100 XP

Introduction

3 minutes

This module is an introduction to both client and server-side programming on Azure Cosmos DB.

After completing this module, you'll be able to:

- Identify classes and methods used to create resources
- Create resources by using the Azure Cosmos DB .NET v3 SDK
- Write stored procedures, triggers, and user-defined functions by using JavaScript

Next unit: Explore Microsoft .NET SDK v3 for Azure Cosmos DB

Continue

How are we doing? ☆ ☆ ☆ ☆ ☆

100 XP

Explore Microsoft .NET SDK v3 for Azure Cosmos DB

3 minutes

This unit focuses on version 3 of the .NET SDK. (**Microsoft.Azure.Cosmos** NuGet package.) If you're familiar with the previous version of the .NET SDK, you may be used to the terms collection and document.

The azure-cosmos-dotnet-v3 GitHub repository includes the latest .NET sample solutions. You use these solutions to perform CRUD (create, read, update, and delete) and other common operations on Azure Cosmos DB resources.

Because Azure Cosmos DB supports multiple API models, version 3 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.

Below are examples showing some of the key operations you should be familiar with. For more examples please visit the GitHub link above. The examples below all use the async version of the methods.

CosmosClient

Creates a new `CosmosClient` with a connection string. `CosmosClient` is thread-safe. Its recommended to maintain a single instance of `CosmosClient` per lifetime of the application which enables efficient connection management and performance.

C# Copy

```
CosmosClient client = new CosmosClient(endpoint, key);
```

Database examples

Create a database

The `CosmosClient.CreateDatabaseIfNotExistsAsync` checks if a database exists, and if it doesn't, creates it. Only the database `id` is used to verify if there is an existing database.

C# Copy

```
// An object containing relevant information about the response
DatabaseResponse databaseResponse = await
client.CreateDatabaseIfNotExistsAsync(databaseId, 10000);
```

Read a database by ID

Reads a database from the Azure Cosmos service as an asynchronous operation.

C# Copy

```
DatabaseResponse readResponse = await database.ReadAsync();
```

Delete a database

Delete a Database as an asynchronous operation.

C# Copy

```
await database.DeleteAsync();
```

Container examples

Create a container

The `Database.CreateContainerIfNotExistsAsync` method checks if a container exists, and if it doesn't, it creates it. Only the container `id` is used to verify if there is an existing container.

C# Copy

```
// Set throughput to the minimum value of 400 RU/s
ContainerResponse simpleContainer = await database.CreateContainerIfNotExistsAsync(
    id: containerId,
```

```
    partitionKeyPath: partitionKey,
    throughput: 400);
```

Get a container by ID

C# Copy

```
Container container = database.GetContainer(containerId);
ContainerProperties containerProperties = await container.ReadContainerAsync();
```

Delete a container

Delete a Container as an asynchronous operation.

C# Copy

```
await database.GetContainer(containerId).DeleteContainerAsync();
```

Item examples

Create an item

Use the `Container.CreateItemAsync` method to create an item. The method requires a JSON serializable object that must contain an `id` property, and a `partitionKey`.

C# Copy

```
ItemResponse<SalesOrder> response = await container.CreateItemAsync(salesOrder, new
PartitionKey(salesOrder.AccountNumber));
```

Read an item

Use the `Container.ReadItemAsync` method to read an item. The method requires type to serialize the item to along with an `id` property, and a `partitionKey`.

C# Copy

```
string id = "[id]";
string accountNumber = "[partition-key]";
ItemResponse<SalesOrder> response = await container.ReadItemAsync(id, new
PartitionKey(accountNumber));
```

Query an item

The `Container.GetItemQueryIterator` method creates a query for items under a container in an Azure Cosmos database using a SQL statement with parameterized values. It returns a `FeedIterator`.

C# Copy

```
QueryDefinition query = new QueryDefinition(
    "select * from sales s where s.AccountNumber = @AccountInput ")
    .WithParameter("@AccountInput", "Account1");

FeedIterator<SalesOrder> resultSet = container.GetItemQueryIterator<SalesOrder>(
    query,
    requestOptions: new QueryRequestOptions()
    {
        PartitionKey = new PartitionKey("Account1"),
        MaxItemCount = 1
    });
```

Additional resources

- 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.
- Visit this article Azure Cosmos DB.NET V3 SDK (Microsoft.Azure.Cosmos) examples for the SQL API for direct links to specific examples in the GitHub repository.

Next unit: Exercise: Create resources by using the Microsoft .NET SDK v3

Continue

How are we doing? ☆ ☆ ☆ ☆ ☆

100 XP

Exercise: Create resources by using the Microsoft .NET SDK v3

15 minutes

In this exercise you'll create a console app to perform the following operations in Azure Cosmos DB:

- Connect to an Azure Cosmos DB account
- Create a database
- Create a container

Prerequisites

- An Azure account with an active subscription. If you don't already have one, you can sign up for a free trial at https://azure.com/free .
- **Visual Studio Code**: You can install Visual Studio Code from https://code.visualstudio.com .
- **Azure CLI**: You can install the Azure CLI from https://docs.microsoft.com/cli/azure/install-azure-cli.
- The **.NET Core 3.1 SDK**, or **.NET 5.0 SDK**. You can install from https://dotnet.microsoft.com/download .

Setting up

Perform the following actions to prepare Azure, and your local environment, for the exercise.

Connecting to Azure

1. Start Visual Studio Code and open a terminal window by selecting **Terminal** from the top application bar, then choosing **New Terminal**.

2. Log in to Azure by using the command below. A browser window should open letting you choose which account to log in with.

```
az login
```

Create resources in Azure

1. Create a resource group for the resources needed for this exercise. Replace `<myLocation>` with a region near you.

```
az group create --location <myLocation> --name az204-cosmos-rg
```

2. Create the Azure Cosmos DB account. Replace `<myCosmosDBacct>` with 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. *This command will take a few minutes to complete.*

```
az cosmosdb create --name <myCosmosDBacct> --resource-group az204-cosmos-rg
```

   Record the `documentEndpoint` shown in the JSON response, it will be used below.

3. Retrieve the primary key for the account by using the command below. Record the `primaryMasterKey` from the command results it will be used in the code below.

```
# Retrieve the primary key
az cosmosdb keys list --name <myCosmosDBacct> --resource-group az204-cosmos-rg
```

Set up the console application

Now that the needed resources are deployed to Azure the next step is to set up the console application using the same terminal window in Visual Studio Code.

1. Create a folder for the project and change in to the folder.

```
md az204-cosmos
cd az204-cosmos
```

2. Create the .NET console app.

```
dotnet new console
```

3. Open the current folder in VS Code using the command below. The `-r` option will open the folder without launching a new VS Code window.

```
code . -r
```

4. Select the *Program.cs* file in the **Explorer** pane to open the file in the editor.

Build the console app

It's time to start adding the packages and code to the project.

Add packages and using statements

1. Open the terminal in VS Code and use the command below to add the `Microsoft.Azure.Cosmos` package to the project.

```
dotnet add package Microsoft.Azure.Cosmos
```

2. Add using statements to include `Microsoft.Azure.Cosmos` and to enable async operations.

C#

```
using System.Threading.Tasks;
using Microsoft.Azure.Cosmos;
```

Add code to connect to an Azure Cosmos DB account

1. Replace the entire `class Program` with the code snippet below. The code snippet adds constants and variables into the class and adds some error checking. Be sure to replace the placeholder values for `EndpointUri` and `PrimaryKey` following the directions in the code comments.

C# Copy

```
public class Program
{
    // Replace <documentEndpoint> with the information created earlier
    private static readonly string EndpointUri = "<documentEndpoint>";

    // Set variable to the Primary Key from earlier.
    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 names of the database and container we will create
    private string databaseId = "az204Database";
    private string containerId = "az204Container";

    public static async Task Main(string[] args)
    {
        try
        {
            Console.WriteLine("Beginning operations...\n");
            Program p = new Program();
            await p.CosmosAsync();

        }
        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 program, press any key to exit.");
            Console.ReadKey();
        }
    }
}
```

2. Below the `Main` method, add a new asynchronous task called `CosmosAsync`, which instantiates our new `CosmosClient`.

C# Copy

```
public async Task CosmosAsync()
{
    // Create a new instance of the Cosmos Client
    this.cosmosClient = new CosmosClient(EndpointUri, PrimaryKey);
}
```

Create a database

1. Copy and paste the `CreateDatabaseAsync` method after the `CosmosAsync` method. `CreateDatabaseAsync` creates a new database with ID `az204Database` if it doesn't already exist.

C# Copy

```
private async Task CreateDatabaseAsync()
{
    // Create a new database using the cosmosClient
    this.database = await
this.cosmosClient.CreateDatabaseIfNotExistsAsync(databaseId);
    Console.WriteLine("Created Database: {0}\n", this.database.Id);
}
```

2. Add the code below at the end of the `CosmosAsync` method, it calls the `CreateDatabaseAsync` method you just added.

C# Copy

```
    // Runs the CreateDatabaseAsync method
    await this.CreateDatabaseAsync();
```

Create a container

1. Copy and paste the `CreateContainerAsync` method below the `CreateDatabaseAsync` method.

```
C#
```

```
private async Task CreateContainerAsync()
{
    // Create a new container
    this.container = await
this.database.CreateContainerIfNotExistsAsync(containerId, "/LastName");
    Console.WriteLine("Created Container: {0}\n", this.container.Id);
}
```

2. Copy and paste the code below where you instantiated the `CosmosClient` to call the **CreateContainer** method you just added.

```
C#
```

```
// Run the CreateContainerAsync method
await this.CreateContainerAsync();
```

Run the application

1. Save your work and, in a terminal in VS Code, run the `dotnet run` command. The console will display the following messages.

```
Beginning operations...

Created Database: az204Database

Created Container: az204Container

End of program, press any key to exit.
```

2. You can verify the results by opening the Azure portal, navigating to your Azure Cosmos DB resource, and use the **Data Explorer** to view the database and container.

Clean up Azure resources

You can now safely delete the *az204-cosmos-rg* resource group from your account by running the command below.

Copy

```
az group delete --name az204-cosmos-rg --no-wait
```

Next unit: Create stored procedures

Continue >

How are we doing? ☆ ☆ ☆ ☆ ☆

100 XP

Create stored procedures

3 minutes

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.

> **Note**
>
> This unit focuses on stored procedures, the following unit covers triggers and user-defined functions.

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.

JavaScript Copy

```
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 stored procedure also includes a parameter to set the description, it's a boolean value. When the parameter is set to true and the description is missing, the stored procedure will throw an exception. Otherwise, the rest of the stored procedure continues to run.

The following example stored procedure takes an 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.

JavaScript Copy

```
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:

JavaScript Copy

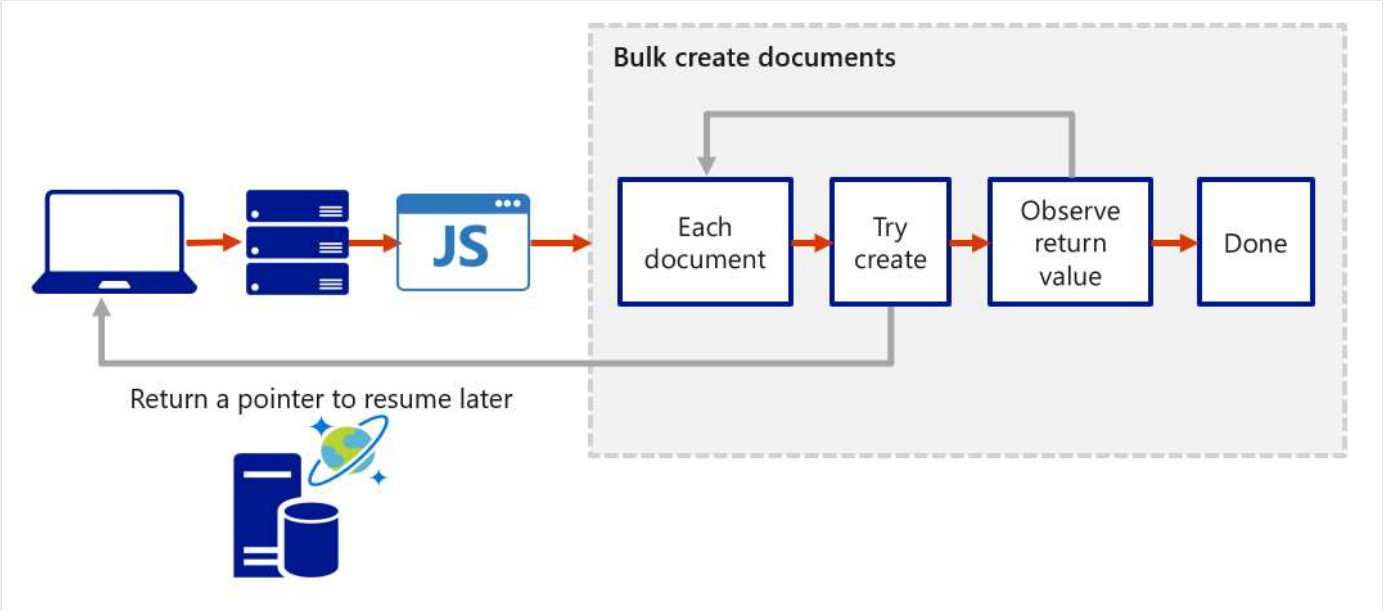
```
function sample(arr) {
    if (typeof arr === "string") arr = JSON.parse(arr);

    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.



Next unit: Create triggers and user-defined functions

Continue >

How are we doing? ☆ ☆ ☆ ☆ ☆

100 XP

Create triggers and user-defined functions

3 minutes

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 and 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.

JavaScript Copy

```
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.

For examples of how to register and call a pre-trigger, visit the pre-triggers article.

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.

JavaScript Copy

```
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.

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:

JSON Copy

```
{
   "name": "User One",
   "country": "USA",
   "income": 70000
}
```

The following is a function definition to calculate income tax for various income brackets:

JavaScript Copy

```
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;
    }
```

Next unit: Knowledge check

Continue >

How are we doing? ☆ ☆ ☆ ☆ ☆

200 XP

Knowledge check

5 minutes

Check your knowledge

1. When defining a stored procedure in the Azure portal input parameters are always sent as what type to the stored procedure?

- (●) String
  - ✔ That's correct. When defining a stored procedure in Azure portal, input parameters are always sent as a string to the stored procedure.
- ( ) Integer
- ( ) Boolean

2. Which of the following would one use to validate properties of an item being created?

- ( ) Pre-trigger
  - ✔ That's correct. Pre-triggers can be used to conform data before it is added to the container.
- (●) Post-trigger
  - ✘ That's incorrect. Post-triggers run after an item is created.
- ( ) User-defined function

Next unit: Summary

Continue

How are we doing? ☆ ☆ ☆ ☆ ☆

100 XP

Summary

3 minutes

In this module you learned how to:

- Identify classes and methods used to create resources
- Create resources by using the Azure Cosmos DB .NET v3 SDK
- Write stored procedures, triggers, and user-defined functions by using JavaScript

Module complete:

Unlock achievement

How are we doing? ☆ ☆ ☆ ☆ ☆