# Azure Cosmos DB

## Key benefits of Azure Cosmos DB

Azure Cosmos DB is designed to provide low latency, elastic scalability of throughput, well-defined semantics for data consistency, and high availability.

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.

### Key benefits of global distribution

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

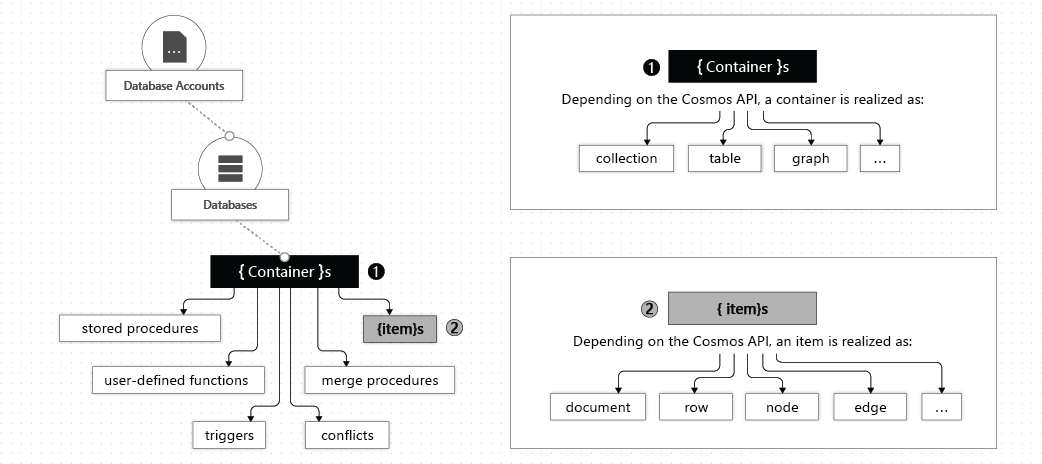
* Unlimited elastic write and read scalability.
* 99.999% read and write availability all around the world.
* Guaranteed reads and writes served in less than 10 milliseconds at the 99th percentile.

## Explore the resource hierarchy

The Azure Cosmos account is the fundamental unit of global distribution and high availability. Your Azure Cosmos account contains a unique DNS name and you can manage an account by using the Azure portal or the Azure CLI, or by using different language-specific SDKs. For globally distributing your data and throughput across multiple Azure regions, you can add and remove Azure regions to your account at any time.

## Elements in an Azure Cosmos account

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



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

Currently, you can create a maximum of 50 Azure Cosmos accounts under an Azure subscription (this is a soft limit that can be increased via support request). After you create an account under your Azure subscription, you can manage the data in your account by creating databases, containers, and items.

### 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 |

### Azure Cosmos containers

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

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

* **Dedicated provisioned throughput mode**: The throughput provisioned on a container is exclusively reserved for that container and it is backed by the SLAs.
* **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.

### Azure Cosmos items

Depending on which API you use, an Azure Cosmos item can represent either a document in a collection, a row in a table, or a node or edge in a graph.

The following table shows the mapping of API-specific entities to an Azure Cosmos item:

| **Cosmos entity** | **SQL API** | **Cassandra API** | **Azure Cosmos DB API for MongoDB** | **Gremlin API** | **Table API** |
| --- | --- | --- | --- | --- | --- |
| Azure Cosmos item | Item | Row | Document | Node or edge | Item |

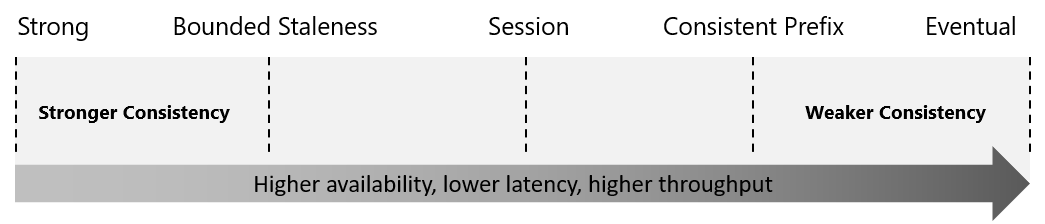
## Consistency levels

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*
* *eventual*

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



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

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.

## Choose the right consistency level

Azure Cosmos DB allows developers to choose among the five consistency models: strong, bounded staleness, session, consistent prefix and eventual. Each of these consistency models can be used for specific real-world scenarios. Each provides 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.
* 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 throughput 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.

### Cassandra, MongoDB, and Gremlin APIs

Azure Cosmos DB provides native support for wire protocol-compatible APIs for popular databases. These include MongoDB, Apache Cassandra, and Gremlin. When using Gremlin API the default consistency level configured on the Azure Cosmos account is used. For details on consistency level mapping between Cassandra API or the API for MongoDB and Azure Cosmos DB's consistency levels see, [Cassandra API consistency mapping](https://docs.microsoft.com/en-us/azure/cosmos-db/cassandra/apache-cassandra-consistency-mapping) and [API for MongoDB consistency mapping](https://docs.microsoft.com/en-us/azure/cosmos-db/mongodb/consistency-mapping).

### 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.**

Probabilistic bounded staleness shows how eventual your eventual consistency is. 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.

## Supported APIs

These APIs allow your applications to treat Azure Cosmos DB as if it were various other databases technologies, without the overhead of management, and scaling approaches.

### Core(SQL) API

This API stores data in document format. It offers the best end-to-end experience as we have full control over the interface, service, and the SDK client libraries. Any new feature that is rolled out to Azure Cosmos DB is first available on SQL API accounts. Azure Cosmos DB SQL API accounts provide support for querying items using the Structured Query Language (SQL) syntax, one of the most familiar and popular query languages.

If you are migrating from other databases such as Oracle, DynamoDB, HBase etc. SQL API is the recommended option. SQL API supports analytics and offers performance isolation between operational and analytical workloads.

### API for MongoDB

This API stores data in a document structure, via BSON format. It is compatible with MongoDB wire protocol; however, it does not use any native MongoDB related code. This API is a great choice if you want to use the broader MongoDB ecosystem and skills, without compromising on using Azure Cosmos DB’s features such as scaling, high availability, geo-replication, multiple write locations, automatic and transparent shard management, transparent replication between operational and analytical stores, and more. You can use your existing MongoDB apps with API for MongoDB by just changing the connection string.

API for MongoDB is compatible with the 4.0, 3.6, and 3.2 MongoDB server versions. Server version 4.0 is recommended as it offers the best performance and full feature support

### Cassandra API

This API stores data in column-oriented schema. Cassandra API is wire protocol compatible with the Apache Cassandra. You should consider Cassandra API if you want to benefit the elasticity and fully managed nature of Azure Cosmos DB and still use most of the native Apache Cassandra features, tools, and ecosystem. This means on Cassandra API you don’t need to manage the OS, Java VM, garbage collector, read/write performance, nodes, clusters, etc.

You can use Apache Cassandra client drivers to connect to the Cassandra API. The Cassandra API enables you to interact with data using the Cassandra Query Language (CQL), and tools like CQL shell, Cassandra client drivers that you're already familiar with. Cassandra API currently only supports OLTP scenarios. Using Cassandra API, you can also use the unique features of Azure Cosmos DB such as change feed.

### Table API

This API stores data in key/value format. If you are currently using Azure Table storage, you may see some limitations in latency, scaling, throughput, global distribution, index management, low query performance. Table API overcomes these limitations and it’s recommended to migrate your app if you want to use the benefits of Azure Cosmos DB. Table API only supports OLTP scenarios.

Applications written for Azure Table storage can migrate to the Table API with little code changes and take advantage of premium capabilities.

### Gremlin API

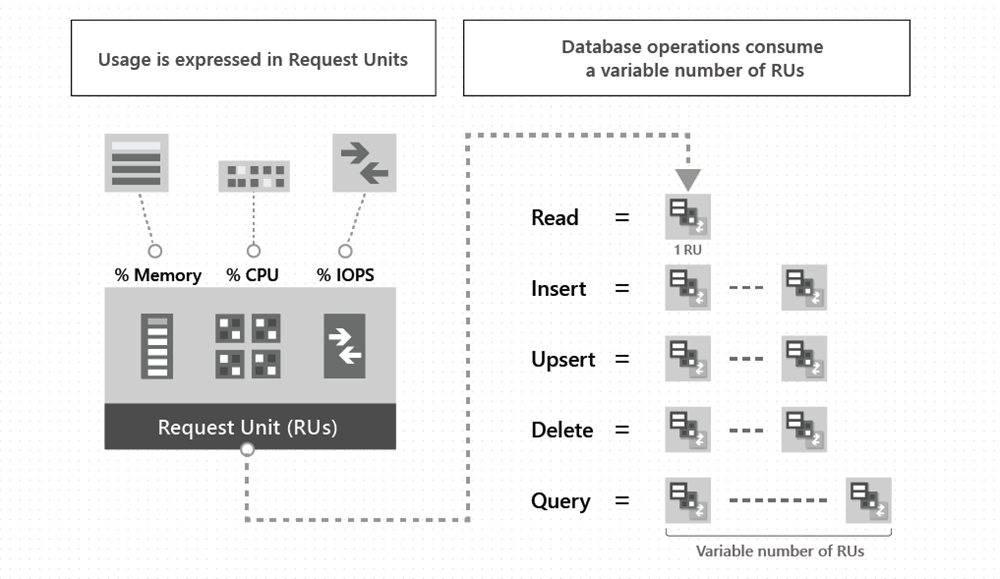
This API allows users to make graph queries and stores data as edges and vertices. Use this API for scenarios involving dynamic data, data with complex relations, data that is too complex to be modeled with relational databases, and if you want to use the existing Gremlin ecosystem and skills. Gremlin API currently only supports OLTP scenarios.

## Request units

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.

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

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



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

1. Provisioned throughput mode:

In this mode, you provision the number of RUs for your application on a per-second basis in increments of 100 RUs per second. To scale the provisioned throughput for your application, you can increase or decrease the number of RUs at any time in increments or decrements of 100 RUs. You can make your changes either programmatically or by using the Azure portal. You can provision throughput at container and database granularity level.

1. Serverless mode:

In this mode, you don't have to provision any throughput when creating resources in your Azure Cosmos account. At the end of your billing period, you get billed for the amount of request units that has been consumed by your database operations.

1. Autoscale mode:

In this mode, you can automatically and instantly scale the throughput (RU/s) of your database or container based on its usage. This mode is well suited for mission-critical workloads that have variable or unpredictable traffic patterns, and require SLAs on high performance and scale.

## Create Azure Cosmos DB resources by using the Azure portal

### Create an Azure Cosmos DB account

Azure portal -> [+ Create a resource] -> Azure Cosmos DB

-> Create Azure Cosmos DB -> Select API option -> Create in the Core (SQL) – Recommended

-> Core (SQL) page -> basic settings ->

“Subscription”: Select the subscription you want to use.

“Resource Group”: Select Create new, then enter az204-cosmos-rg.

“Account Name”: Enter a unique name to identify your Azure Cosmos account. The name can only contain lowercase letters, numbers, and the hyphen (-) character. It must be between 3-31 characters in length.

“Location”: Use the location that is closest to your users to give them the fastest access to the data.

“Capacity mode”: Select Serverless.

-> Review + create -> Create.

It takes a few minutes to create the account. Wait for the portal page to display Your deployment is complete.

-> Go to resource

### Add a database and a container

You can use the Data Explorer in the Azure portal to create a database and container.

Your Azure Cosmos DB account page -> Left menu -> Data Explorer -> [+New Container]

-> Add container ->

“Database ID” -> Create new -> *ToDoList*.

“Container ID” Enter *Items*

“Partition key” Enter */category*. The samples in this demo use */category* as the partition key.

->OK.

The Data Explorer displays the new database and the container that you created.

### Add data to your database

Data Explorer-> ToDoList database -> Items container -> Items -> [+ New Item]

Add the following structure to the item on the right side of the Items pane:

{

"id": "1",

"category": "personal",

"name": "groceries",

"description": "Pick up apples and strawberries.",

"isComplete": false

}

-> Save

Your items can have any structure, because Azure Cosmos DB doesn't impose any schema on your data.

# Partitioning in Azure Cosmos DB

In partitioning, the items in a container are divided into distinct subsets called logical partitions.

## Logical partitions

Logical partitions are formed based on the value of a partition key that is associated with each item in a container. All the items in a logical partition have the same partition key value.

In addition to a partition key that determines the item's logical partition, each item in a container has an item ID which is 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.**

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. You don't have to worry about deleting a logical partition when the underlying data is deleted.

## Physical partitions

A container is scaled by distributing data and throughput across physical partitions. Internally, one or more logical partitions are mapped to a single physical partition. Typically smaller containers have many logical partitions but they only require a single physical partition. Unlike logical partitions, physical partitions are an internal implementation of the system and they are entirely managed by Azure Cosmos DB.

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

* The number of throughput provisioned (each individual physical partition can provide a throughput of up to 10,000 request units per second). The 10,000 RU/s limit for physical partitions implies that logical partitions also have a 10,000 RU/s limit, as each logical partition is only mapped to one physical partition.
* The total data storage (each individual physical partition can store up to 50GB data).

### Note:

Physical partitions are an internal implementation of the system and they are entirely managed by Azure Cosmos DB. **When developing your solutions, don't focus on physical partitions** because you can't control them. Instead, **focus on your partition keys**.

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

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

## Choose a partition key

The following are the two partition key components:

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

Selecting your partition key is a simple but important design choice in Azure Cosmos DB.

Once you select your partition key, it is not possible to change it in-place. If you need to change your partition key, you should move your data to a new container with your new desired partition key.

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

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

### Partition keys for read-heavy containers

**For large read-heavy containers** you might want to choose a partition key that appears frequently as a filter in your queries. Queries can be [efficiently routed to only the relevant physical partitions](https://docs.microsoft.com/en-us/azure/cosmos-db/how-to-query-container#in-partition-query) by including the partition key in the filter predicate.

If most of your workload's requests are queries and most of your queries have an equality filter on the same property, this property can be a good partition key choice.

**However, if your container is small**, you probably don't have enough physical partitions to need to worry about the performance impact of cross-partition queries. If your container could grow to more than a few physical partitions, then you should make sure you pick a partition key that minimizes cross-partition queries.

Your container will require more than a few physical partitions when either of the following are true:

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

### Using item ID as the partition key

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

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

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

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

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

* If the *item ID* is the partition key, it will become a unique identifier throughout your entire container. You won't be able to have items that have a duplicate *item ID*.
* If you have a read-heavy container that has a lot of [physical partitions](https://docs.microsoft.com/en-us/azure/cosmos-db/partitioning-overview#physical-partitions), queries will be more efficient if they have an equality filter with the *item ID*.
* You can't run stored procedures or triggers across multiple logical partitions.

## Create a synthetic partition key

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 unit 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,

"partitionKey": "abc-123-2018"

}

### 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.

### 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.**

**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.**

**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.**

# Work with Azure Cosmos DB

## Create a database & container with .Net

### NuGet package used:

**Microsoft.Azure.Cosmos**

[Find the code here](https://github.com/AjinkyaApte88/General/tree/main/mslearn-cosmos-db).

## Create stored procedures

Azure Cosmos DB provides language-integrated, transactional execution of JavaScript that lets you write **stored procedures**, **triggers**, and **user-defined functions (UDFs)**.

**To call a stored procedure, trigger, or 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**](https://docs.microsoft.com/en-us/azure/cosmos-db/sql/how-to-use-stored-procedures-triggers-udfs)**.**

### Writing stored procedures

Stored procedures can create, update, read, query, and delete items inside an Azure Cosmos container. Stored procedures are registered per collection, and can operate on any document or an attachment present in that collection.

Here is a simple stored procedure that returns a "Hello World" response.

var helloWorldStoredProc = {

id: "helloWorld",

serverScript: function () {

var context = getContext();

var response = context.getResponse();

response.setBody("Hello, World");

}

}

The context object provides access to all operations that can be performed in Azure Cosmos DB, as well as access to the request and response objects. In this case, you use the response object to set the body of the response to be sent back to the client.

### Create an item using stored procedure

When you create an item by using stored procedure it 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:

* The error object in case the operation fails
* A return value

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 following example stored procedure takes an input parameter named documentToCreate and the parameter’s value is the body of a document to be created in the current collection. The callback throws an error if the operation fails. Otherwise, it sets the id of the created document as the body of the response to the client.

function createSampleDocument(documentToCreate) {

var context = getContext();

var collection = context.getCollection();

var accepted = collection.createDocument(

collection.getSelfLink(),

documentToCreate,

function (error, documentCreated) {

context.getResponse().setBody(documentCreated.id)

}

);

if (!accepted) return;

}

### Arrays as input parameters for stored procedures

When defining a stored procedure in the 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);

});

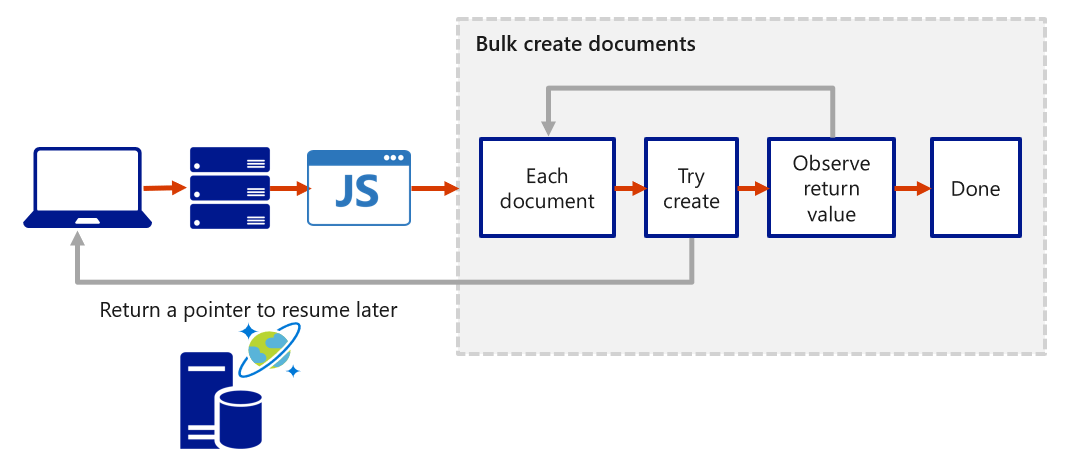
}

### Bounded execution

All Azure Cosmos DB operations must complete within a limited amount of time. Stored procedures have a limited amount of time to run on the server. All collection functions return a Boolean value that represents whether that operation will complete or not.

### Transactions within stored procedures

You can implement transactions on items within a container by using a stored procedure. JavaScript functions can implement a continuation-based model to batch or resume execution. The continuation value can be any value of your choice and your applications can then use this value to resume a transaction from a new starting point. The diagram below depicts how the transaction continuation model can be used to repeat a server-side function until the function finishes its entire processing workload.



## Create triggers and user-defined functions

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. Triggers are not automatically executed, they must be specified for each database operation where you want them to execute. After you define a trigger, you should register it by using the Azure Cosmos DB SDKs.

**For examples of how to register and call a trigger, see**[**pre-triggers**](https://docs.microsoft.com/en-us/azure/cosmos-db/sql/how-to-use-stored-procedures-triggers-udfs#pre-triggers)**and**[**post-triggers**](https://docs.microsoft.com/en-us/azure/cosmos-db/sql/how-to-use-stored-procedures-triggers-udfs#post-triggers)**.**

### 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, it adds 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 is not permitted.

### Post-triggers

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.

### 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 example assume there is a container called "Incomes" with properties as follows:

{

"name": "User One",

"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;

}