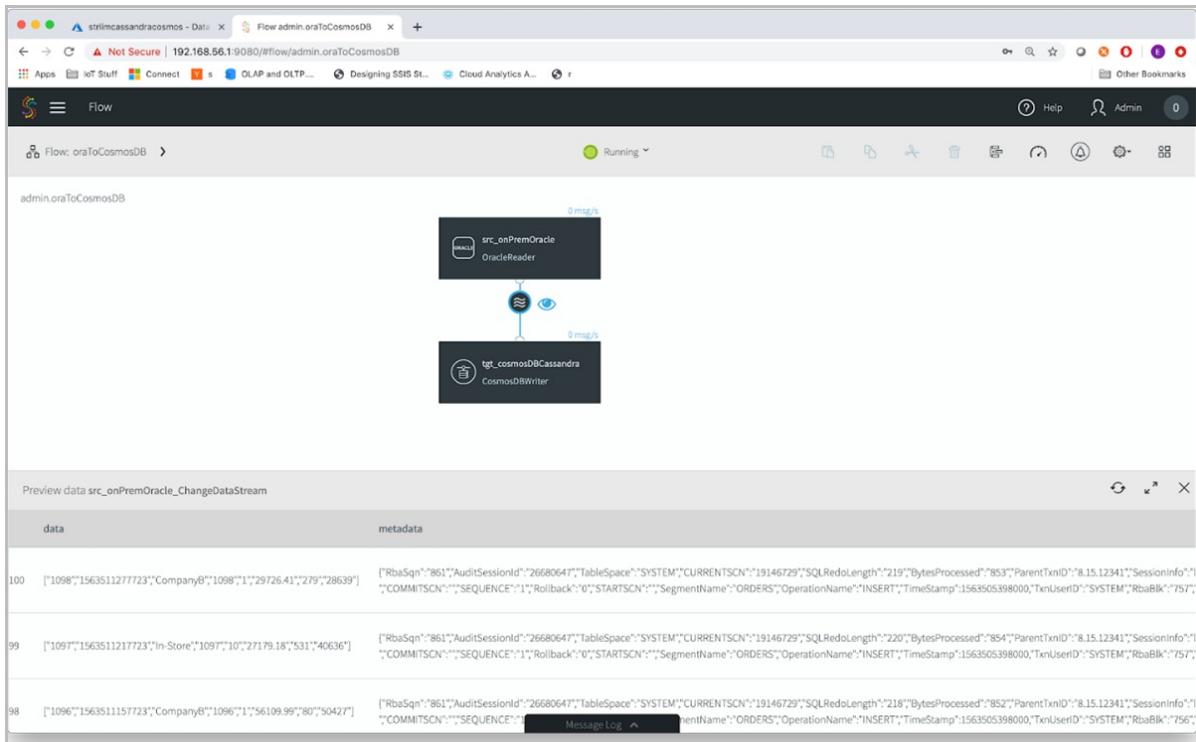
through Azure, we'll use the default option.



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

8/22/2019 • 5 minutes to read • [Edit Online](#)

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 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.

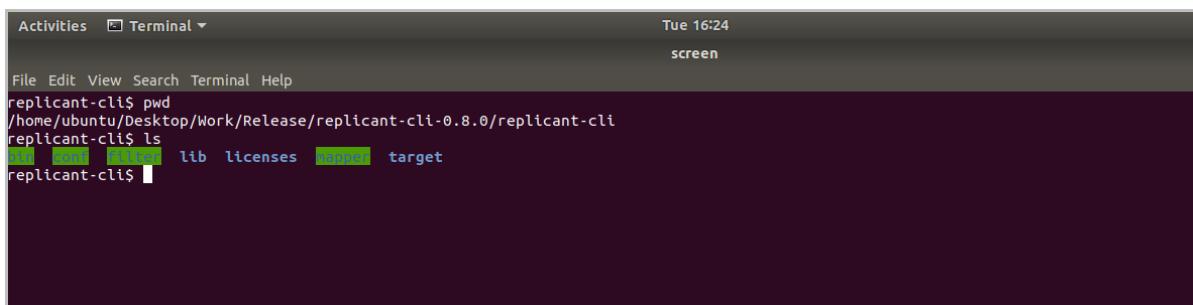
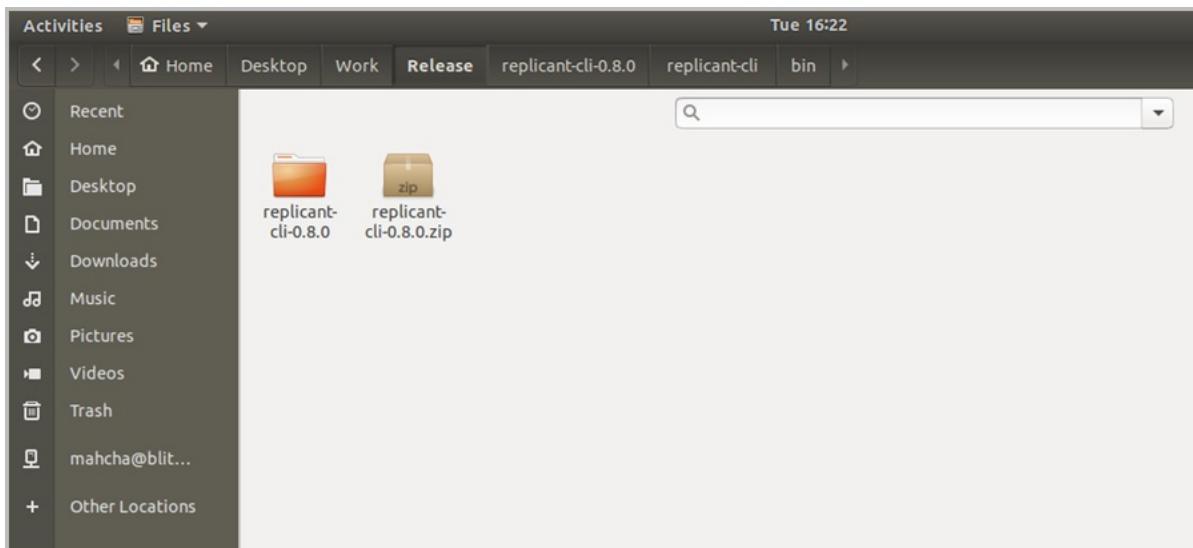
1. From the computer where you plan to install the Blitzz replicant, add a security certificate. This certificate is required by the Blitzz replicant to establish an SSL connection with the specified Azure Cosmos DB account. You can add the certificate with the following steps:

```

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 Blitz installation and the binary files either by requesting a demo on the [Blitz 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

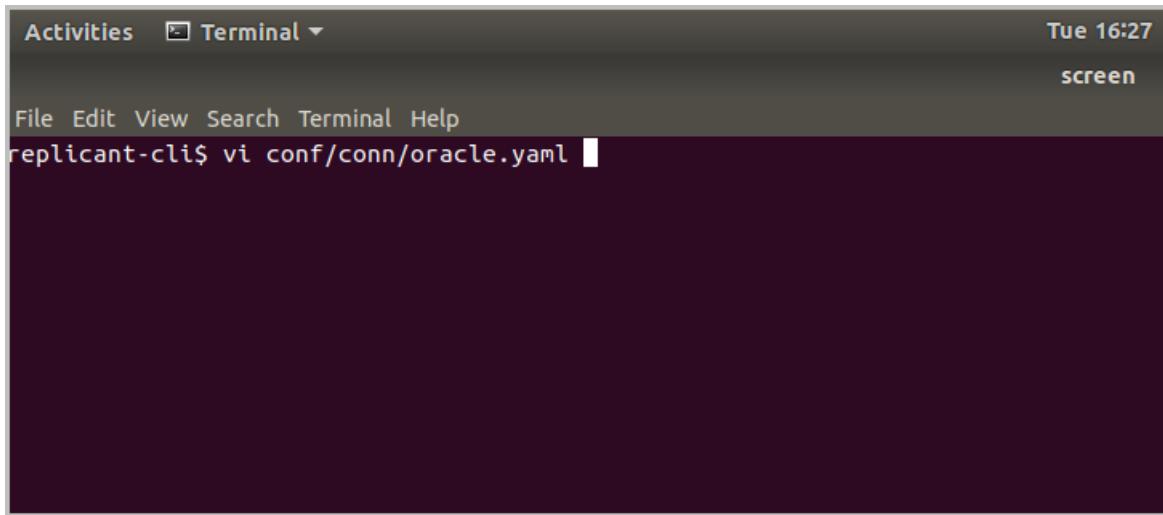
host: localhost
port: 53546

service-name: IO

username: '<Username of your Oracle database>'
password: '<Password of your Oracle database>'

conn-cnt: 30
use-ssl: false

```

A screenshot of a terminal window titled 'screen'. The window displays a configuration file with the following content:

```
File Edit View Search Terminal Help
type: ORACLE

host: localhost
port: 53546

service-name: IO

username: 'replicant'
password: 'Replicant#123'

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_blitz"
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.yaml` 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: 'bltzzdemo'
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.

screen

File Edit View Search Terminal Help

---

Oracle --> CosmosDB

---

Progress: ======100.00%=====

	Elapsed time: 00:13:51	Initial load time: 00:01:36	Rate: 0.00			
Table name	Rows	Size	Inserted	Deleted	Updated	Rate
partsupp	8000	1176000	8000	0	0	0.00
orders	15000	1425000	15000	0	0	0.00
lineitem	60175	8966075	60175	0	0	0.00
nation	25	2600	25	0	0	0.00
region	5	480	10	5	5	0.00
customer	1500	189000	1500	0	0	0.00
part	2000	192000	2000	0	0	0.00
supplier	100	11300	100	0	0	0.00

---

- 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.
- 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 [Blitz 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

8/22/2019 • 5 minutes to read • [Edit Online](#)

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 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 SSL, encryption.

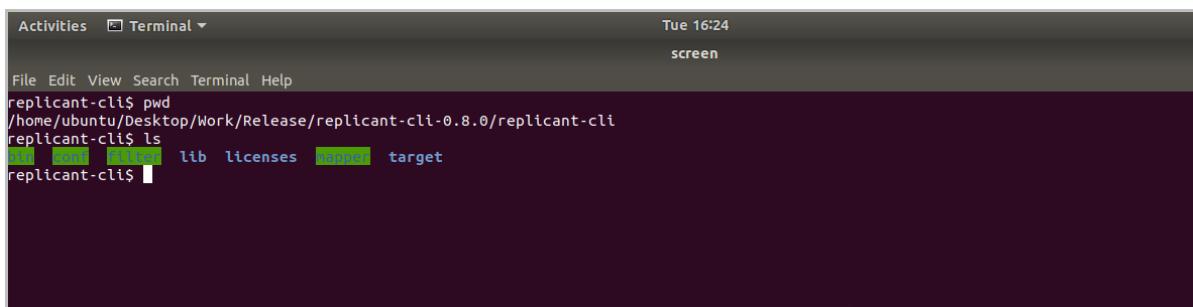
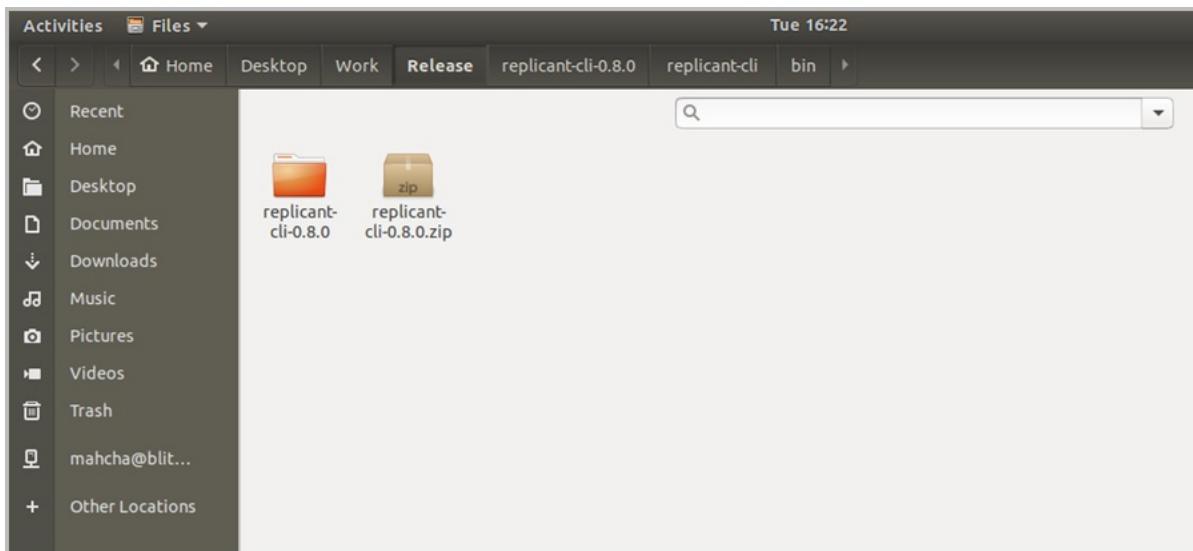
## Steps to migrate data

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 SSL 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 Blitz installation and the binary files either by requesting a demo on the [Blitz website](#). Alternatively, you can also send an [email](#) to the team.



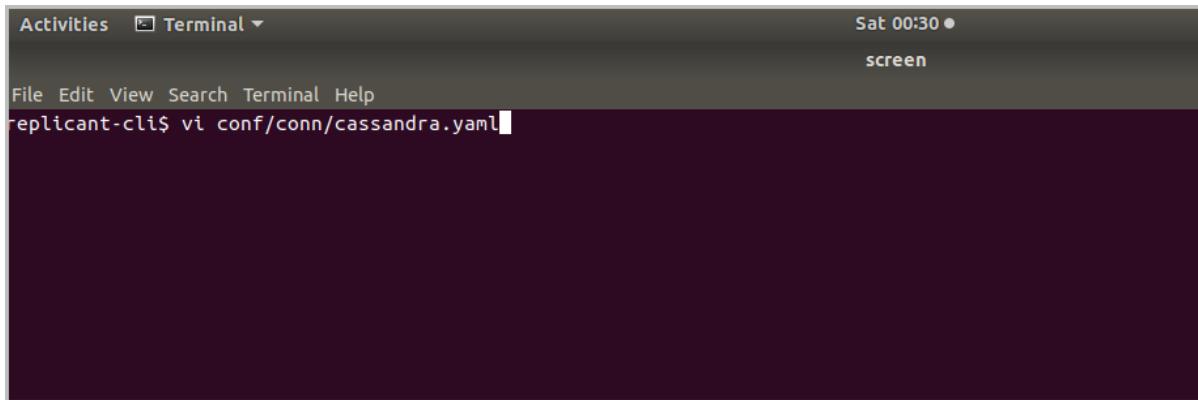
3. From the CLI terminal, set up the source database configuration. Open the configuration file using `vi conf/conn/cassandra.yml` command and add a comma-separated list of IP addresses of the Cassandra nodes, port number, username, password, and any other required details. The following is an example of contents in the configuration file:

```
type: CASSANDRA

host: 172.17.0.2
port: 9042

username: 'cassandra'
password: 'cassandra'

max-connections: 30
```

A screenshot of a terminal window titled "nileshgohad@nileshgohad-ThinkPad-T480s<Blitzz Config>". The file "cassandra.yaml" is open and contains the following configuration:

```
File Edit View Search Terminal Help
type: CASSANDRA
host: 172.17.0.2
port: 9042
username: 'cassandra'
password: 'cassandra'
max-connections: 30
```

The "type" field is set to "CASSANDRA", "host" is "172.17.0.2", "port" is "9042", "username" is "cassandra", "password" is "cassandra", and "max-connections" is "30". There are several blank lines below these settings.

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.yaml` 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.

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 [Blitz 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 hundreds of terabytes of data into Azure Cosmos DB

10/24/2019 • 9 minutes to read • [Edit Online](#)

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.

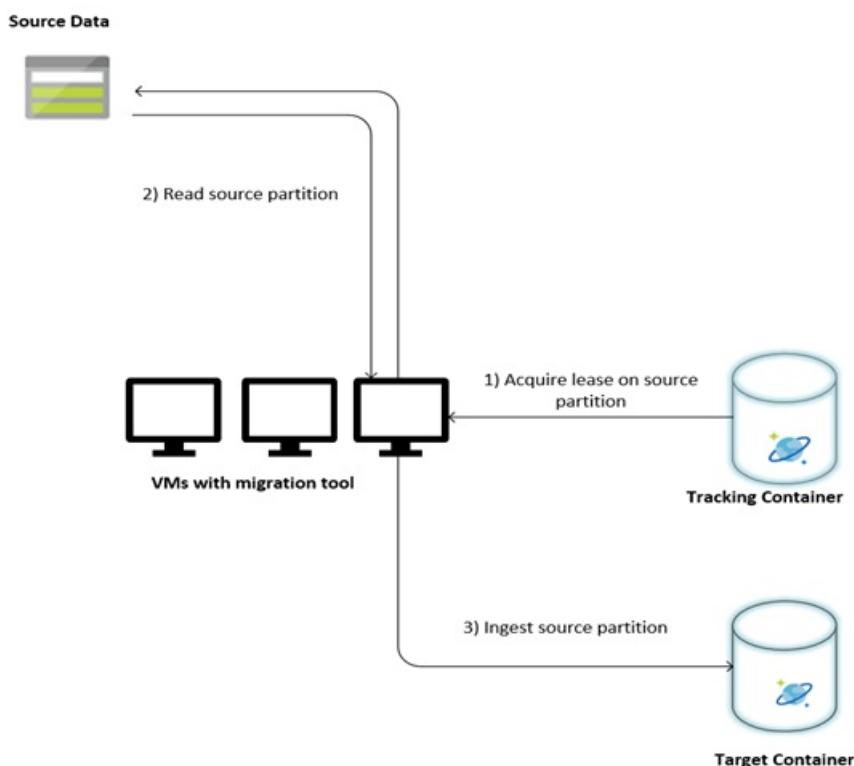
## Custom tool with bulk executor library

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": 3950252.280000003,
    "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 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.

## Contact the Azure Cosmos DB team

Although you can follow this guide to successfully migrate large datasets to Azure Cosmos DB, for large scale migrations, it is recommended that you reach out the Azure Cosmos DB product team to validate the data modelling and a general architecture review. Based on your dataset and workload, the product team can also suggest other performance and cost optimizations that could be applicable to you. To contact the Azure Cosmos DB team for assistance with large scale migrations, you can open a support ticket under the "General Advisory" problem type and "Large (TB+) migrations" problem subtype as shown below.

[Basics](#)[Solutions](#)[Details](#)[Review + create](#)

Create a new support request to get assistance with billing, subscription, technical (including advisory) or quota management issues.

Complete the Basics tab by selecting the options that best describe your problem. Providing detailed, accurate information can help to solve your issues faster.

* Issue type	Technical
* Subscription	Microsoft Azure Internal Consumption (ac899844-6480-47...)
Can't find your subscription? <a href="#">Show more</a>	
* Service	<input checked="" type="radio"/> My services <input type="radio"/> All services
Cosmos DB	
* Resource	<input type="text"/> migration-test-sit
I want to migrate 50 terabytes of data into Cosmos DB	<input checked="" type="checkbox"/>
* Problem type	General Advisory
Large (TB+) migrations	

## 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 one-to-few relational data into Azure Cosmos DB SQL API account

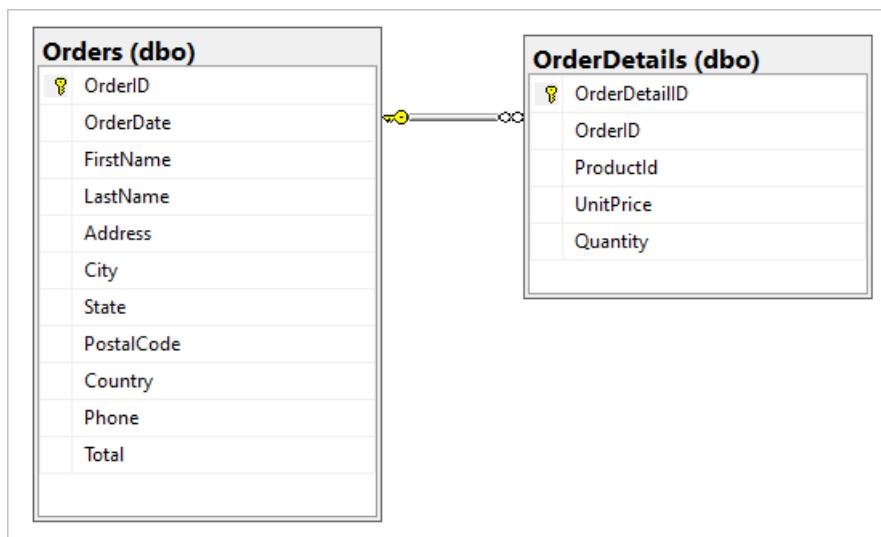
1/10/2020 • 8 minutes to read • [Edit Online](#)

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:

```

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;

```

00 %

OrderID	OrderDate	La...	Add...	C...	S...	I C...	P...	Total	OrderDetails
1	2019-12...	D...	123...	C.	IL	U...	5...	10.0000	[{"OrderDetailId":10,"ProductId":200,"UnitPrice":3.5000,"Quantity":2}, {"OrderDetailId":11,"ProductId":201,"UnitPrice":3.0000,"Quantity":1}]
2	2019-12...	D...	456...	C.	IL	U...	5...	100.0000	[{"OrderDetailId":12,"ProductId":200,"UnitPrice":3.5000,"Quantity":2}, {"OrderDetailId":13,"ProductId":202,"UnitPrice":5.0000,"Quantity":15}, {"OrderDetailId":14,"Pr...
3	2019-12...	D...	789...	C.	IL	U...	5...	50.0000	[{"OrderDetailId":15,"ProductId":201,"UnitPrice":3.0000,"Quantity":10}, {"OrderDetailId":16,"ProductId":202,"UnitPrice":5.0000,"Quantity":1}, {"OrderDetailId":17,"Pr...

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)
    )

```

Linked service: AzureSqlDatabase1

**value**

```
{"id":1000,"OrderDate":"2019-12-10T00:00:00","FirstName":"John","LastName":"Doe","Address":"123 W Main St","City":"Cityville","State":"IL","PostalCode":"60001","Country":"United States","Phone":"555-555-5555","Total":10.0000,"OrderDetails":[{"OrderDetailId":10,"ProductId":200,"UnitPrice":3.5000,"Quantity":2}, {"OrderDetailId":11,"ProductId":201,"UnitPrice":3.0000,"Quantity":1}]}
```

```
{"id":1001,"OrderDate":"2019-12-11T00:00:00","FirstName":"Mary","LastName":"Doe","Address":"456 W Main St","City":"Cityville","State":"IL","PostalCode":"60001","Country":"United States","Phone":"555-555-5551","Total":100.0000,"OrderDetails":[{"OrderDetailId":12,"ProductId":200,"UnitPrice":3.5000,"Quantity":2}, {"OrderDetailId":13,"ProductId":202,"UnitPrice":5.0000,"Quantity":1}, {"OrderDetailId":14,"ProductId":203,"UnitPrice":9.0000,"Quantity":2}]}  
{"id":1002,"OrderDate":"2019-12-12T00:00:00","FirstName":"Mike","LastName":"Doe","Address":"789 W Main St","City":"Cityville","State":"IL","PostalCode":"60002","Country":"United States","Phone":"555-555-5552","Total":50.0000,"OrderDetails":[{"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}, {"OrderDetailId":20,"ProductId":206,"UnitPrice":1.0000,"Quantity":2}]}  
Each Order is one JSON document
```

**Copy data** ✓

SqlJsonToBlobText

Open New

Query \*

```
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) as
        OrderDetails
    FROM [YourTable]
)
```

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".

The screenshot shows the 'Connection' tab of the Copy Activity configuration page. The 'Linked service' dropdown is set to 'AzureBlobStorage1'. The 'File path' field contains 'storedatabase / Orders / @concat(pipeline().RunId,'.json')'. The 'Column delimiter' is set to 'Tab (\t)', and the 'Quote character' is set to 'No quote character'. Both of these settings are highlighted with red boxes. Other settings include 'Compression type: none', 'Row delimiter: Auto detect (\r,\n, or \r\n)', 'Encoding: Default(UTF-8)', 'Escape character: Backslash (\')', and 'First row as header: unchecked'. The 'Null value' field is empty.

### 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.

General Connection Schema Parameters

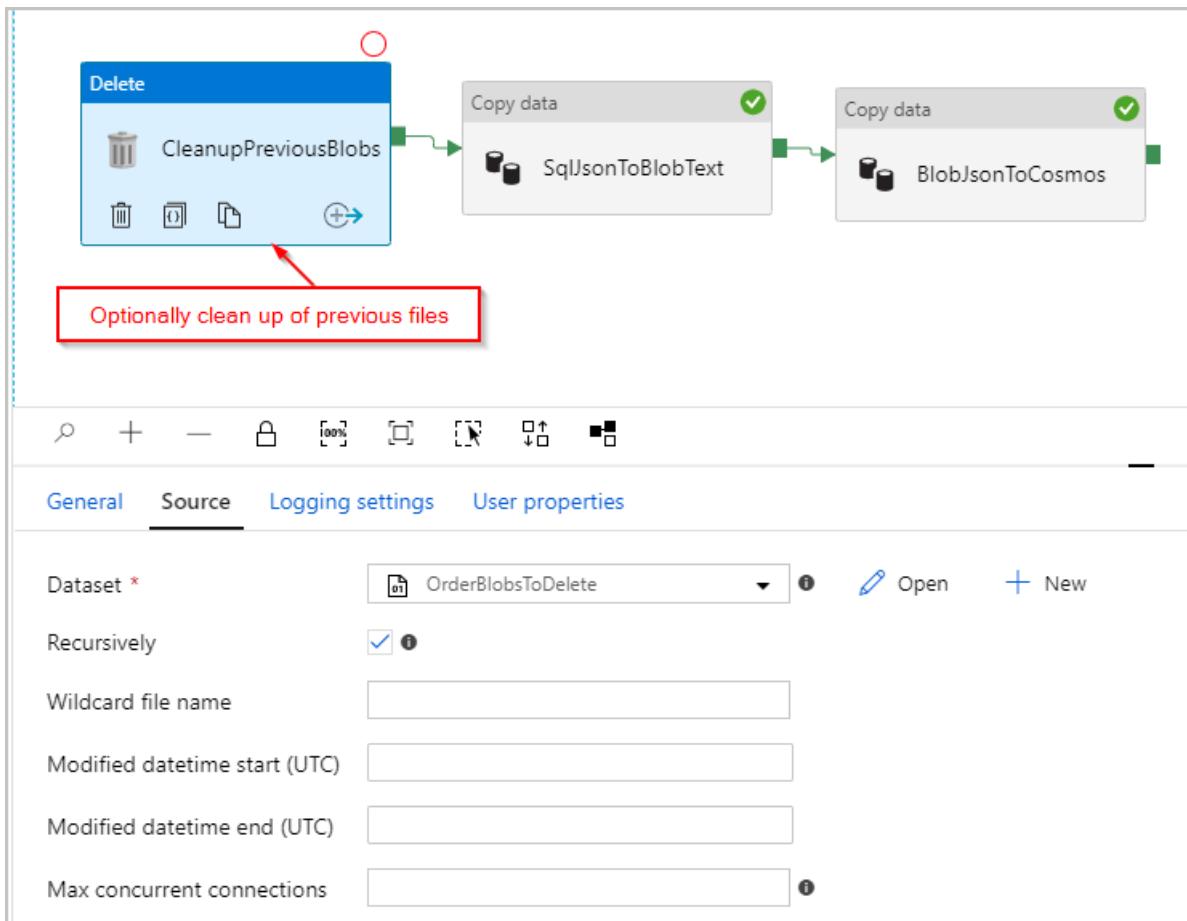
Linked service \* AzureBlobStorage1 Test connection Open New

File path \* storedatabase / Orders / @concat(pipeline().RunId,'.json')

Compression type none

Encoding Default(UTF-8)

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:

The screenshot shows the Azure Storage Explorer interface. A red box highlights the file name "d1706e71-a7a8-4796-80d3-77d6b36fa837.json". Below it, a red arrow points from the file name to the JSON content in the main pane. The JSON content is as follows:

```

1  [{"id":1000,"OrderDate":"2019-12-10T00:00:00","FirstName":"John","LastName":"Doe","Address":"123 W Main St", "City": "Anytown", "State": "IL", "PostalCode": "60001", "Country": "United States", "Phone": "555-555-5551", "Total": 100}, {"id":1001,"OrderDate":"2019-12-11T00:00:00","FirstName":"Mary","LastName":"Doe","Address":"456 W Main St", "City": "Anytown", "State": "IL", "PostalCode": "60001", "Country": "United States", "Phone": "555-555-5552", "Total": 100}, {"id":1002,"OrderDate":"2019-12-12T00:00:00","FirstName":"Mike","LastName":"Doe","Address":"789 W Main St", "City": "Anytown", "State": "IL", "PostalCode": "60001", "Country": "United States", "Phone": "555-555-5553", "Total": 100}]

```

We also see Orders documents with properly embedded OrderDetails inserted into our Cosmos DB collection:

The screenshot shows the Azure portal's SQL API blade. The left sidebar shows the database structure: StoreDatabase > Orders > Items. A red box highlights the item with id 1001. The right pane displays the JSON representation of this item. A red box highlights the nested "OrderDetails" array, and a red arrow points from this box to a callout text: "Nested JSON array stored properly as part of the CosmosDB document". The JSON content is as follows:

```

1  {
2    "id": "1001",
3    "OrderDate": "2019-12-11T00:00:00",
4    "FirstName": "Mary",
5    "LastName": "Doe",
6    "Address": "456 W Main St",
7    "City": "Anytown",
8    "State": "IL",
9    "PostalCode": "60001",
10   "Country": "United States",
11   "Phone": "555-555-5551",
12   "Total": 100,
13   "OrderDetails": [
14     {
15       "OrderDetailId": 12,
16       "productId": 200,
17       "UnitPrice": 3.5,
18       "Quantity": 2
19     },
20     {
21       "OrderDetailId": 13,
22       "productId": 202,
23       "UnitPrice": 5,
24       "Quantity": 15
25     },
26     {
27       "OrderDetailId": 14,
28       "productId": 203,
29       "UnitPrice": 9,
30       "Quantity": 2
31     }
32   ],
33   "_rid": "tJN4AMgz6mICAAAAAAA==",
34   "_self": "dbs/tJN4AA==/colls/tJN4AMgz6mI=/docs/tJN4AMgz6mICAAAAAAA==/",
35   "_etag": "\"77064c74-0000-0200-0000-5defd60e0000\"",
36   "_attachments": "attachments/",
37   "_ts": 1575998990
38 }

```

## 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.

The screenshot shows the Databricks Spark Cluster UI for cluster1. The 'Libraries' tab is selected and highlighted with a red box. The table lists two libraries:

Name	Type	Status	Source
azuresqlspark	JAR	Installed	dbfs/FileStore/jars/8f98cada_14ab_4adc_9ff3_a53bf2911987-azuresqlspark_2_4_0_2_11_1_3_4_uber_1_c81a3.jar
com.microsoft.azure.azure-sqldb-spark:1.0.2	Maven	Installed	

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 explicitly 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

```
1 // Connect to Azure SQL https://docs.databricks.com/data/data-sources/sql-databases-azure.html
2 import com.microsoft.azure.sqldb.spark.config.Config
3 import com.microsoft.azure.sqldb.spark.connect._
4
5 val configSql = Config(Map(
6   "url"          -> "https://[REDACTED].database.windows.net",
7   "databaseName" -> "[REDACTED]",
8   "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",
9   "user"         -> "[REDACTED]",
10  "password"     -> "[REDACTED]" // NOTE: For clarity and simplicity, this example includes secrets
11  explicitely as a string, but you should always use Databricks secrets
12 ))
13
14 // Create DataFrame from Azure SQL query
15 val orders = sqlContext.read.sqlDB(configSql)
16 display(orders)
```

▶ (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}]

String containing JSON array

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 explicitly 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)

```

```

1 // Convert DataFrame to proper nested schema
2 import org.apache.spark.sql.types._

3
4 val orderDetailsSchema = ArrayType(StructType(Array(
5     StructField("OrderDetailId", IntegerType, true),
6     StructField("ProductId", IntegerType, true),
7     StructField("UnitPrice", DoubleType, true),
8     StructField("Quantity", IntegerType, true)
9 )))

10
11 val ordersWithSchema = orders.select(
12     col("OrderId").cast("string").as("id"),
13     col("OrderDate").cast("string"),
14     col("FirstName").cast("string"),
15     col("LastName").cast("string"),
16     col("Address").cast("string"),
17     col("City").cast("string"),
18     col("State").cast("string"),
19     col("PostalCode").cast("string"),
20     col("Country").cast("string"),
21     col("Phone").cast("string"),
22     col("Total").cast("double"),
23     from_json(col("OrderDetails"), orderDetailsSchema).as("OrderDetails")
24 )

25
26 display(ordersWithSchema)
27
28 // Save nested data to Cosmos DB
29 CosmosDBSpark.save(ordersWithSchema, configCosmos)

```

(2) Spark Jobs

ordersWithSchema: org.apache.spark.sql.DataFrame = [id: string, OrderDate: string ... 10 more fields]

Last Name	Address	City	State	Postal Code	Country	Phone	Total	Order Details
Doe	123 W Main St	Cityville	IL	60001	United States	555-555-5555	10	array 0: {"OrderDetailId":10,"ProductId":200,"UnitPrice":10.0,"Quantity":1} 1: {"OrderDetailId":11,"ProductId":201,"UnitPrice":10.0,"Quantity":1}
Doe	456 W Main St	Cityville	IL	60001	United States	555-555-5551	100	array 0: {"OrderDetailId":12,"ProductId":200,"UnitPrice":3.0,"Quantity":10} 1: {"OrderDetailId":13,"ProductId":202,"UnitPrice":5.0,"Quantity":20} 2: {"OrderDetailId":14,"ProductId":203,"UnitPrice":9.0,"Quantity":10}

## 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://{}:{};database={};user={};password={}".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 dictionary 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:

The screenshot shows the Azure Cosmos DB SQL API interface. On the left, the navigation pane is visible with options like 'SQL API', 'StoreDatabase', 'Orders', 'Items', 'Scale & Settings', 'Stored Procedures', 'User Defined Functions', and 'Triggers'. The main area has a title 'Items' with a search bar and a button 'Edit Filter'. Below this is a table with columns 'id' and '/id'. The table contains three rows: 1002, 1000, and 1001. Row 1000 is selected and highlighted with a dashed border. A 'Load more' button is present at the bottom of the table. To the right of the table, a code editor displays the JSON representation of the selected document. The JSON object includes properties like 'Address', 'OrderDetails' (which is highlighted with a red box), 'FirstName', 'State', 'Phone', 'Total', 'PostalCode', 'Country', 'id', 'LastName', 'City', 'OrderDate', '\_rid', '\_self', '\_etag', '\_attachments', and '\_ts'. The 'OrderDetails' array contains two objects, each with 'UnitPrice', 'OrderDetailId', 'Quantity', and 'ProductId'.

```
1  {
2   "Address": "123 W Main St",
3   "OrderDetails": [
4     {
5       "UnitPrice": 3.5,
6       "OrderDetailId": 10,
7       "Quantity": 2,
8       "ProductId": 200
9     },
10    {
11      "UnitPrice": 3,
12      "OrderDetailId": 11,
13      "Quantity": 1,
14      "ProductId": 201
15    }
16  ],
17  "FirstName": "John",
18  "State": "IL",
19  "Phone": "555-555-5555",
20  "Total": 10,
21  "PostalCode": "60001",
22  "Country": "United States",
23  "id": "1000",
24  "LastName": "Doe",
25  "City": "Cityville",
26  "OrderDate": "2019-12-10 00:00:00",
27  "_rid": "tJN4AMgz6mIIAAAAAAA==",
28  "_self": "dbs/tJN4AA==/colls/tJN4AMgz6mI=/docs/tJN4AMg",
29  "_etag": "\\"7800a68a-0000-0200-0000-5deff8d60000\\\"",
30  "_attachments": "attachments/",
31  "_ts": 1576007894
32 }
```

## 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

2/14/2020 • 8 minutes to read • [Edit Online](#)

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
<pre>SELECT META(TravelDocument).id AS id, TravelDocument.* FROM TravelDocument WHERE _type = "com.xx.xx.xx.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</pre>	<pre>SELECT c.id,c FROM c JOIN m in c.country='India' WHERE c._type = "com.xx.xx.xx.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</pre>

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. For 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-AsynInSpring](#) 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",
  "includedPaths":
  [
    {
      "path": "/**",
      "indexes":
      [
        {
          "kind": "Range",
          "dataType": "Number"
        },
        {
          "kind": "Range",
          "dataType": "String"
        },
        {
          "kind": "Spatial",
          "dataType": "Point"
        }
      ]
    }
  ],
  "excludedPaths":
  [
    {
      "path": "/path/to/single/excluded/property/?"
    },
    {
      "path": "/path/to/root/of/multiple/excluded/properties/*"
    }
  ]
}
```

Replace the above indexing policy with the following policy:

```
{
  "indexingMode": "none"
}
```

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.

# Azure Cosmos DB bulk executor library overview

12/13/2019 • 2 minutes to read • [Edit Online](#)

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.

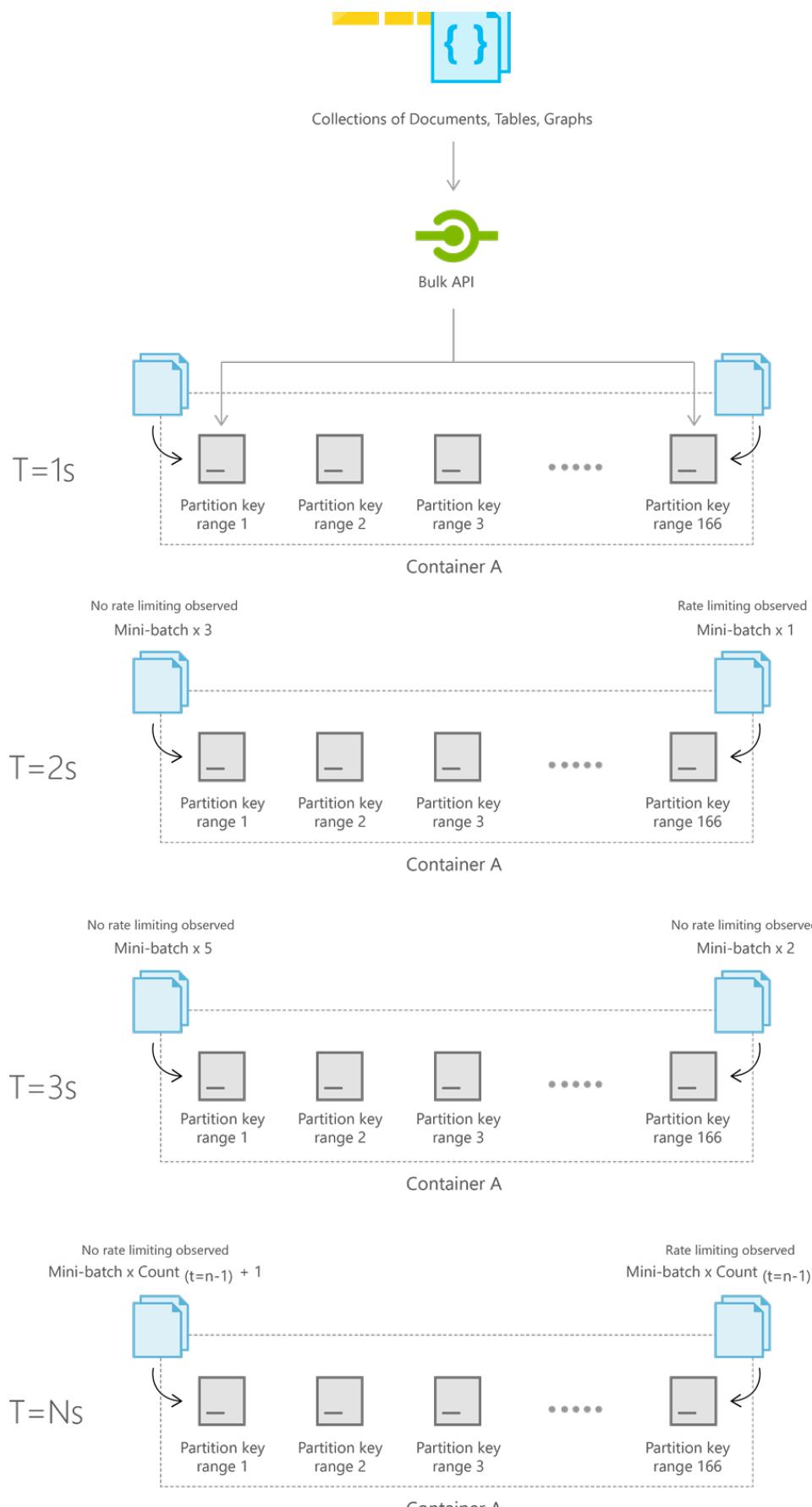
## Key features of the bulk executor library

- 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 an [AIMD-style congestion control mechanism](#) for each Azure Cosmos DB partition key range to 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

12/13/2019 • 7 minutes to read • [Edit Online](#)

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.

PARAMETER	DESCRIPTION
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));
}

```

- 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
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](#).

# Use bulk executor Java library to perform bulk operations on Azure Cosmos DB data

12/13/2019 • 8 minutes to read • [Edit Online](#)

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](#).

## 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.

PARAMETER	DESCRIPTION
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).

- 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
```

- 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 master 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](#).

- 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));
```

- 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.
Duration getTotalTimeTaken()	The total time taken by the bulk update API call to complete execution.

PARAMETER	DESCRIPTION
List<Exception> getErrors()	Gets the list of errors if some documents out of the batch supplied to the bulk update API call failed to get inserted.

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 master 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, see [bulk executor SDK details](#).

# Azure Cosmos DB .NET SDK for SQL API: Download and release notes

10/9/2019 • 16 minutes to read • [Edit Online](#)

<b>SDK download</b>	<a href="#">NuGet</a>
<b>API documentation</b>	<a href="#">.NET API reference documentation</a>
<b>Samples</b>	<a href="#">.NET code samples</a>
<b>Get started</b>	<a href="#">Get started with the Azure Cosmos DB .NET SDK</a>
<b>Web app tutorial</b>	<a href="#">Web application development with Azure Cosmos DB</a>
<b>Current supported framework</b>	<a href="#">Microsoft .NET Framework 4.5</a>

## Release notes

### NOTE

If you are using .NET Framework, please see the latest version 3.x of the [.NET SDK](#), which targets .NET Standard.

## 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](#)

## 2.9.0

- 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

## 2.3.0

- Added IdleTcpConnectionTimeout, OpenTcpConnectionTimeout, MaxRequestsPerTcpConnection and MaxTcpConnectionsPerEndpoint to ConnectionPolicy.

## 2.2.3

- Diagnostics improvements

## 2.2.2

- Added environment variable setting "POCO.SerializationOnly".
- 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 constructor overload which takes a HttpResponseMessageHandler, 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 JsonProperty 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 JsonSerializer 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.
- Performance 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 DocumentClient 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.

Any requests to Azure Cosmos DB using a retired SDK are rejected by the service.

**WARNING**

All versions **1.x** of the .NET SDK for SQL API will be retired on **August 30, 2020**.

VERSION	RELEASE DATE	RETIREMENT DATE
2.9.2	November 15, 2019	---
2.9.1	November 13, 2019	---

VERSION	RELEASE DATE	RETIREMENT DATE
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	---
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 30, 2020
1.21.1	March 09, 2018	August 30, 2020
1.20.2	February 21, 2018	August 30, 2020
1.20.1	February 05, 2018	August 30, 2020
1.19.1	November 16, 2017	August 30, 2020
1.19.0	November 10, 2017	August 30, 2020
1.18.1	November 07, 2017	August 30, 2020
1.18.0	October 17, 2017	August 30, 2020

VERSION	RELEASE DATE	RETIREMENT DATE
1.17.0	August 10, 2017	August 30, 2020
1.16.1	August 07, 2017	August 30, 2020
1.16.0	August 02, 2017	August 30, 2020
1.15.0	June 30, 2017	August 30, 2020
1.14.1	May 23, 2017	August 30, 2020
1.14.0	May 10, 2017	August 30, 2020
1.13.4	May 09, 2017	August 30, 2020
1.13.3	May 06, 2017	August 30, 2020
1.13.2	April 19, 2017	August 30, 2020
1.13.1	March 29, 2017	August 30, 2020
1.13.0	March 24, 2017	August 30, 2020
1.12.1	March 14, 2017	August 30, 2020
1.12.0	February 15, 2017	August 30, 2020
1.11.4	February 06, 2017	August 30, 2020
1.11.3	January 26, 2017	August 30, 2020
1.11.1	December 21, 2016	August 30, 2020
1.11.0	December 08, 2016	August 30, 2020
1.10.0	September 27, 2016	August 30, 2020
1.9.5	September 01, 2016	August 30, 2020
1.9.4	August 24, 2016	August 30, 2020
1.9.3	August 15, 2016	August 30, 2020
1.9.2	July 23, 2016	August 30, 2020
1.8.0	June 14, 2016	August 30, 2020
1.7.1	May 06, 2016	August 30, 2020
1.7.0	April 26, 2016	August 30, 2020

VERSION	RELEASE DATE	RETIREMENT DATE
1.6.3	April 08, 2016	August 30, 2020
1.6.2	March 29, 2016	August 30, 2020
1.5.3	February 19, 2016	August 30, 2020
1.5.2	December 14, 2015	August 30, 2020
1.5.1	November 23, 2015	August 30, 2020
1.5.0	October 05, 2015	August 30, 2020
1.4.1	August 25, 2015	August 30, 2020
1.4.0	August 13, 2015	August 30, 2020
1.3.0	August 05, 2015	August 30, 2020
1.2.0	July 06, 2015	August 30, 2020
1.1.0	April 30, 2015	August 30, 2020
1.0.0	April 08, 2015	August 30, 2020

## FAQ

**1. How will customers be notified of the retiring SDK?**

Microsoft will provide 12 month advance notification to the end of support of the retiring SDK in order to facilitate a smooth transition to a supported SDK. Further, customers will be notified through various communication channels – Azure Management Portal, Developer Center, blog post, and direct communication to assigned service administrators.

**2. Can customers author applications using a "to-be" retired Azure Cosmos DB SDK during the 12 month period?**

Yes, customers will have full access to author, deploy and modify applications using the "to-be" retired Azure Cosmos DB SDK during the 12 month grace period. During the 12 month grace period, customers are advised to migrate to a newer supported version of Azure Cosmos DB SDK as appropriate.

**3. Can customers author and modify applications using a retired Azure Cosmos DB SDK after the 12 month notification period?**

After the 12 month notification period, the SDK will be retired. Any access to Azure Cosmos DB by an applications using a retired SDK will not be permitted by the Azure Cosmos DB platform. Further, Microsoft will not provide customer support on the retired SDK.

**4. What happens to Customer's running applications that are using unsupported Azure Cosmos DB SDK version?**

Any attempts made to connect to the Azure Cosmos DB service with a retired SDK version will be rejected.

**5. Will new features and functionality be applied to all non-retired SDKs?**

New features and functionality will only be added to new versions. If you are using an old, non-retired, version of the SDK your requests to Azure Cosmos DB will still function as previous but you will not have access to any new capabilities.

## **6. What should I do if I cannot update my application before a cut-off date?**

We recommend that you upgrade to the latest SDK as early as possible. Once an SDK has been tagged for retirement you will have 12 months to update your application. If, for whatever reason, you cannot complete your application update within this timeframe then please contact the [Cosmos DB Team](#) and request their assistance before the cutoff date.

## See also

To learn more about Cosmos DB, see [Microsoft Azure Cosmos DB](#) service page.

# Azure Cosmos DB .NET Core SDK for SQL API: Release notes and resources

10/9/2019 • 14 minutes to read • [Edit Online](#)

<b>SDK download</b>	<a href="#">NuGet</a>
<b>API documentation</b>	<a href="#">.NET API reference documentation</a>
<b>Samples</b>	<a href="#">.NET code samples</a>
<b>Get started</b>	<a href="#">Get started with the Azure Cosmos DB .NET</a>
<b>Web app tutorial</b>	<a href="#">Web application development with Azure Cosmos DB</a>
<b>Current supported framework</b>	<a href="#">.NET Standard 1.6 and .NET Standard 1.5</a>

## Release Notes

### NOTE

If you are using .NET Core, please see the latest version 3.x of the [.NET SDK](#), which targets .NET Standard.

## 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](#)

## 2.9.0

- 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 nt!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

## 2.3.0

- Added IdleTcpConnectionTimeout, OpenTcpConnectionTimeout, MaxRequestsPerTcpConnection and MaxTcpConnectionsPerEndpoint to ConnectionPolicy.

## 2.2.3

- Diagnostics improvements

## 2.2.2

- Added environment variable setting "POCO.SerializationOnly".
- 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 constructor overload which takes a HttpResponseMessageHandler, 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 JsonProperty 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 JsonSerializer 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.
- Performance 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 `DocumentClient` 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.

Any requests to Azure Cosmos DB using a retired SDK are rejected by the service.

**WARNING**

All versions **1.x** of the .NET SDK for SQL API will be retired on **August 30, 2020**.

VERSION	RELEASE DATE	RETIREMENT DATE
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	---

VERSION	RELEASE DATE	RETIREMENT DATE
<a href="#">2.5.1</a>	July 02, 2019	---
<a href="#">2.4.1</a>	June 20, 2019	---
<a href="#">2.4.0</a>	May 05, 2019	---
<a href="#">2.3.0</a>	April 04, 2019	---
<a href="#">2.2.3</a>	February 11, 2019	---
<a href="#">2.2.2</a>	February 06, 2019	---
<a href="#">2.2.1</a>	December 24, 2018	---
<a href="#">2.2.0</a>	December 07, 2018	---
<a href="#">2.1.3</a>	October 15, 2018	---
<a href="#">2.1.2</a>	October 04, 2018	---
<a href="#">2.1.1</a>	September 27, 2018	---
<a href="#">2.1.0</a>	September 21, 2018	---
<a href="#">2.0.0</a>	September 07, 2018	---
<a href="#">1.22.0</a>	April 19, 2018	August 30, 2020
<a href="#">1.21.1</a>	March 09, 2018	August 30, 2020
<a href="#">1.20.2</a>	February 21, 2018	August 30, 2020
<a href="#">1.20.1</a>	February 05, 2018	August 30, 2020
<a href="#">1.19.1</a>	November 16, 2017	August 30, 2020
<a href="#">1.19.0</a>	November 10, 2017	August 30, 2020
<a href="#">1.18.1</a>	November 07, 2017	August 30, 2020
<a href="#">1.18.0</a>	October 17, 2017	August 30, 2020
<a href="#">1.17.0</a>	August 10, 2017	August 30, 2020
<a href="#">1.16.1</a>	August 07, 2017	August 30, 2020
<a href="#">1.16.0</a>	August 02, 2017	August 30, 2020
<a href="#">1.15.0</a>	June 30, 2017	August 30, 2020

VERSION	RELEASE DATE	RETIREMENT DATE
<a href="#">1.14.1</a>	May 23, 2017	August 30, 2020
<a href="#">1.14.0</a>	May 10, 2017	August 30, 2020
<a href="#">1.13.4</a>	May 09, 2017	August 30, 2020
<a href="#">1.13.3</a>	May 06, 2017	August 30, 2020
<a href="#">1.13.2</a>	April 19, 2017	August 30, 2020
<a href="#">1.13.1</a>	March 29, 2017	August 30, 2020
<a href="#">1.13.0</a>	March 24, 2017	August 30, 2020
<a href="#">1.12.1</a>	March 14, 2017	August 30, 2020
<a href="#">1.12.0</a>	February 15, 2017	August 30, 2020
<a href="#">1.11.4</a>	February 06, 2017	August 30, 2020
<a href="#">1.11.3</a>	January 26, 2017	August 30, 2020
<a href="#">1.11.1</a>	December 21, 2016	August 30, 2020
<a href="#">1.11.0</a>	December 08, 2016	August 30, 2020
<a href="#">1.10.0</a>	September 27, 2016	August 30, 2020
<a href="#">1.9.5</a>	September 01, 2016	August 30, 2020
<a href="#">1.9.4</a>	August 24, 2016	August 30, 2020
<a href="#">1.9.3</a>	August 15, 2016	August 30, 2020
<a href="#">1.9.2</a>	July 23, 2016	August 30, 2020
<a href="#">1.8.0</a>	June 14, 2016	August 30, 2020
<a href="#">1.7.1</a>	May 06, 2016	August 30, 2020
<a href="#">1.7.0</a>	April 26, 2016	August 30, 2020
<a href="#">1.6.3</a>	April 08, 2016	August 30, 2020
<a href="#">1.6.2</a>	March 29, 2016	August 30, 2020
<a href="#">1.5.3</a>	February 19, 2016	August 30, 2020
<a href="#">1.5.2</a>	December 14, 2015	August 30, 2020

VERSION	RELEASE DATE	RETIREMENT DATE
1.5.1	November 23, 2015	August 30, 2020
1.5.0	October 05, 2015	August 30, 2020
1.4.1	August 25, 2015	August 30, 2020
1.4.0	August 13, 2015	August 30, 2020
1.3.0	August 05, 2015	August 30, 2020
1.2.0	July 06, 2015	August 30, 2020
1.1.0	April 30, 2015	August 30, 2020
1.0.0	April 08, 2015	August 30, 2020

## See Also

To learn more about Cosmos DB, see [Microsoft Azure Cosmos DB](#) service page.

# Azure Cosmos DB .NET Standard SDK for SQL API: Download and release notes

12/2/2019 • 8 minutes to read • [Edit Online](#)

<b>SDK download</b>	<a href="#">NuGet</a>
<b>API documentation</b>	<a href="#">.NET API reference documentation</a>
<b>Samples</b>	<a href="#">.NET code samples</a>
<b>Get started</b>	<a href="#">Get started with the Azure Cosmos DB .NET SDK</a>
<b>Web app tutorial</b>	<a href="#">Web application development with Azure Cosmos DB</a>
<b>Current supported framework</b>	<a href="#">Microsoft .NET Standard 2.0</a>

## Changelog

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](#).

## Unreleased

### Added

### Fixed

## [3.7.0-preview](#) - 2020-02-25

- [#1210](#) Change Feed pull model

## [3.6.0](#) - 2020-01-23

### Added

- [#1097](#) Add GeospatialConfig to ContainerProperties, BoundingBoxProperties to SpatialPath
- [#1061](#) Add Stream payload to ExecuteStoredProcedureStreamAsync
- [#1062](#) Add additional diagnostic information including the ability to track time through the different SDK layers
- [#1107](#) Add 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](#) Fixed SDK properties like DatabaseProperties to have same JSON attributes
- [#1116](#) Fixed a deadlock on scenarios with SynchronizationContext while executing async query operations

- [#1143](#) Fixed permission resource link and authorization issue when doing a query with resource token for a specific partition key
- [#1150](#) Fixed NullReferenceException when using a non-existent Lease Container.

## 3.5.1 - 2019-12-11

### Fixed

- [#1060](#) Fixed 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 BadRequest with large headers
- [#1078](#) Fixed a deadlock on scenarios with SynchronizationContext while executing async SDK API
- [#1081](#) Fixed race condition in serializer caused null reference exception.
- [#1086](#) Fix possible NullReferenceException on a TransactionalBatch code path
- [#1091](#) Fixed 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 with no PK

## 3.5.0 - 2019-12-03

### Added

- [#979](#) Make SessionToken on QueryRequestOptions public.
- [#995](#) Included session token in diagnostics.
- [#1000](#) Add PortReuseMode to CosmosClientOptions.
- [#1017](#) Adding ClientSideRequestStatistics to gateway calls and making endtime nullable
- [#1038](#) Add Selflink to resource properties

### Fixed

- [#921](#) Fixed error handling to preserve stack trace in certain scenarios
- [#944](#) Change Feed Processor won't use user serializer for internal operations
- [#988](#) Fixed 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](#) Fixed grabbing extra page, updated continuation token on exception path, and non ascii character in order by continuation token.
- [#1013](#) Gateway OperationCanceledException are now returned as request timeouts
- [#1020](#) Direct package update removes debug statements
- [#1023](#) Fixed ThroughputResponse.IsReplacePending header mapping
- [#1036](#) Fixed query responses to return null Content if it is a failure
- [#1045](#) Added stack trace and inner exception to CosmosException
- [#1050](#) Add mocking constructors to TransactionalBatchOperationResult

## 3.4.1 - 2019-11-06

### Fixed

- [#978](#) Fixed 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](#) Fixed a bug causing query response to create a new stream for each content call
- [#918](#) Fixed serializer being used for Scripts, Permissions, and Conflict related iterators
- [#936](#) Fixed bulk requests with large resources to have natural exception

### [3.3.3 - 2019-10-30](#)

- [#837](#) Fixed group by bug for non-Windows platforms
- [#927](#) Fixed query returning partial results instead of error

### [3.3.2 - 2019-10-16](#)

#### **Fixed**

- [#905](#) Fixed linq camel case bug

### [3.3.1 - 2019-10-11](#)

#### **Fixed**

- [#895](#) Fixed 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](#) Added PartitionKeyDefinitionVersion to container builder

#### **Fixed**

- [#835](#) Fixed a bug that caused sortedRanges exceptions
- [#846](#) Statistics not getting populated correctly on CosmosException.
- [#857](#) Fixed reusability of the Bulk support across Container instances
- [#860](#) Fixed base user agent string
- [#876](#) Default connection timeout reduced from 60s to 10s

### [3.2.0 - 2019-09-17](#)

#### **Added**

- [#100](#) Configurable Tcp settings to CosmosClientOptions
- [#615, #775](#) Added request diagnostics to Response's
- [#622](#) Added CRUD and query operations for Users and Permissions which enables [ResourceToken](#) support
- [#716](#) Added camel case serialization on LINQ query generation
- [#729, #776](#) Added aggregate(CountAsync/SumAsync etc.) extensions for LINQ query

- [#743](#) Added 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 (Thanks to jkonecki)
- [#769](#) Session Consistency + Gateway mode session-token bug fix: Under few rare non-success cases session token might be in-correct
- [#772](#) Fixed 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](#) Fixed SpatialPath serialization and compatibility with older index versions
- [#619](#) Fixed PInvokeStackImbalance exception for .NET Framework
- [#626](#) FeedResponse status code now return OK for success instead of the invalid status code 0 or Accepted
- [#629](#) Fixed CreateContainerIfNotExistsAsync validation to limited to partitionKeyPath only
- [#630](#) Fixed User Agent to contain environment and package information

## 3.1.0 - 2019-07-29 - Unlisted

#### Added

- [#541](#) Added consistency level to client and query options
- [#544](#) Added continuation token support for LINQ
- [#557](#) Added trigger options to item request options
- [#572](#) Added partition key validation on CreateContainerIfNotExistsAsync
- [#581](#) Added LINQ to QueryDefinition API
- [#592](#) Added CreateIfNotExistsAsync to container builder
- [#597](#) Added continuation token property to ResponseMessage
- [#604](#) Added LINQ ToStreamIterator extension method

#### Fixed

- [#548](#) Fixed mis-typed message in CosmosException.ToString();
- [#558](#) LocationCache ConcurrentDict lock contention fix

- [#561](#) GetItemLinqQueryable now works with null query
- [#567](#) Query correctly handles different language cultures
- [#574](#) Fixed 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 highly performant stream APIs
- 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.

Any requests to Azure Cosmos DB using a retired SDK are rejected by the service.

VERSION	RELEASE DATE	RETIREMENT DATE
<a href="#">3.6.0</a>	January 23, 2020	---
<a href="#">3.5.1</a>	December 11, 2019	---
<a href="#">3.5.0</a>	December 03, 2019	---
<a href="#">3.4.1</a>	November 06, 2019	---
<a href="#">3.4.0</a>	November 04, 2019	---
<a href="#">3.3.3</a>	October 30, 2019	---
<a href="#">3.3.2</a>	October 16, 2019	---
<a href="#">3.3.1</a>	October 11, 2019	---
<a href="#">3.3.0</a>	October 8, 2019	---

VERSION	RELEASE DATE	RETIREMENT DATE
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

### 1. How will customers be notified of the retiring SDK?

Microsoft will provide 12 month advance notification to the end of support of the retiring SDK in order to facilitate a smooth transition to a supported SDK. Further, customers will be notified through various communication channels – Azure Management Portal, Developer Center, blog post, and direct communication to assigned service administrators.

### 2. Can customers author applications using a "to-be" retired Azure Cosmos DB SDK during the 12 month period?

Yes, customers will have full access to author, deploy and modify applications using the "to-be" retired Azure Cosmos DB SDK during the 12 month grace period. During the 12 month grace period, customers are advised to migrate to a newer supported version of Azure Cosmos DB SDK as appropriate.

### 3. Can customers author and modify applications using a retired Azure Cosmos DB SDK after the 12 month notification period?

After the 12 month notification period, the SDK will be retired. Any access to Azure Cosmos DB by an applications using a retired SDK will not be permitted by the Azure Cosmos DB platform. Further, Microsoft will not provide customer support on the retired SDK.

### 4. What happens to Customer's running applications that are using unsupported Azure Cosmos DB SDK version?

Any attempts made to connect to the Azure Cosmos DB service with a retired SDK version will be rejected.

### 5. Will new features and functionality be applied to all non-retired SDKs?

New features and functionality will only be added to new versions. If you are using an old, non-retired, version of the SDK your requests to Azure Cosmos DB will still function as previous but you will not have access to any new capabilities.

### 6. What should I do if I cannot update my application before a cut-off date?

We recommend that you upgrade to the latest SDK as early as possible. Once an SDK has been tagged for retirement you will have 12 months to update your application. If, for whatever reason, you cannot complete your application update within this timeframe then please contact the [Cosmos DB Team](#) and request their assistance before the cutoff date.

## See also

To learn more about Cosmos DB, see [Microsoft Azure Cosmos DB](#) service page.

# .NET Change Feed Processor SDK: Download and release notes

12/13/2019 • 8 minutes to read • [Edit Online](#)

<b>SDK download</b>	<a href="#">NuGet</a>
<b>API documentation</b>	<a href="#">Change Feed Processor library API reference documentation</a>
<b>Get started</b>	<a href="#">Get started with the Change Feed Processor .NET SDK</a>
<b>Current supported framework</b>	<a href="#">Microsoft .NET Framework 4.5</a> <a href="#">Microsoft .NET Core</a>

## 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

#### 2.2.8

- 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 health 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, e.g. 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 re-balancing a lot of 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 re-balancing 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 Multi Master accounts 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, e.g. 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.

Any request to Cosmos DB using a retired SDK will be rejected by the service.

VERSION	RELEASE DATE	RETIREMENT DATE
2.2.8	October 28, 2019	---

VERSION	RELEASE DATE	RETIREMENT DATE
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	---
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

### **1. How will customers be notified of the retiring SDK?**

Microsoft will provide 12 month advance notification to the end of support of the retiring SDK in order to facilitate a smooth transition to a supported SDK. Further, customers will be notified through various communication channels – Azure Management Portal, Developer Center, blog post, and direct communication to assigned service administrators.

### **2. Can customers author applications using a "to-be" retired Azure Cosmos DB SDK during the 12 month period?**

Yes, customers will have full access to author, deploy and modify applications using the "to-be" retired Azure Cosmos DB SDK during the 12 month grace period. During the 12 month grace period, customers are advised to migrate to a newer supported version of Azure Cosmos DB SDK as appropriate.

### **3. Can customers author and modify applications using a retired Azure Cosmos DB SDK after the 12 month notification period?**

After the 12 month notification period, the SDK will be retired. Any access to Azure Cosmos DB by an applications using a retired SDK will not be permitted by the Azure Cosmos DB platform. Further, Microsoft will not provide customer support on the retired SDK.

### **4. What happens to Customer's running applications that are using unsupported Azure Cosmos DB SDK version?**

Any attempts made to connect to the Azure Cosmos DB service with a retired SDK version will be rejected.

## **5. Will new features and functionality be applied to all non-retired SDKs?**

New features and functionality will only be added to new versions. If you are using an old, non-retired, version of the SDK your requests to Azure Cosmos DB will still function as previous but you will not have access to any new capabilities.

## **6. What should I do if I cannot update my application before a cut-off date?**

We recommend that you upgrade to the latest SDK as early as possible. Once an SDK has been tagged for retirement you will have 12 months to update your application. If, for whatever reason, you cannot complete your application update within this timeframe then please contact the [Cosmos DB Team](#) and request their assistance before the cutoff date.

## See also

To learn more about Cosmos DB, see [Microsoft Azure Cosmos DB service page](#).

# .NET bulk executor library: Download information

1/17/2020 • 3 minutes to read • [Edit Online](#)

<b>Description</b>	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.
<b>SDK download</b>	<a href="#">NuGet</a>
<b>Bulk executor library in GitHub</b>	<a href="#">GitHub</a>
<b>API documentation</b>	<a href="#">.NET API reference documentation</a>
<b>Get started</b>	<a href="#">Get started with the bulk executor library .NET SDK</a>
<b>Current supported framework</b>	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.

### **1.8.8**

- 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 for SQL API: Release notes and resources

2/21/2020 • 9 minutes to read • [Edit Online](#)

The SQL API Java SDK supports synchronous operations. For asynchronous support, use the [SQL API Async Java SDK](#).

<b>SDK Download</b>	<a href="#">Maven</a>
<b>API documentation</b>	<a href="#">Java API reference documentation</a>
<b>Contribute to SDK</b>	<a href="#">GitHub</a>
<b>Get started</b>	<a href="#">Get started with the Java SDK</a>
<b>Web app tutorial</b>	<a href="#">Web application development with Azure Cosmos DB</a>
<b>Minimum supported runtime</b>	<a href="#">Java Development Kit (JDK) 7 +</a>

## Release notes

### 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 pkRangeld.

### 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 available for download from <https://aka.ms/cosmosdb-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 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 SQL SDK for Java will be retired on **May 30, 2020**.

#### WARNING

All versions of the SQL SDK for Java prior to version **1.0.0** were retired on **February 29, 2016**.

VERSION	RELEASE DATE	RETIREMENT DATE
<a href="#">2.4.7</a>	Feb 20, 2020	---
<a href="#">2.4.6</a>	Jan 24, 2020	---

VERSION	RELEASE DATE	RETIREMENT DATE
<a href="#">2.4.5</a>	Nov 10, 2019	---
<a href="#">2.4.4</a>	Oct 24, 2019	---
<a href="#">2.4.2</a>	Sep 26, 2019	---
<a href="#">2.4.1</a>	Jul 18, 2019	---
<a href="#">2.4.0</a>	May 04, 2019	---
<a href="#">2.3.0</a>	Apr 24, 2019	---
<a href="#">2.2.3</a>	Apr 16, 2019	---
<a href="#">2.2.2</a>	Apr 05, 2019	---
<a href="#">2.2.0</a>	Mar 27, 2019	---
<a href="#">2.1.3</a>	Mar 13, 2019	---
<a href="#">2.1.2</a>	Mar 09, 2019	---
<a href="#">2.1.1</a>	Dec 13, 2018	---
<a href="#">2.1.0</a>	Nov 20, 2018	---
<a href="#">2.0.0</a>	Sept 21, 2018	---
<a href="#">1.16.4</a>	Sept 10, 2018	May 30, 2020
<a href="#">1.16.3</a>	Sept 09, 2018	May 30, 2020
<a href="#">1.16.2</a>	June 29, 2018	May 30, 2020
<a href="#">1.16.1</a>	May 16, 2018	May 30, 2020
<a href="#">1.16.0</a>	March 15, 2018	May 30, 2020
<a href="#">1.15.0</a>	Nov 14, 2017	May 30, 2020
<a href="#">1.14.0</a>	Oct 28, 2017	May 30, 2020
<a href="#">1.13.0</a>	August 25, 2017	May 30, 2020
<a href="#">1.12.0</a>	July 11, 2017	May 30, 2020
<a href="#">1.11.0</a>	May 10, 2017	May 30, 2020
<a href="#">1.10.0</a>	March 11, 2017	May 30, 2020

VERSION	RELEASE DATE	RETIREMENT DATE
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
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

VERSION	RELEASE DATE	RETIREMENT DATE
0.9.0-prelease	December 10, 2014	February 29, 2016

## FAQ

### **1. How will customers be notified of the retiring SDK?**

Microsoft will provide 12 month advance notification to the end of support of the retiring SDK in order to facilitate a smooth transition to a supported SDK. Further, customers will be notified through various communication channels – Azure Management Portal, Developer Center, blog post, and direct communication to assigned service administrators.

### **2. Can customers author applications using a "to-be" retired Azure Cosmos DB SDK during the 12 month period?**

Yes, customers will have full access to author, deploy and modify applications using the "to-be" retired Azure Cosmos DB SDK during the 12 month grace period. During the 12 month grace period, customers are advised to migrate to a newer supported version of Azure Cosmos DB SDK as appropriate.

### **3. Can customers author and modify applications using a retired Azure Cosmos DB SDK after the 12 month notification period?**

After the 12 month notification period, the SDK will be retired. Any access to Azure Cosmos DB by an applications using a retired SDK will not be permitted by the Azure Cosmos DB platform. Further, Microsoft will not provide customer support on the retired SDK.

### **4. What happens to Customer's running applications that are using unsupported Azure Cosmos DB SDK version?**

Any attempts made to connect to the Azure Cosmos DB service with a retired SDK version will be rejected.

### **5. Will new features and functionality be applied to all non-retired SDKs?**

New features and functionality will only be added to new versions. If you are using an old, non-retired, version of the SDK your requests to Azure Cosmos DB will still function as previous but you will not have access to any new capabilities.

### **6. What should I do if I cannot update my application before a cut-off date?**

We recommend that you upgrade to the latest SDK as early as possible. Once an SDK has been tagged for retirement you will have 12 months to update your application. If, for whatever reason, you cannot complete your application update within this timeframe then please contact the [Cosmos DB Team](#) and request their assistance before the cutoff date.

## See also

To learn more about Cosmos DB, see [Microsoft Azure Cosmos DB](#) service page.

# Java bulk executor library: Download information

11/4/2019 • 2 minutes to read • [Edit Online](#)

<b>Description</b>	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.
<b>SDK download</b>	<a href="#">Maven</a>
<b>Bulk executor library in GitHub</b>	<a href="#">GitHub</a>
<b>API documentation</b>	<a href="#">Java API reference documentation</a>
<b>Get started</b>	<a href="#">Get started with the bulk executor library Java SDK</a>
<b>Minimum supported runtime</b>	<a href="#">Java Development Kit (JDK) 7+</a>

# Azure Cosmos DB Async Java SDK for SQL API: Release notes and resources

11/4/2019 • 7 minutes to read • [Edit Online](#)

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.

<b>SDK Download</b>	<a href="#">Maven</a>
<b>API documentation</b>	<a href="#">Java API reference documentation</a>
<b>Contribute to SDK</b>	<a href="#">GitHub</a>
<b>Get started</b>	<a href="#">Get started with the Async Java SDK</a>
<b>Code sample</b>	<a href="#">GitHub</a>
<b>Performance tips</b>	<a href="#">GitHub readme</a>
<b>Minimum supported runtime</b>	<a href="#">JDK 8</a>

## Changelog

### 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 Partition 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	---

VERSION	RELEASE DATE	RETIREMENT DATE
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	---
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

### 1. How will customers be notified of the retiring SDK?

Microsoft will provide 12 month advance notification to the end of support of the retiring SDK in order to facilitate a smooth transition to a supported SDK. Further, customers will be notified through various communication channels – Azure Management Portal, Developer Center, blog post, and direct communication to assigned service administrators.

### 2. Can customers author applications using a "to-be" retired Azure Cosmos DB SDK during the 12 month period?

Yes, customers will have full access to author, deploy and modify applications using the "to-be" retired Azure Cosmos DB SDK during the 12 month grace period. During the 12 month grace period, customers are advised to migrate to a newer supported version of Azure Cosmos DB SDK as appropriate.

### 3. Can customers author and modify applications using a retired Azure Cosmos DB SDK after the 12 month notification period?

After the 12 month notification period, the SDK will be retired. Any access to Azure Cosmos DB by an applications using a retired SDK will not be permitted by the Azure Cosmos DB platform. Further, Microsoft will not provide customer support on the retired SDK.

#### **4. What happens to Customer's running applications that are using unsupported Azure Cosmos DB SDK version?**

Any attempts made to connect to the Azure Cosmos DB service with a retired SDK version will be rejected.

#### **5. Will new features and functionality be applied to all non-retired SDKs?**

New features and functionality will only be added to new versions. If you are using an old, non-retired, version of the SDK your requests to Azure Cosmos DB will still function as previous but you will not have access to any new capabilities.

#### **6. What should I do if I cannot update my application before a cut-off date?**

We recommend that you upgrade to the latest SDK as early as possible. Once an SDK has been tagged for retirement you will have 12 months to update your application. If, for whatever reason, you cannot complete your application update within this timeframe then please contact the [Cosmos DB Team](#) and request their assistance before the cutoff date.

### See also

To learn more about Cosmos DB, see [Microsoft Azure Cosmos DB](#) service page.

# Azure Cosmos DB Node.js SDK for SQL API: Release notes and resources

9/1/2019 • 14 minutes to read • [Edit Online](#)

RESOURCE	LINK
Download SDK	<a href="#">NPM</a>
API Documentation	<a href="#">JavaScript SDK reference documentation</a>
SDK installation instructions	<a href="#">Installation instructions</a>
Contribute to SDK	<a href="#">GitHub</a>
Samples	<a href="#">Node.js code samples</a>
Getting started tutorial	<a href="#">Get started with the JavaScript SDK</a>
Web app tutorial	<a href="#">Build a Node.js web application using Azure Cosmos DB</a>
Current supported platform	<a href="#">Node.js v12.x</a> - SDK Version 3.x.x <a href="#">Node.js v10.x</a> - SDK Version 3.x.x <a href="#">Node.js v8.x</a> - SDK Version 3.x.x <a href="#">Node.js v6.x</a> - SDK Version 2.x.x <a href="#">Node.js v4.2.0</a> - SDK Version 1.x.x <a href="#">Node.js v0.12</a> - SDK Version 1.x.x <a href="#">Node.js v0.10</a> - 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 available for download from <https://aka.ms/cosmosdb-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 SSL 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 SSL 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.

Any request to Cosmos DB using a retired SDK will be rejected by the service.

### WARNING

All versions **1.x** of the Node client SDK for SQL API will be retired on **August 30, 2020**. This affects only the client-side Node SDK and does not affect server-side scripts (stored procedures, triggers, and UDFs).

VERSION	RELEASE DATE	RETIREMENT DATE
<a href="#">3.1.0</a>	July 26, 2019	---
<a href="#">3.0.4</a>	July 22, 2019	---

VERSION	RELEASE DATE	RETIREMENT DATE
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	---
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

VERSION	RELEASE DATE	RETIREMENT DATE
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
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

## **1. How will customers be notified of the retiring SDK?**

Microsoft will provide 12 month advance notification to the end of support of the retiring SDK in order to facilitate a smooth transition to a supported SDK. Further, customers will be notified through various communication channels – Azure Management Portal, Developer Center, blog post, and direct communication to assigned service administrators.

## **2. Can customers author applications using a "to-be" retired Azure Cosmos DB SDK during the 12 month period?**

Yes, customers will have full access to author, deploy and modify applications using the "to-be" retired Azure Cosmos DB SDK during the 12 month grace period. During the 12 month grace period, customers are advised to migrate to a newer supported version of Azure Cosmos DB SDK as appropriate.

## **3. Can customers author and modify applications using a retired Azure Cosmos DB SDK after the 12 month notification period?**

After the 12 month notification period, the SDK will be retired. Any access to Azure Cosmos DB by an applications using a retired SDK will not be permitted by the Azure Cosmos DB platform. Further, Microsoft will not provide customer support on the retired SDK.

## **4. What happens to Customer's running applications that are using unsupported Azure Cosmos DB SDK version?**

Any attempts made to connect to the Azure Cosmos DB service with a retired SDK version will be rejected.

## **5. Will new features and functionality be applied to all non-retired SDKs?**

New features and functionality will only be added to new versions. If you are using an old, non-retired, version of the SDK your requests to Azure Cosmos DB will still function as previous but you will not have access to any new capabilities.

## **6. What should I do if I cannot update my application before a cut-off date?**

We recommend that you upgrade to the latest SDK as early as possible. Once an SDK has been tagged for retirement you will have 12 months to update your application. If, for whatever reason, you cannot complete your application update within this timeframe then please contact the [Cosmos DB Team](#) and request their assistance before the cutoff date.

## 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

8/29/2019 • 6 minutes to read • [Edit Online](#)

<b>Download SDK</b>	<a href="#">PyPI</a>
<b>API documentation</b>	<a href="#">Python API reference documentation</a>
<b>SDK installation instructions</b>	<a href="#">Python SDK installation instructions</a>
<b>Contribute to SDK</b>	<a href="#">GitHub</a>
<b>Get started</b>	<a href="#">Get started with the Python SDK</a>
<b>Current supported platform</b>	<a href="#">Python 2.7 and Python 3.5</a>

## Release notes

### 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

- Support for multi-region writes.
- Namespace changed to `azure.cosmos`.
- Collection and document concepts renamed to container and item, `document_client` renamed to `cosmos_client`.

### 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

- Bug fix for aggregate dictionary.
- Bug fix for trimming slashes in the resource link.
- Added tests for Unicode encoding.

### 2.2.0

- 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 Cosmos DB 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 a 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, 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.
- 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 multi-region database accounts.

### 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

- Implemented [partitioned collections](#) and [user-defined performance levels](#).

## 1.5.0

- Add Hash & Range partition resolvers to assist with sharding applications across multiple partitions.

## 1.4.2

- Implement Upsert. New UpsertXXX methods added to support Upsert feature.
- Implement ID Based Routing. No public API changes, all changes internal.

## 1.2.0

- 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.

## 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 recommend that you always upgrade to the latest SDK version as early as possible.

Any request to Cosmos DB using a retired SDK are rejected by the service.

### WARNING

All versions of the Python SDK for SQL API prior to version **1.0.0** were retired on **February 29, 2016**.

### WARNING

All versions 1.x and 2.x of the Python SDK for SQL API will be retired on **August 30, 2020**.

VERSION	RELEASE DATE	RETIREMENT DATE
3.0.2	Nov 15, 2018	---
3.0.1	Oct 04, 2018	---

VERSION	RELEASE DATE	RETIREMENT DATE
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
0.9.2-prelease	November 25, 2014	February 29, 2016
0.9.1-prelease	September 23, 2014	February 29, 2016

VERSION	RELEASE DATE	RETIREMENT DATE
0.9.0-prelease	August 21, 2014	February 29, 2016

## FAQ

### **1. How will customers be notified of the retiring SDK?**

Microsoft will provide 12 month advance notification to the end of support of the retiring SDK in order to facilitate a smooth transition to a supported SDK. Further, customers will be notified through various communication channels – Azure Management Portal, Developer Center, blog post, and direct communication to assigned service administrators.

### **2. Can customers author applications using a "to-be" retired Azure Cosmos DB SDK during the 12 month period?**

Yes, customers will have full access to author, deploy and modify applications using the "to-be" retired Azure Cosmos DB SDK during the 12 month grace period. During the 12 month grace period, customers are advised to migrate to a newer supported version of Azure Cosmos DB SDK as appropriate.

### **3. Can customers author and modify applications using a retired Azure Cosmos DB SDK after the 12 month notification period?**

After the 12 month notification period, the SDK will be retired. Any access to Azure Cosmos DB by an applications using a retired SDK will not be permitted by the Azure Cosmos DB platform. Further, Microsoft will not provide customer support on the retired SDK.

### **4. What happens to Customer's running applications that are using unsupported Azure Cosmos DB SDK version?**

Any attempts made to connect to the Azure Cosmos DB service with a retired SDK version will be rejected.

### **5. Will new features and functionality be applied to all non-retired SDKs?**

New features and functionality will only be added to new versions. If you are using an old, non-retired, version of the SDK your requests to Azure Cosmos DB will still function as previous but you will not have access to any new capabilities.

### **6. What should I do if I cannot update my application before a cut-off date?**

We recommend that you upgrade to the latest SDK as early as possible. Once an SDK has been tagged for retirement you will have 12 months to update your application. If, for whatever reason, you cannot complete your application update within this timeframe then please contact the [Cosmos DB Team](#) and request their assistance before the cutoff date.

## See also

To learn more about Cosmos DB, see [Microsoft Azure Cosmos DB](#) service page.

# Azure Cosmos Emulator - Release notes and download information

2/12/2020 • 2 minutes to read • [Edit Online](#)

This article shows the Azure Cosmos 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

<a href="#">MSI download</a>	Microsoft Download Center
<a href="#">Get started</a>	Develop locally with Azure Cosmos emulator

## Release notes

### 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 shutdown 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.

# Azure Cosmos DB query cheat sheets

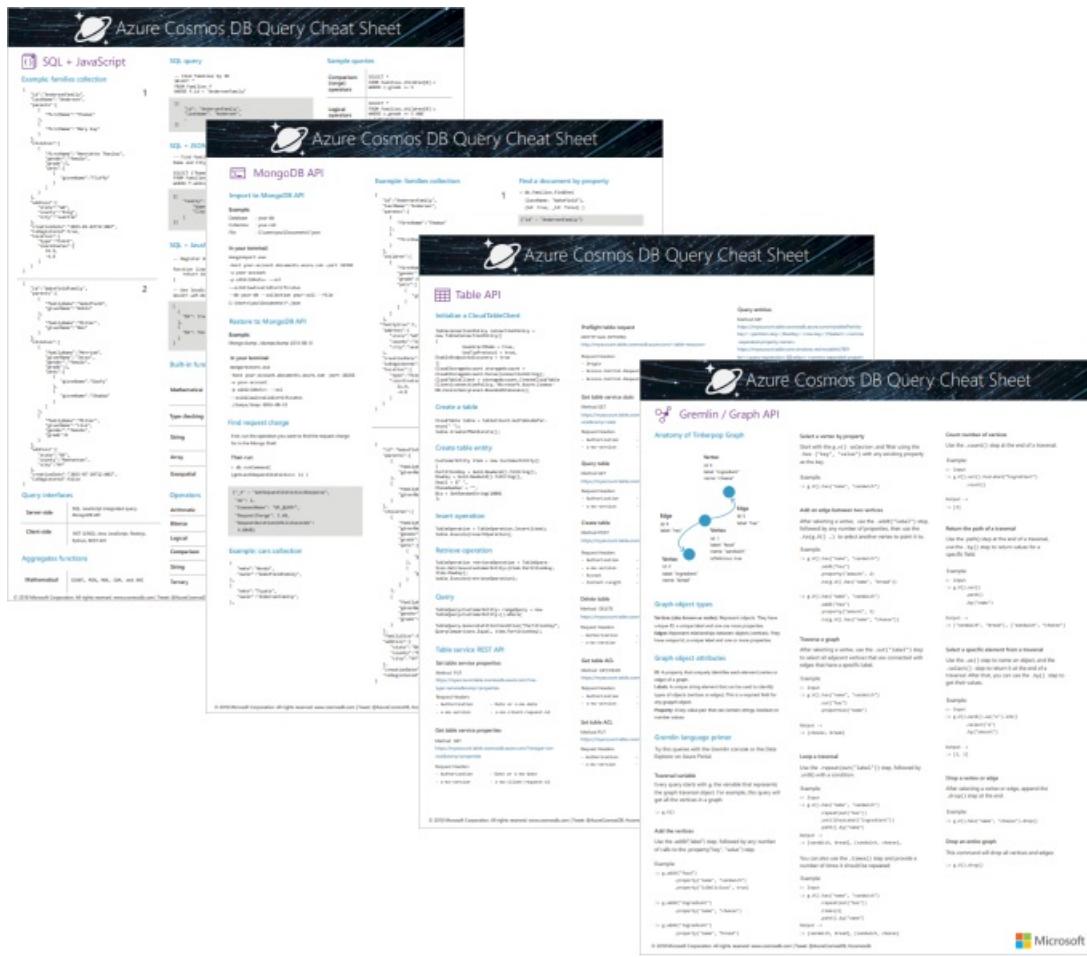
5/28/2019 • 2 minutes to read • [Edit Online](#)

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.

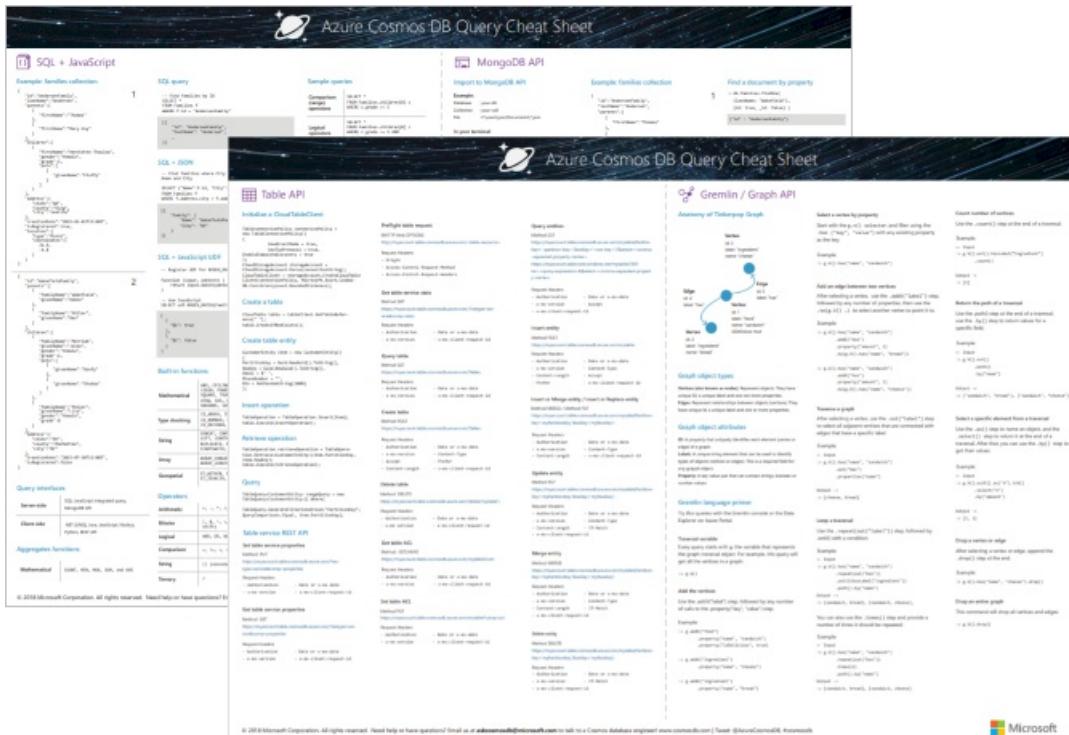
## Letter-sized cheat sheets

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

2/18/2020 • 43 minutes to read • [Edit Online](#)

## 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 GraphSON. 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:

- [MSDN forum](#)
- [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 Active Directory (AAD) accounts or accounts belonging to corporate AAD 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.
- 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 master key?

A master 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 master keys. The primary master key and secondary master 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 [throughput limits per container and database](#) articles for more information.

## 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, a container maps to 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.

## Azure Cosmos DB's API for MongoDB

### What is the Azure Cosmos DB's 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-master 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 SSL, so be sure to use TLSv1.2.

For more information, see [Connect to your Cosmos database with Azure Cosmos DB's API for MongoDB](#).

## Are there additional error codes that I need to deal with 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 <a href="#">Azure portal</a> .  Example: <pre>db.getCollection('users').aggregate([     {\$match: {name: "Andy"}},     {\$sort: {age: -1}} ])</pre>

## 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

# Table API

## How can I use the Table API offering?

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](#).

## 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](#).

## 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
GET, PUT	/? restype=service@comp=properties	<a href="#">Set Table Service Properties</a> and <a href="#">Get Table Service Properties</a>	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>	<a href="#">Pre-flight CORS table request</a>	This is part of CORS which Azure Cosmos DB doesn't currently support.
GET	/? restype=service@comp=stats	<a href="#">Get Table Service Stats</a>	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	<a href="#">Get Table ACL</a> and <a href="#">Set Table ACL</a>	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.

In addition 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	" "
TableServiceQuery	" "

## How do I provide feedback about the SDK or bugs?

You can share your feedback in any of the following ways:

- [User voice](#)
- [MSDN forum](#)
- [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.

## What is the connection string that I need to use to connect to the Table API?

The connection string is:

```
DefaultEndpointsProtocol=https;AccountName=<AccountNamefromCosmos DB;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 10-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.

## **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 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.

## **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.

### **How can I evaluate the efficiency of my 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.

## Cassandra API

**What is the protocol version supported by Azure Cosmso DB Cassandra API? Is there a plan to support other protocols?**

Apache Cassandra API for Azure Cosmos DB supports today CQL version 4. If you have feedback about

supporting other protocols, let us know via [user voice feedback](#) or send an email to [askcosmosdbcassandra@microsoft.com](mailto:askcosmosdbcassandra@microsoft.com).

### Why is choosing a throughput for a table a requirement?

Azure Cosmos DB sets default throughput for your container based on where you create the table from - portal or CQL. 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 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 distributed across the underlying physical partitions equally.

### What is the default RU/s of table when created through CQL? What If I need to change it?

Azure Cosmos DB uses request units per second (RU/s) as a currency for providing throughput. Tables created through CQL have 400 RU. You can change the RU from the 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 operation. This guarantee is possible when the engine can enforce governance on the tenant's operations. This is possible based on setting the throughput, which ensures that you get the guaranteed throughput and latency, because platform reserves this capacity and guarantees operation success. When you go over this capacity, you get overloaded error message indicating your capacity was used up. 0x1001 Overloaded: the request can't be processed because "Request Rate is large". At this juncture, it's essential to see what operations and their volume causes this issue. You can get an idea about consumed capacity going over the provisioned capacity with metrics on the portal. Then you need to ensure capacity is consumed nearly equally across all underlying partitions. If you see most of the throughput is consumed by one partition, you have skew of workload.

Metrics are available that show you how throughput is used over hours, 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 right location. In Azure Cosmos DB, it's used to find 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 take away here is that a logical partition shouldn't go over the 10-GB limit today.

### What happens when I get a quota full" notification indicating that a partition is full?

Azure Cosmos DB is a SLA-based system that 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.

The 10-GB limit on the number of entities or items per logical partition you should adhere to. 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 only come if your data is skewed: that is, you have lot of data for one partition key (more than 10 GB). You can find the distribution of data using the storage portal. Way to fix this error is to recreate the table and choose a granular primary (partition key), which allows better distribution of data.

### **Is it possible to use Cassandra API as key value store with millions or billions of individual partition keys?**

Azure Cosmos DB can store unlimited data by scaling out the storage. This is independent of the throughput. Yes you can always just use Cassandra API to store and retrieve key/values by specifying right primary/partition key. These individual keys get their own logical partition and sit atop physical partition without issues.

### **Is it possible to create more than one table with Apache Cassandra API of Azure Cosmos DB?**

Yes, it's possible to create more than one table with Apache Cassandra API. Each of those tables is treated as unit for throughput and storage.

### **Is it possible to create more than one table in succession?**

Azure Cosmos DB is resource governed system for both data and control plane activities. Containers like collections, tables are runtime entities that are provisioned for given throughput capacity. The creation of these containers in quick succession isn't expected activity and throttled. If you have tests that drop/create tables immediately, try to space them out.

### **What is maximum number of tables that can be created?**

There's no physical limit on number of tables, send an email at [askcosmosdbcassandra@microsoft.com](mailto:askcosmosdbcassandra@microsoft.com) if you have large number of tables (where the total steady size goes over 10 TB of data) that need to be created from usual 10s or 100s.

### **What is the maximum # of keyspace that we can create?**

There's no physical limit on number of keyspaces as they're metadata containers, send an email at [askcosmosdbcassandra@microsoft.com](mailto:askcosmosdbcassandra@microsoft.com) if you have large number of keyspaces for some reason.

### **Is it possible to bring in lot of data after starting from normal table?**

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.

### **Is it possible to supply yaml file settings to configure Apache Cassandra API of Azure Cosmos DB behavior?**

Apache Cassandra API of Azure Cosmos DB is a platform service. It 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 multitude of other settings. Azure Cosmos DB's Apache Cassandra API focuses on providing read and write performance that you require without the overhead of configuration and management.

### **Will Apache Cassandra API for Azure Cosmos DB support node addition/cluster status/node status commands?**

Apache Cassandra API is a platform service that makes capacity planning, responding to the elasticity demands for throughput & storage a breeze. With Azure Cosmos DB you provision throughput, you need. Then you can scale it up and down any number of times through the day without worrying about adding/deleting nodes or managing them. This implies you don't need to use the node, cluster management tool too.

### **What happens with respect to various config 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 setup replicas or other things. All writes are always durably quorum committed in any region where you write while providing performance guarantees.

**What happens with respect to various settings for table metadata like bloom filter, caching, read repair change, gc\_grace, compression memtable\_flush\_period, and more?**

Azure Cosmos DB provides performance for reads/writes and throughput without need for touching any of the configuration settings and accidentally manipulating them.

**Is time-to-live (TTL) supported for Cassandra tables?**

Yes, TTL is supported.

**Is it possible to monitor node status, replica status, gc, and OS parameters earlier with various tools? What needs to be monitored now?**

Azure Cosmos DB is a platform service that helps you increase productivity and not worry about managing and monitoring infrastructure. You just need to take care of throughput that's available on portal metrics to find if you're getting throttled and increase or decrease that throughput. Monitor [SLAs](#). Use [Metrics](#) Use [Diagnostic logs](#).

**Which client SDKs can work with Apache Cassandra API of Azure Cosmos DB?**

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](mailto:askcosmosdbcassandra@microsoft.com).

**Is composite partition key supported?**

Yes, you can use regular syntax to create composite partition key.

**Can I use sstableloader for data loading?**

No, sstableloader isn't supported.

**Can an on-premises Apache Cassandra cluster be paired with Azure Cosmos DB's Cassandra API?**

At present Azure Cosmos DB has an optimized experience for cloud environment without overhead of operations. If you require pairing, send mail to [askcosmosdbcassandra@microsoft.com](mailto:askcosmosdbcassandra@microsoft.com) with a description of your scenario. We are working on offering to help pair the on-premises/different cloud Cassandra cluster to Cosomos DB's Cassandra API.

**Does Cassandra API provide full backups?**

Azure Cosmos DB provides two free full backups taken at four hours interval today across all APIs. This ensures you don't need to set up a backup schedule and other things. If you want to modify retention and frequency, send an email to [askcosmosdbcassandra@microsoft.com](mailto: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 Azure Cosmos DB 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 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.

**Does the Apache Cassandra API index all attributes of an entity by default?**

Cassandra API is planning to support Secondary indexing to help create selective index on certain attributes.

**Can I use the new Cassandra API SDK locally with the emulator?**

Yes this is supported.

**Azure Cosmos DB as a platform seems to have lot of capabilities, such as change feed and other functionality. Will these capabilities be added to the Cassandra API?**

The Apache Cassandra API provides the same CQL functionality as Apache Cassandra. We do plan to look into

feasibility of supporting various capabilities in future.

**Feature x of regular Cassandra API isn't working as today, where can the feedback be provided?**

Provide feedback via [user voice feedback](#).

# Azure Cosmos DB whitepapers

12/5/2019 • 2 minutes to read • [Edit Online](#)

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
<a href="#">Schema-Agnostic Indexing with Azure Cosmos DB</a>	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.
<a href="#">Azure Cosmos DB and personal data</a>	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

2/4/2020 • 2 minutes to read • [Edit Online](#)

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.

## Migration tools

PARTNER	CAPABILITIES & EXPERIENCE	SUPPORTED COUNTRIES/REGIONS	CONTACT
	Move real-time data to Azure Cosmos DB from a wide variety of data sources. Striim simplifies the real-time collection and movement of data from a wide variety of on-premises sources, including enterprise document and relational databases, sensors, and log files into Azure Cosmos DB.	USA	<a href="#">Website</a>

## Systems Integrator partners

PARTNER	CAPABILITIES & EXPERIENCE	SUPPORTED COUNTRIES/REGIONS	CONTACT
	NoSQL migration; New app development	USA	<a href="#">Website</a>
	NoSQL migration, App innovation (existing apps), New app development	US, Norway, Finland, Belarus, Argentina	<a href="#">Website</a>
	NoSQL migration, App innovation (existing apps), New app development	USA	<a href="#">Website</a>
	New app development, App innovation (existing apps)	Austria, Germany, Switzerland, Italy, Norway, Spain, UK	<a href="#">Website</a>

Partner	Capabilities & Experience	Supported Countries/Regions	Contact
 A division of Insight	NoSQL migration, App innovation (existing apps), New app development	North America, Asia-Pacific	<a href="#">Website</a>
	NoSQL migration, App innovation (existing apps), New app development	USA	<a href="#">Website</a>
	NoSQL migration; App innovation (existing apps); New app development	US, France, UK, Netherlands, Finland	<a href="#">Website</a>
	App innovation (existing apps), New app development	UK	<a href="#">Website</a>
	NoSQL migration	US, Canada, UK, Denmark, Netherlands, Switzerland, Australia, Japan	<a href="#">Website</a>
	NoSQL migration	Global	<a href="#">Website</a>
	NoSQL migration, New app development	Argentina, Chile, Colombia, Mexico	<a href="#">Website</a>
	NoSQL migration, New app development	Brazil	<a href="#">Website</a>
	App development (new apps)	USA	<a href="#">Website</a>
	NoSQL migration	USA	<a href="#">Website</a>
	NoSQL migration, App innovation (existing apps)	UK	<a href="#">Website</a>
	NoSQL migration	US	<a href="#">Website</a>

PARTNER	CAPABILITIES & EXPERIENCE	SUPPORTED COUNTRIES/REGIONS	CONTACT
	NoSQL migration	Croatia, Sweden, Denmark, Ireland, Bulgaria, Slovenia, Cyprus, Malta, Lithuania, the Czech Republic, Iceland and Switzerland and Liechtenstein	<a href="#">Website</a>
	NoSQL migration	Ireland	<a href="#">Website</a>
	NoSQL migration	Portugal	<a href="#">Website</a>
	NoSQL migration, App innovation (existing apps), New app development	USA	<a href="#">Website</a>
	App innovation (existing apps), New app development	US, UK, France, Malaysia, Denmark, Norway, Sweden	<a href="#">Website</a>
	NoSQL migration	USA	<a href="#">Website</a>
	NoSQL migration, New app development	Germany	<a href="#">Website</a>
	New app development	Portugal, UK	<a href="#">Website</a>

## Next steps

To learn more about some of Microsoft's other partners, see the [Microsoft Partner site](#).

# Understand multi-master benefits in Azure Cosmos DB

12/5/2019 • 2 minutes to read • [Edit Online](#)

Cosmos DB account operators should choose appropriate global distribution configuration to ensure the latency, availability and RTO requirements for their applications. Azure Cosmos accounts configured with multiple write locations offer significant benefits over accounts with single write location including, 99.999% write availability SLA, <10 ms write latency SLA at the 99th percentile and RTO = 0 in a regional disaster.

## Comparison of features

APPLICATION REQUIREMENT	MULTIPLE WRITE LOCATIONS	SINGLE WRITE LOCATION	NOTE
Write latency SLA of <10ms at P99	<b>Yes</b>	No	Accounts with Single Write Location incur additional cross-region network latency for each write.
Read latency SLA of <10ms at P99	<b>Yes</b>	Yes	
Write SLA of 99.999%	<b>Yes</b>	No	Accounts with Single Write Location guarantee SLA of 99.99%
RTO = 0	<b>Yes</b>	No	Zero down time for writes in case of regional disasters. Accounts with single write location have RTO of 15 min.

## Next Steps

If you would still like to disable `EnableMultipleWriteLocations` in your Azure Cosmos account, please [open a support ticket](#).

# Common Azure Cosmos DB use cases

11/8/2019 • 8 minutes to read • [Edit Online](#)

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 globally distributed database service. 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.

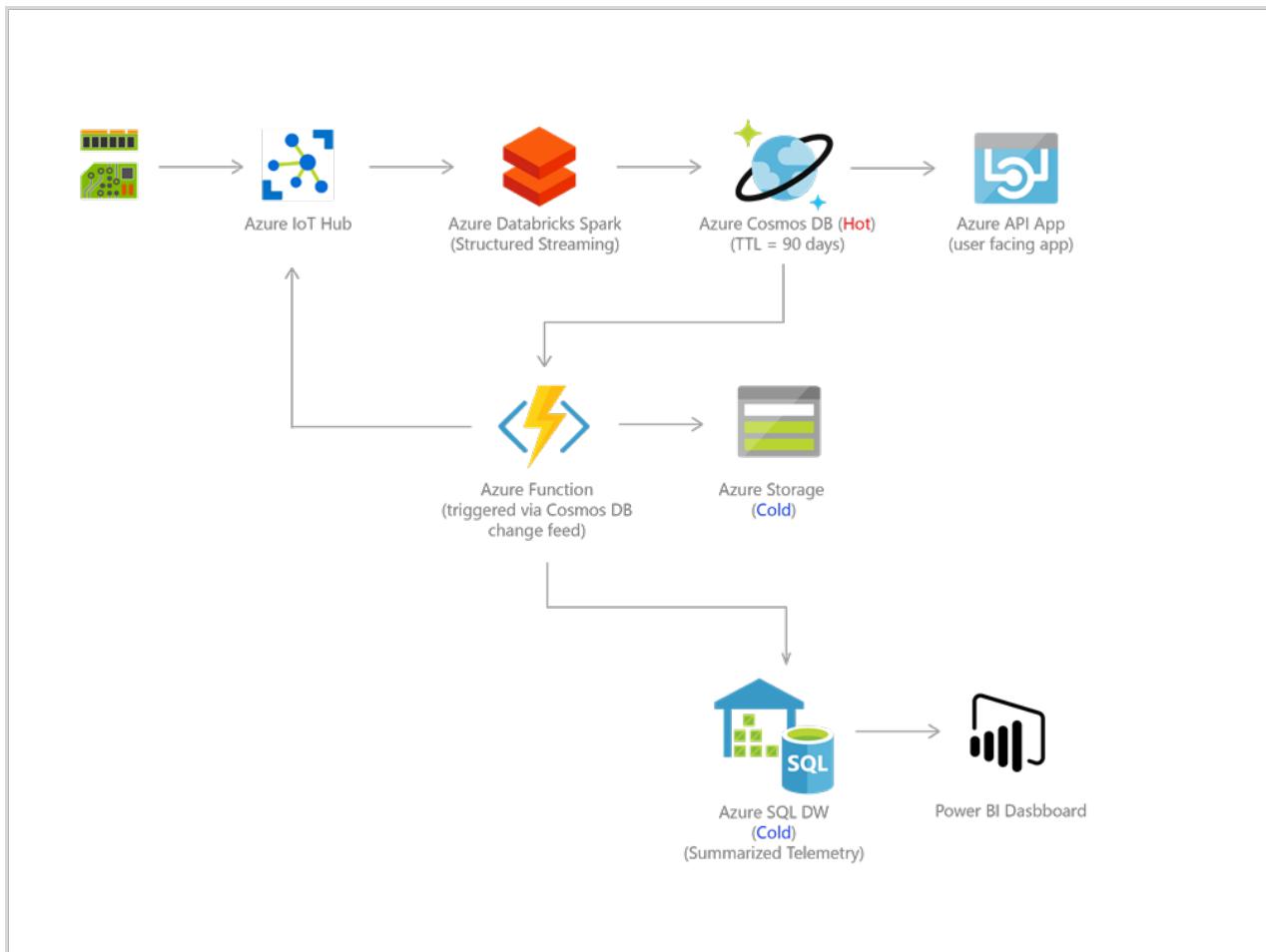
- Azure Cosmos DB natively partitions your data for high availability and scalability. Azure Cosmos DB offers 99.99% guarantees for availability, throughput, low latency, and consistency on all single-region accounts and all multi-region accounts with relaxed consistency, and 99.999% read availability 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](#).

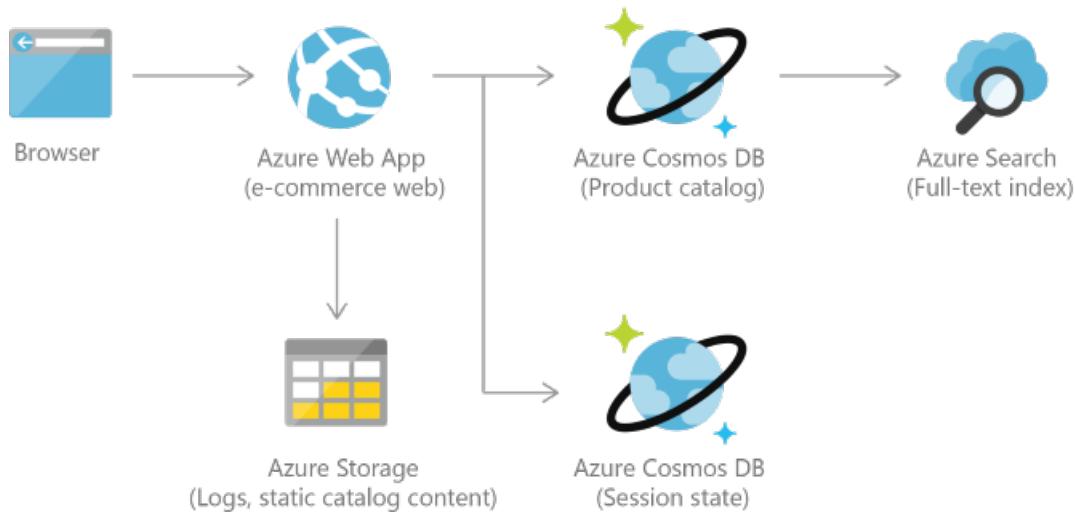
## Retail and marketing

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.

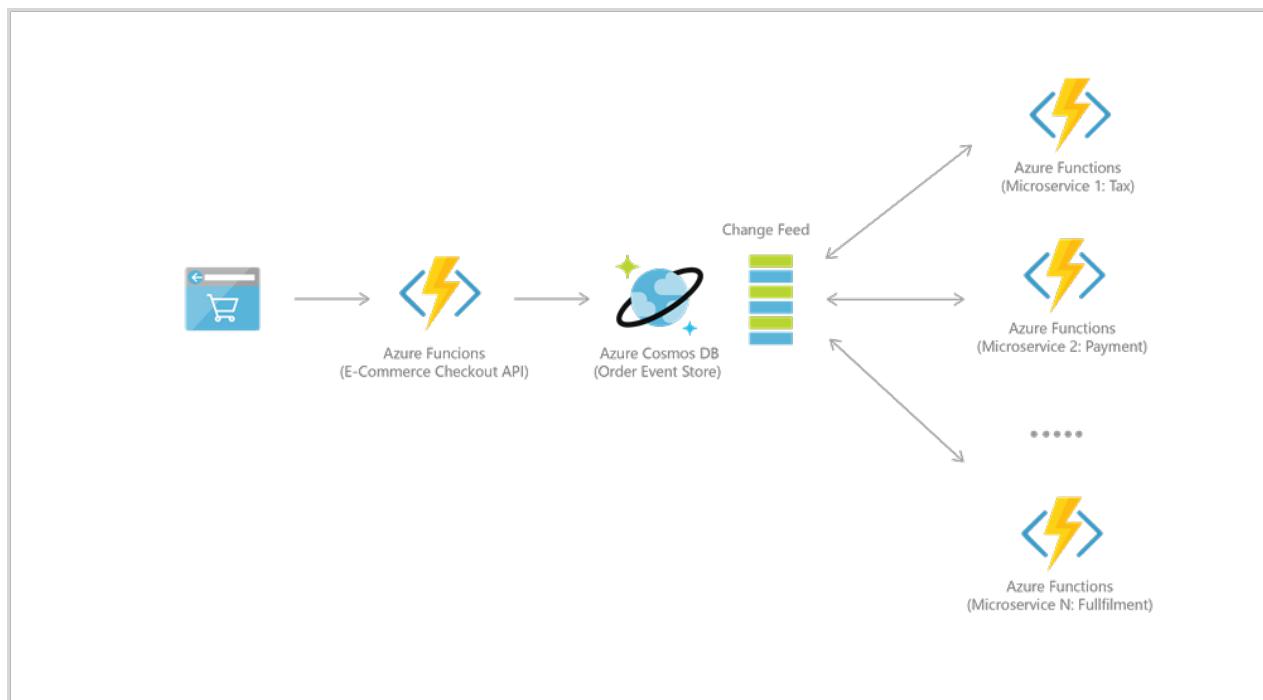
Catalog data usage scenarios involve storing and querying a set of attributes for entities such as people, places, and products. Some examples of catalog data are user accounts, product catalogs, IoT device registries, and bill of materials systems. Attributes for this data may vary and can change over 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](#).

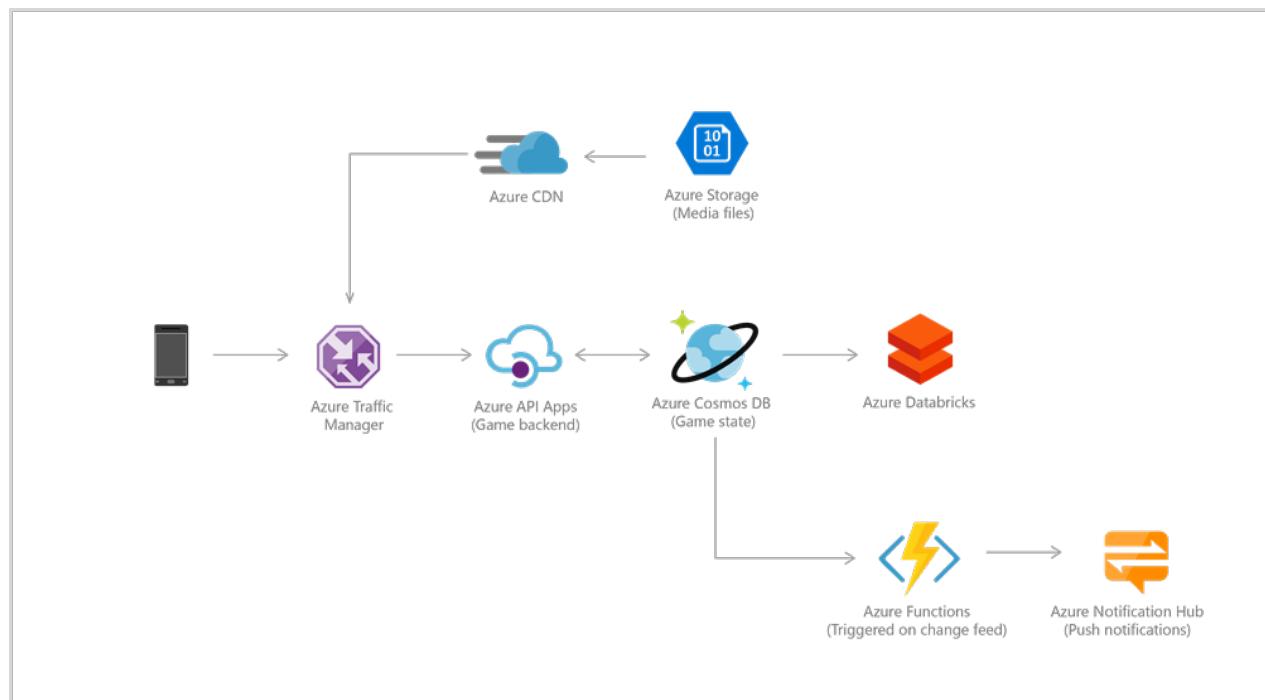
## Gaming

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

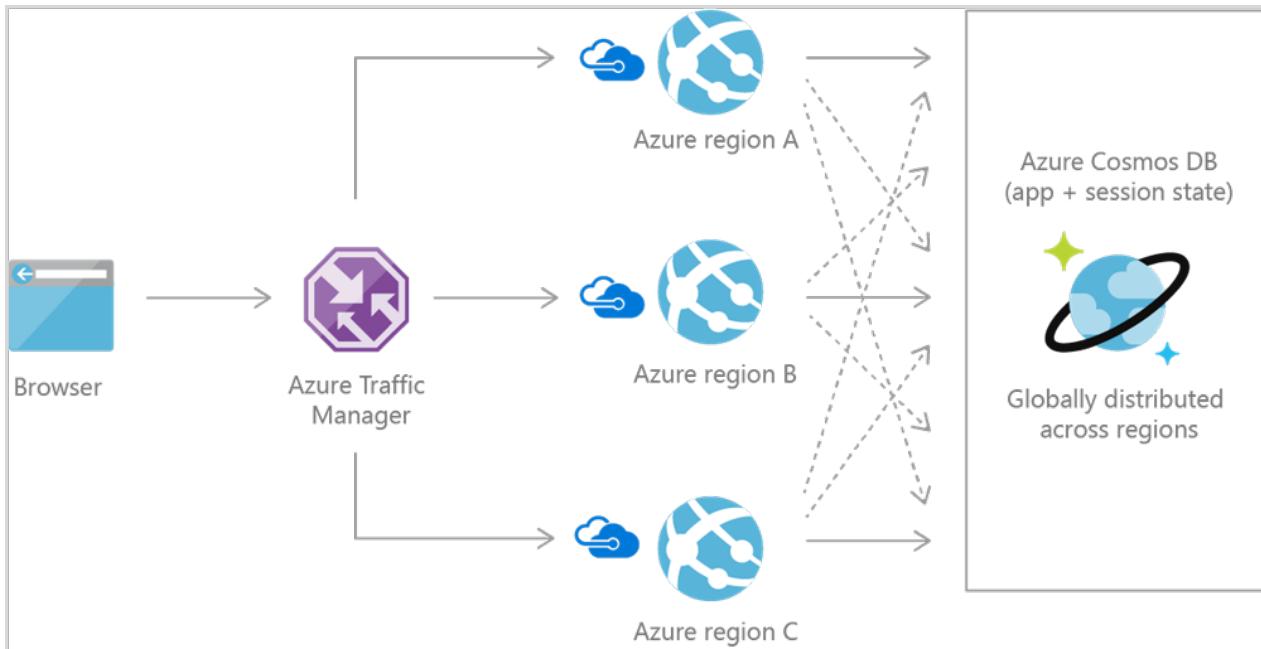
Azure Cosmos DB is commonly used within web and mobile applications, and is well suited for modeling social interactions, integrating with third-party services, and for building rich personalized experiences. The Cosmos DB SDKs can be used to build rich iOS and Android applications using the popular [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.

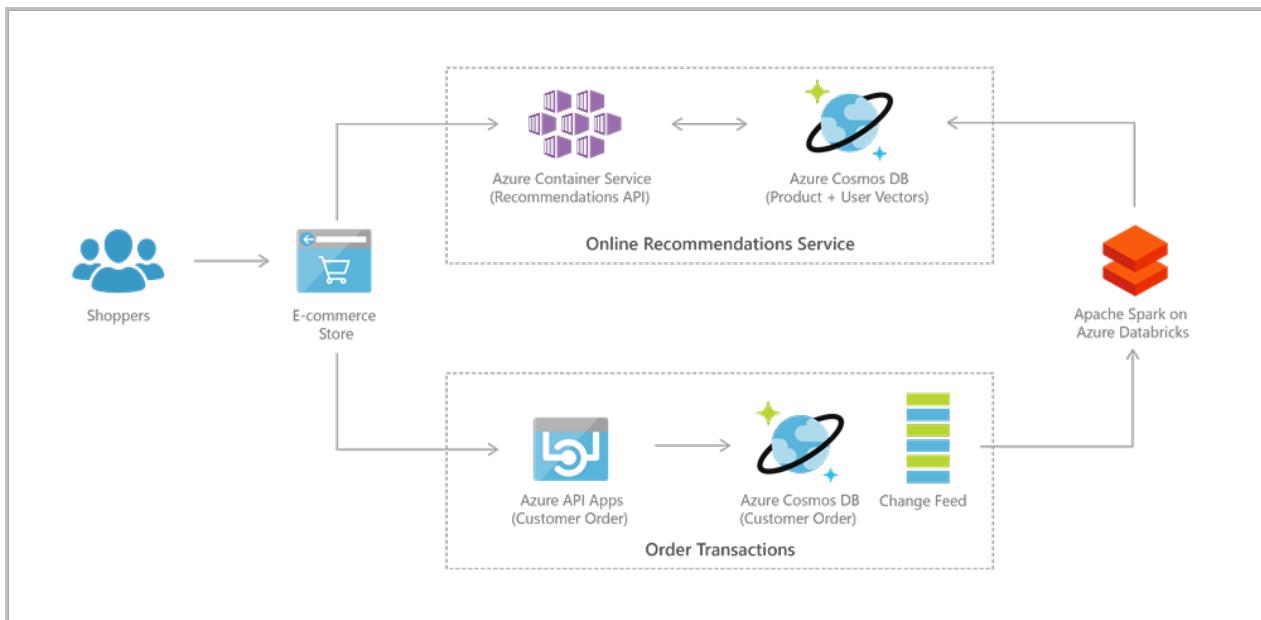
Many of the social applications run at global scale and can exhibit unpredictable usage patterns. Flexibility in scaling the data store is essential as the application layer scales to match usage demand. You can scale out by adding additional data partitions under a Cosmos DB account. In addition, you 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

11/7/2019 • 13 minutes to read • [Edit Online](#)

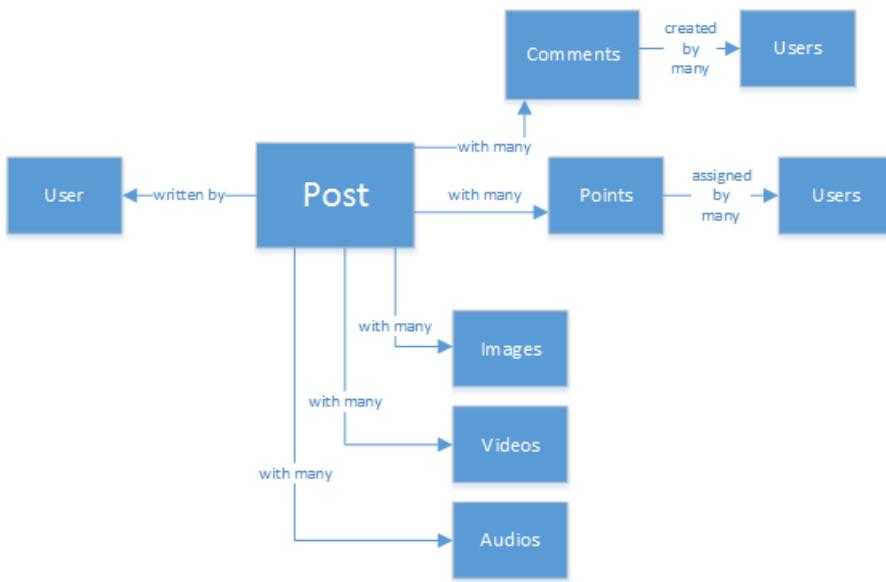
Living in a massively interconnected society means that, at some point in life, you become part of a **social network**. You use social networks to keep in touch with friends, colleagues, family, or sometimes to share your passion with people with common interests.

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 [vertices](#) 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](#).

\* STORAGE CAPACITY (up to 10TB and more) ?

10 GB	250 GB	Custom
-------	--------	--------

THROUGHPUT CAPACITY (RU/s) ?

 - +

Between 2500 and 250000. You can provision lower or higher throughput capacity via support request. [Click here](#).

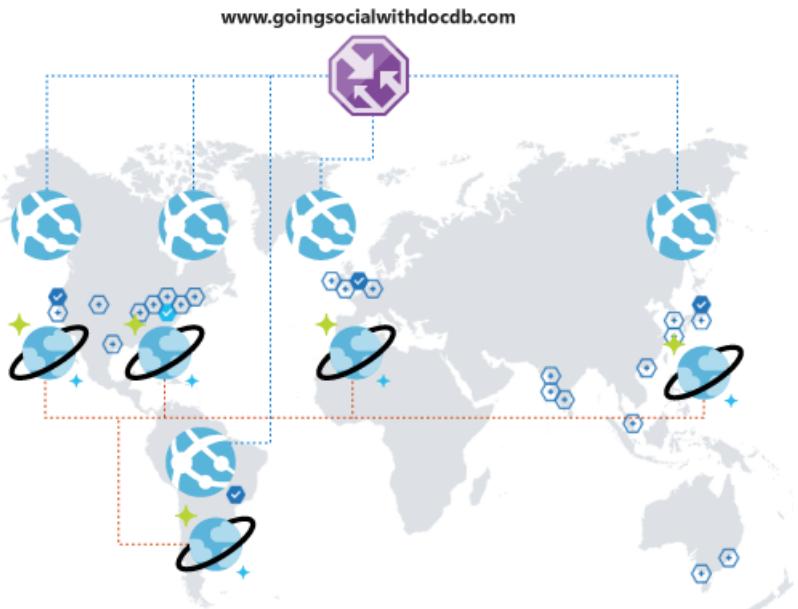
Estimated hourly spend \$0.80USD

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!

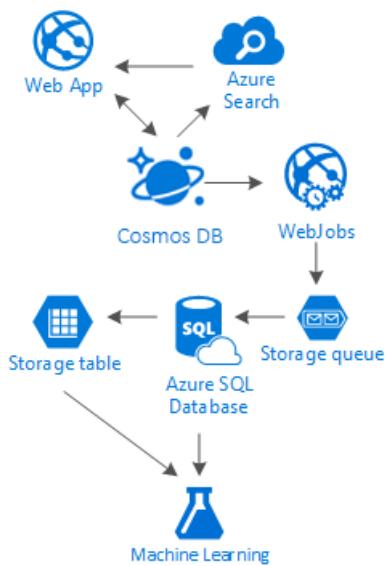
Cosmos DB lets you [replicate your data globally](#) and transparently with a couple of clicks and automatically select among the available regions from your [client code](#). This process also means that you can have [multiple failover regions](#).

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](#).