**DP 200 - Implementing a Data Platform Solution**

**Lab 4 - Building Globally Distributed Databases with Cosmos DB**

**Estimated Time**: 90 minutes

**Pre-requisites**: It is assumed that the case study for this lab has already been read. It is assumed that the content and lab for module 1: Azure for the Data Engineer has also been completed

**Lab files**: The files for this lab are located in the *Allfiles\Labfiles\Starter\DP-200.4* folder.

**Lab overview**

The students will be able to describe and demonstrate the capabilities that Azure Cosmos DB can bring to an organization. They will be able to create a Cosmos DB instance and show how to upload and query data through a portal and through a .Net application. They will then be able to demonstrate how to enable global scale of the Cosmos DB database.

**Lab objectives**

After completing this lab, you will be able to:

1. Create an Azure Cosmos DB database built to scale
2. Insert and query data in your Azure Cosmos DB database
3. Build a .NET Core app for Azure Cosmos DB in Visual Studio Code
4. Distribute your data globally with Azure Cosmos DB

**Scenario**

The developers and Information Services department at AdventureWorks are aware that a new service known as Cosmos DB recently released on Azure can provided planetary scale access to data in near real-time. They want to understand the capability that the service can offer and how it can bring value to AdventureWorks, and in what circumstances.

The Information Services department want to understand how the service can be setup and how data can be uploaded. The developers would like to see an example of an application that can be used to upload data to the Cosmos. Both would like to understand how the claim of planetary scale can be met.

At the end of this lad, you will have:

1. Create an Azure Cosmos DB database built to scale
2. Insert and query data in your Azure Cosmos DB database
3. Build a .NET Core app for Azure Cosmos DB in Visual Studio Code
4. Distribute your data globally with Azure Cosmos DB

**IMPORTANT**: As you go through this lab, make a note of any issue(s) that you have encountered in any provisioning or configuration tasks and log it in the table in the document located at *\Labfiles\DP-200-Issues-Doc.docx*. Document the Lab number, note the technology, Describe the issue, and what was the resolution. Save this document as you will refer back to it in a later module.

**Exercise 1: Create an Azure Cosmos DB database built to scale**

Estimated Time: 10 minutes

Individual exercise

The main task for this exercise are as follows:

1. Create an Azure Cosmos DB instance

**Task 1: Create an Azure Cosmos DB instance**

1. In the Azure portal, navigate to the **+ Create a resource** blade.
2. In the New blade, navigate to the **Search the Marketplace** text box, and type the word **Cosmos**. Click **Azure Cosmos DB** in the list that appears.
3. In the **Azure Cosmos DB** blade, click **Create**.
4. From the **Create Azure Cosmos DB Account** screen, create an Azure Cosmos DB Account with the following settings:
   * Subscription: the name of the subscription you are using in this lab
   * Resource group name: **awrgstudxx**, where **xx** are your initials
   * Account name: **awcdbstudxx**, where **xx** are your initials.
   * API: **Core(SQL)**
   * Location: the name of the Azure region which is closest to the lab location and where you can provision Azure VMs.
5. In the **Create Azure Cosmos DB Account** blade, click **Review + create**.
6. After the validation of the **Create Azure Cosmos DB Account**\* blade, click **Create**.

**Note**: The provision will take approximately 5 minutes. What is often avoided in these labs is a description of the additional tabs when you provision any service in Azure. You may notice that in the provisioning screen there will be additional tabs such as Network, Tags or Advanced. This enables you to define any customized settings for a service. For example, the network tab of many services enables you to define the configuration of virtual networks, so that you are able to control and secure the network traffic against a given data service. The Tags option are name/value pairs that enable you to categorize resources and view consolidated billing by applying the same tag to multiple resources and resource groups. Advanced tabs will vary dependent on the service that has it. But it is important to note that you have control over these areas and you will want to collaborate with your Network admins or indeed your finance department to see how these options should be configured.

1. When the provisioning is complete, the "Your deployment is complete" screen appears, click on **Go to resource** and move onto the next exercise.

**Result** In this exercise, you have provisioned an Azure Cosmos DB Account

**Exercise 2: Insert and query data in your Azure Cosmos DB database**

Estimated Time: 20 minutes

Individual exercise

The main tasks for this exercise are as follows:

1. Setup your Azure Cosmos DB database and collection
2. Add data using the portal
3. Run queries in the Azure portal
4. Run complex operations on your data

**Task 1: Setup your Azure Cosmos DB database and collection**

1. In the Azure portal, in the **awcdbstudxx - Quick start** screen, click on the **Overview** option in the blade
2. In the **awcdbstudxx** screen, click **+ Add Container**. This opens up **Data Explorer** with the **Add Container** screen.
3. In the **Add Container** screen, create a Products database with a container named Clothing with the following settings:
   * Database id: **Products**
   * Container Id: **Clothing**
   * Partition key: **/productId**
   * Leave the remaining options with their default values

1.In the **Add Container** screen, click **OK**

**Task 2: Add data using the portal**

1. In the **awcdbstudcto - Data Explorer** screen, on the Data Explorer toolbar, opposite the button for New Collection, click on the th **Open Full Screen** button. In the Open Full Screen dialog box, click **Open**. A new tab opens up in Microsoft Edge.
2. In the **SQL API** pane, expand **Clothing** and click on **Items**. A new document appears with a sample JSON that you will now replace.
3. In the Documents pane, click on the icon for **New Item**.
4. Copy the following code and paste it into the **Items** tab:

{

"id": "1",

"productId": "33218896",

"category": "Women's Clothing",

"manufacturer": "Contoso Sport",

"description": "Quick dry crew neck t-shirt",

"price": "14.99",

"shipping": {

"weight": 1,

"dimensions": {

"width": 6,

"height": 8,

"depth": 1

}

}

}

1. Once you've added the JSON to the Items tab, click **Save**.
2. In the Documents pane, click on the icon for **New Item**.
3. Copy the following code and paste it into the **Items** tab:

{

"id": "2",

"productId": "33218897",

"category": "Women's Outerwear",

"manufacturer": "Contoso",

"description": "Black wool pea-coat",

"price": "49.99",

"shipping": {

"weight": 2,

"dimensions": {

"width": 8,

"height": 11,

"depth": 3

}

}

}

1. Once you've added the JSON to the Items tab, click **Save**.
2. You can see each document that has been saved by clicking each document on the left-hand menu.

**Task 3: Run queries in the Azure portal.**

1. In the Azure portal, in the **documents** screen, click on the button **New SQL Query**.

**Note**: A Query 1 screen appears which shows the query **SELECT \* FROM c** .

1. Write a query that returns a JSON file showing details for productId 1.
2. SELECT \*
3. FROM Products p

WHERE p.id ="1"

1. Click on the **Execute Query** icon. The following result is returned

[

{

"id": "1",

"productId": "33218896",

"category": "Women's Clothing",

"manufacturer": "Contoso Sport",

"description": "Quick dry crew neck t-shirt",

"price": "14.99",

"shipping": {

"weight": 1,

"dimensions": {

"width": 6,

"height": 8,

"depth": 1

}

},

"\_rid": "I2YsALxG+-EBAAAAAAAAAA==",

"\_self": "dbs/I2YsAA==/colls/I2YsALxG+-E=/docs/I2YsALxG+-EBAAAAAAAAAA==/",

"\_etag": "\"0000844e-0000-1a00-0000-5ca79f840000\"",

"\_attachments": "attachments/",

"\_ts": 1554489220

}

]

1. In the existing query window. Write a query that returns the id, manufacturer and description in a JSON file for productId

SELECT

p.id,

p.manufacturer,

p.description

FROM Products p

WHERE p.id ="1"

1. Click on the **Execute Query** icon. The following result is returned

[

{

"id": "1",

"manufacturer": "Contoso Sport",

"description": "Quick dry crew neck t-shirt"

}

]

1. In the existing query window, write a query that returns returns the price, description, and product ID for all products, ordered by price, in ascending order.

SELECT p.price, p.description, p.productId

FROM Products p

ORDER BY p.price ASC

Click on the **Execute Query** icon. The following result is returned

[

{

"price": "14.99",

"description": "Quick dry crew neck t-shirt",

"productId": "33218896"

},

{

"price": "49.99",

"description": "Black wool pea-coat",

"productId": "33218897"

}

]

**Task 4: Run complex operations on your data**

1. In the Azure portal, in the **documents** screen, click on the button **New Stored Procedure**.

**Note**: A New Stored Procedure screen appears which shows a sample stored procedure .

1. In the New Stored Procedure screen, in the **Stored Procedure Id** text box, type **createMyDocument**.
2. Use the following code to create a stored procedure in the Stored Procedure Body.

function createMyDocument() {

var context = getContext();

var collection = context.getCollection();

var doc = {

"id": "3",

"productId": "33218898",

"description": "Contoso microfleece zip-up jacket",

"price": "44.99"

};

var accepted = collection.createDocument(collection.getSelfLink(),

doc,

function (err, documentCreated) {

if (err) throw new Error('Error' + err.message);

context.getResponse().setBody(documentCreated)

});

if (!accepted) return;

}

1. In the New Stored Procedure screen, click **Save**.
2. In the New Stored Procedure screen, click **Execute**.
3. In the Input Parameters screen, type **33218898** in the **Partition Key Value** text box, and then click **Execute**.

The following result is returned

```JSON

{

"id": "3",

"productId": "33218898",

"description": "Contoso microfleece zip-up jacket",

"price": "44.99",

"\_rid": "I2YsALxG+-EDAAAAAAAAAA==",

"\_self": "dbs/I2YsAA==/colls/I2YsALxG+-E=/docs/I2YsALxG+-EDAAAAAAAAAA==/",

"\_etag": "\"0000874e-0000-1a00-0000-5ca7a7050000\"",

"\_attachments": "attachments/"

}

```

1. In the Azure portal, in the **documents** screen, click on the drop down button **New Stored Procedure** and click **New UDF** .

**Note**: A New UDF 1 screen appears which shows **function userDefinedFunction(){}**

1. In the New Stored Procedure screen, in the **User Defined Function Id** text box, type **producttax**.
2. Use the following code to create a UDF in the User Defined Function Body.

function producttax(price) {

if (price == undefined)

throw 'no input';

var amount = parseFloat(price);

if (amount < 1000)

return amount \* 0.1;

else if (amount < 10000)

return amount \* 0.2;

else

return amount \* 0.4;

}

1. In the New UDF 1 screen, click **Save**.
2. Click on the Query 1 tab, and replace the existing query with the following query:

SELECT c.id, c.productId, c.price, udf.producttax(c.price) AS producttax FROM c

1. In the Query 1 screen, click **Execute Query**.

The following result is returned

```JSON

[

{

"id": "1",

"productId": "33218896",

"price": "14.99",

"producttax": 1.499

},

{

"id": "2",

"productId": "33218897",

"price": "49.99",

"producttax": 4.9990000000000005

},

{

"id": "3",

"productId": "33218898",

"price": "44.99",

"producttax": 4.4990000000000005

}

]

```

**Exercise 3: Build a .NET Core app for Azure Cosmos DB in Visual Studio Code**

Estimated Time: 45 minutes

Individual exercise

The main tasks for this exercise are as follows:

1. Enable Azure Cosmos DB resources in Visual Studio Code
2. Setup an application in Visual Studio Code
3. Create, read, update, and delete NoSQL data programmatically
4. Query using the Azure Cosmos DB .NET Core SDK
5. Create and run stored procedures from your application

**Task 1: Enable Azure Cosmos DB resources in Visual Studio Code**

1. Open up Visual Studio Code.
2. On the left hand menu, click the extensions button.
3. In the Search Extensions in Marketplace text box, type **Cosmos DB** and click on **Azure Cosmos DB**. A document appears in Visual Studio Code and then click on **install**

**Note**: A moving bar will appear whilst the extension is being installed. Then a tick will appear next to Installed once completed.

1. After installation is complete, click **Reload**.
2. In the Search Extensions in Marketplace text box, type **Azure Account** and click on **Azure Account extension**. A document appears in Visual Studio Code and then click on **install**

**Note**: A moving bar will appear whilst the extension is being installed. Then a tick will appear next to Installed once completed.

1. After installation is complete, click **Reload**.
2. In Visual Studio Code, sign in to Azure by clicking **View**, **Command Palette** and typing **Azure: Sign In**.

**Note**: Follow the prompts in the web browser to pick an account which authenticates your Visual Studio Code session. A message in the browser will confirm that you are signed in. You can close down this tab.

1. In Visual Studio Code, click the **Azure icon** on the left menu, under Cosmos DB, right-click your Subscription, and click **Create Account**.
2. In Visual Studio Code, an **Open Folder** dialog box appears.
3. Create a new folder named **learning-module** in the location of your choice, and then click **Select Folder**.

**Note**: It is possible at this point that Visual Studio Code restarts as you are setting up your working folder. As a result you have to re- open you lab documentation.

1. In Visual Studio Code, click the **Azure icon** on the left menu, under **Cosmos DB**, right-click your **Subscription**, and click **Create Account**.
2. In the text box at the top of the screen named **Create new Cosmos DB Account**, enter a unique name for your Azure Cosmos DB account, named **xxcosmos**, where xx are your initials and then press **Enter** on the keyboard.
3. Next, select **SQL (DocumentDB)**, then select the **awrgstudxx** resource group, and then select a **location** closest to you.

**Note**: A bar will appear at the bottom of Visual Studio Code stating "Creating Cosmos DB Account..." and will take approximately 5 minutes. The bar will change to state that the creation has occurred successfully.

1. After the account is created, expand your Azure subscription in the Azure: Cosmos DB pane and the extension displays the new Azure Cosmos DB account.
2. Right-click your new account, and then click **Create Database**.
3. In the input palette at the top of the screen, type **Users** for the database name and press **Enter**.
4. Enter **WebCustomers** for the collection name and press **Enter**.
5. Enter **userId** for the partition key and press **Enter**.
6. Finally, confirm **1000** for the initial throughput capacity and press **Enter**.
7. Expand the account in the Azure: Cosmos DB pane, and the new Users database and WebCustomers collection are displayed.

**Task 2: Setup an application in Visual Studio Code**

1. Ensure that file auto-save is enabled by clicking on the **File** menu and checking **Auto Save** if it is blank. You will be copying in several blocks of code, and this will ensure you are always operating against the latest edits of your files.
2. Open the integrated terminal from Visual Studio Code by selecting **View**, **Terminal** from the main menu.
3. Click in the terminal window, press **Enter** on the keyboard and then copy and paste the following command.

dotnet new console

**Note**: This command creates a Program.cs file in your **learning-module** folder with a basic "Hello World" program already written, along with a C# project file named learning-module.csproj.

1. In the terminal window, copy and paste the following command to run the **"Hello World!"** program.

dotnet run

1. At the terminal prompt, copy and paste the following command block to install the required NuGet packages.

dotnet add package System.Net.Http

dotnet add package System.Configuration

dotnet add package System.Configuration.ConfigurationManager

dotnet add package Microsoft.Azure.DocumentDB.Core

dotnet add package Newtonsoft.Json

dotnet add package System.Threading.Tasks

dotnet add package System.Linq

dotnet restore

1. The terminal screen will load package and display the command at the end **dotnet restore**, in the terminal window press **Enter**.
2. At the top of the Explorer pane, click **Program.cs** to open the file. Add the following using statements after **using System**.

using System.Configuration;

using System.Linq;

using System.Threading.Tasks;

using System.Net;

using Microsoft.Azure.Documents;

using Microsoft.Azure.Documents.Client;

using Newtonsoft.Json;

**Important**: Wait for about 15 seconds until you will get a message about adding required missing assets, click **Yes**.

1. In the learning-module folder, create a new file named **App.config**, and add the following code:

<?xml version="1.0" encoding="utf-8"?>

<configuration>

<appSettings>

<add key="accountEndpoint" value="<replace with your Endpoint URL>" />

<add key="accountKey" value="<replace with your Primary Key>" />

</appSettings>

</configuration>

1. Copy your connection string by clicking the **Azure icon** on the left, expanding your Subscription, **right-clicking your new Azure Cosmos DB account**, and then clicking **Copy Connection String**.
2. Paste the connection string into the end of the App.config file, and then copy the **AccountEndpoint** portion from the connection string into the **accountEndpoint** value in App.config.
3. Now copy the **AccountKey** value from the connection string into the **accountKey** value, and then **delete the original connection string** you copied in.
4. Your final App.config file looks similar to this.

<?xml version="1.0" encoding="utf-8"?>

<configuration>

<appSettings>

<add key="accountEndpoint" value="https://my-account.documents.azure.com:443/" />

<add key="accountKey" value="6e7sRxunccGEeO7IVlMdeFt5BdsllfSGLYc28KyjzkESiCu7tfWbTaZXAErt2v88gOcMbOYgwp1q4NYDifD7ew==" />

</appSettings>

</configuration>

1. At the terminal prompt, copy and paste the following command to run the program.

dotnet run

**Note**: The program displays Hello World! in the terminal.

1. In the Explorer pane, click Program.cs to open the file. Add the following to the beginning of the Program class.

private DocumentClient client;

1. Add a new asynchronous task to create a new client, and check whether the Users database exists by adding the following method after the Main method.

private async Task BasicOperations()

{

this.client = new DocumentClient(new Uri(ConfigurationManager.AppSettings["accountEndpoint"]), ConfigurationManager.AppSettings["accountKey"]);

await this.client.CreateDatabaseIfNotExistsAsync(new Database { Id = "Users" });

await this.client.CreateDocumentCollectionIfNotExistsAsync(UriFactory.CreateDatabaseUri("Users"), new DocumentCollection { Id = "WebCustomers" });

Console.WriteLine("Database and collection validation complete");

}

1. In the integrated terminal, again, copy and paste the following command to run the program to ensure that the code runs. "Hello World!" is returned in the console.

dotnet run

1. Copy and paste the following code into the Main method, overwriting the current **Console.WriteLine("Hello World!");** code:

try

{

Program p = new Program();

p.BasicOperations().Wait();

}

catch (DocumentClientException de)

{

Exception baseException = de.GetBaseException();

Console.WriteLine("{0} error occurred: {1}, Message: {2}", de.StatusCode, de.Message, baseException.Message);

}

catch (Exception e)

{

Exception baseException = e.GetBaseException();

Console.WriteLine("Error: {0}, Message: {1}", e.Message, baseException.Message);

}

finally

{

Console.WriteLine("End of demo, press any key to exit.");

Console.ReadKey();

}

1. In the integrated terminal, again, copy and paste the following command to run the program to ensure it runs.

dotnet run

1. The following result should be displayed

Database and collection validation complete

End of demo, press any key to exit.

**Result** In this exercise, you have created a client side application that can connect to Azure Cosmos DB

**Task 3: Create, read, update, and delete NoSQL data programmatically**

**Create Documents**

1. Create a **User** class that represents the objects to store in Azure Cosmos DB. We will also create **OrderHistory** and **ShippingPreference** classes that are used within **User**. Note that documents must have an Id property serialized as id in JSON.
2. To create these classes, copy and paste the following **User**, **OrderHistory**, **ShippingPreference** and **CouponsUsed** classes underneath the **BasicOperations** method.

public class User

{

[JsonProperty("id")]

public string Id { get; set; }

[JsonProperty("userId")]

public string UserId { get; set; }

[JsonProperty("lastName")]

public string LastName { get; set; }

[JsonProperty("firstName")]

public string FirstName { get; set; }

[JsonProperty("email")]

public string Email { get; set; }

[JsonProperty("dividend")]

public string Dividend { get; set; }

[JsonProperty("OrderHistory")]

public OrderHistory[] OrderHistory { get; set; }

[JsonProperty("ShippingPreference")]

public ShippingPreference[] ShippingPreference { get; set; }

[JsonProperty("CouponsUsed")]

public CouponsUsed[] Coupons { get; set; }

public override string ToString()

{

return JsonConvert.SerializeObject(this);

}

}

public class OrderHistory

{

public string OrderId { get; set; }

public string DateShipped { get; set; }

public string Total { get; set; }

}

public class ShippingPreference

{

public int Priority { get; set; }

public string AddressLine1 { get; set; }

public string AddressLine2 { get; set; }

public string City { get; set; }

public string State { get; set; }

public string ZipCode { get; set; }

public string Country { get; set; }

}

public class CouponsUsed

{

public string CouponCode { get; set; }

}

private void WriteToConsoleAndPromptToContinue(string format, params object[] args)

{

Console.WriteLine(format, args);

Console.WriteLine("Press any key to continue ...");

Console.ReadKey();

}

1. In the integrated terminal, again, copy and paste the following command to run the program to ensure it runs.

dotnet run

1. The following result should be displayed

Database and collection validation complete

End of demo, press any key to exit.

1. Now copy and paste the **CreateUserDocumentIfNotExists** task under the **WriteToConsoleAndPromptToContinue** method at the end of the Program.cs file.

private async Task CreateUserDocumentIfNotExists(string databaseName, string collectionName, User user)

{

try

{

await this.client.ReadDocumentAsync(UriFactory.CreateDocumentUri(databaseName, collectionName, user.Id), new RequestOptions { PartitionKey = new PartitionKey(user.UserId) });

this.WriteToConsoleAndPromptToContinue("User {0} already exists in the database", user.Id);

}

catch (DocumentClientException de)

{

if (de.StatusCode == HttpStatusCode.NotFound)

{

await this.client.CreateDocumentAsync(UriFactory.CreateDocumentCollectionUri(databaseName, collectionName), user);

this.WriteToConsoleAndPromptToContinue("Created User {0}", user.Id);

}

else

{

throw;

}

}

}

1. Then, return to the **BasicOperations** method and add the following to the end of that method.

User yanhe = new User

{

Id = "1",

UserId = "yanhe",

LastName = "He",

FirstName = "Yan",

Email = "yanhe@contoso.com",

OrderHistory = new OrderHistory[]

{

new OrderHistory {

OrderId = "1000",

DateShipped = "08/17/2018",

Total = "52.49"

}

},

ShippingPreference = new ShippingPreference[]

{

new ShippingPreference {

Priority = 1,

AddressLine1 = "90 W 8th St",

City = "New York",

State = "NY",

ZipCode = "10001",

Country = "USA"

}

},

};

await this.CreateUserDocumentIfNotExists("Users", "WebCustomers", yanhe);

User nelapin = new User

{

Id = "2",

UserId = "nelapin",

LastName = "Pindakova",

FirstName = "Nela",

Email = "nelapin@contoso.com",

Dividend = "8.50",

OrderHistory = new OrderHistory[]

{

new OrderHistory {

OrderId = "1001",

DateShipped = "08/17/2018",

Total = "105.89"

}

},

ShippingPreference = new ShippingPreference[]

{

new ShippingPreference {

Priority = 1,

AddressLine1 = "505 NW 5th St",

City = "New York",

State = "NY",

ZipCode = "10001",

Country = "USA"

},

new ShippingPreference {

Priority = 2,

AddressLine1 = "505 NW 5th St",

City = "New York",

State = "NY",

ZipCode = "10001",

Country = "USA"

}

},

Coupons = new CouponsUsed[]

{

new CouponsUsed{

CouponCode = "Fall2018"

}

}

};

await this.CreateUserDocumentIfNotExists("Users", "WebCustomers", nelapin);

1. In the integrated terminal, again, copy and paste the following command to run the program to ensure it runs.

dotnet run

1. The following result should be displayed

Database and collection validation complete

Created User 1

Press any key to continue ...

Created User 2

Press any key to continue ...

End of demo, press any key to exit.

**Read Documents**

1. To read documents from the database, copy in the following code and place after the **WriteToConsoleAndPromptToContinue** method in the Program.cs file.

private async Task ReadUserDocument(string databaseName, string collectionName, User user)

{

try

{

await this.client.ReadDocumentAsync(UriFactory.CreateDocumentUri(databaseName, collectionName, user.Id), new RequestOptions { PartitionKey = new PartitionKey(user.UserId) });

this.WriteToConsoleAndPromptToContinue("Read user {0}", user.Id);

}

catch (DocumentClientException de)

{

if (de.StatusCode == HttpStatusCode.NotFound)

{

this.WriteToConsoleAndPromptToContinue("User {0} not read", user.Id);

}

else

{

throw;

}

}

}

1. Copy and paste the following code to the end of the BasicOperations method, after the **await this.CreateUserDocumentIfNotExists("Users", "WebCustomers", nelapin);** line:

await this.ReadUserDocument("Users", "WebCustomers", yanhe);

1. In the integrated terminal, again, copy and paste the following command to run the program to ensure it runs.

dotnet run

1. The following result should be displayed

Database and collection validation complete

User 1 already exists in the database

Press any key to continue ...

User 2 already exists in the database

Press any key to continue ...

Read user 1

Press any key to continue ...

End of demo, press any key to exit.

**Replace Documents**

1. Copy and paste the **ReplaceUserDocument** method after the **ReadUserDocument** method in the Program.cs file.

private async Task ReplaceUserDocument(string databaseName, string collectionName, User updatedUser)

{

try

{

await this.client.ReplaceDocumentAsync(UriFactory.CreateDocumentUri(databaseName, collectionName, updatedUser.Id), updatedUser, new RequestOptions { PartitionKey = new PartitionKey(updatedUser.UserId) });

this.WriteToConsoleAndPromptToContinue("Replaced last name for {0}", updatedUser.LastName);

}

catch (DocumentClientException de)

{

if (de.StatusCode == HttpStatusCode.NotFound)

{

this.WriteToConsoleAndPromptToContinue("User {0} not found for replacement", updatedUser.Id);

}

else

{

throw;

}

}

}

1. Copy and paste the following code to the end of the **BasicOperations** method, after the **await this.CreateUserDocumentIfNotExists("Users", "WebCustomers", nelapin);** line.

yanhe.LastName = "Suh";

await this.ReplaceUserDocument("Users", "WebCustomers", yanhe);

1. In the integrated terminal, again, copy and paste the following command to run the program to ensure it runs.

dotnet run

1. The following result should be displayed

Database and collection validation complete

User 1 already exists in the database

Press any key to continue ...

User 2 already exists in the database

Press any key to continue ...

Replaced last name for Suh

Press any key to continue ...

Read user 1

Press any key to continue ...

End of demo, press any key to exit

**Delete Documents**

1. Copy and paste the **DeleteUserDocument** method underneath your **ReplaceUserDocument** method.

private async Task DeleteUserDocument(string databaseName, string collectionName, User deletedUser)

{

try

{

await this.client.DeleteDocumentAsync(UriFactory.CreateDocumentUri(databaseName, collectionName, deletedUser.Id), new RequestOptions { PartitionKey = new PartitionKey(deletedUser.UserId) });

Console.WriteLine("Deleted user {0}", deletedUser.Id);

}

catch (DocumentClientException de)

{

if (de.StatusCode == HttpStatusCode.NotFound)

{

this.WriteToConsoleAndPromptToContinue("User {0} not found for deletion", deletedUser.Id);

}

else

{

throw;

}

}

}

1. Copy and paste the following code to the end of the **BasicOperations** method.

await this.DeleteUserDocument("Users", "WebCustomers", yanhe);

1. In the integrated terminal, again, copy and paste the following command to run the program to ensure it runs.

dotnet run

1. The following result should be displayed

Database and collection validation complete

User 1 already exists in the database

Press any key to continue ...

User 2 already exists in the database

Press any key to continue ...

Replaced last name for Suh

Press any key to continue ...

Read user 1

Press any key to continue ...

Deleted user 1

End of demo, press any key to exit.

**Task 4: Query using the Azure Cosmos DB .NET Core SDK**

1. The following sample shows how a query could be performed in SQL, LINQ, or LINQ lambda from your .NET code. Copy the code and add it after **private async Task CreateUserDocumentIfNotExists** towards the end of the Program.cs file.

private void ExecuteSimpleQuery(string databaseName, string collectionName)

{

// Set some common query options

FeedOptions queryOptions = new FeedOptions { MaxItemCount = -1, EnableCrossPartitionQuery = true };

// Here we find nelapin via their LastName

IQueryable<User> userQuery = this.client.CreateDocumentQuery<User>(

UriFactory.CreateDocumentCollectionUri(databaseName, collectionName), queryOptions)

.Where(u => u.LastName == "Pindakova");

// The query is executed synchronously here, but can also be executed asynchronously via the IDocumentQuery<T> interface

Console.WriteLine("Running LINQ query...");

foreach (User user in userQuery)

{

Console.WriteLine("\tRead {0}", user);

}

// Now execute the same query via direct SQL

IQueryable<User> userQueryInSql = this.client.CreateDocumentQuery<User>(

UriFactory.CreateDocumentCollectionUri(databaseName, collectionName),

"SELECT \* FROM User WHERE User.lastName = 'Pindakova'", queryOptions );

Console.WriteLine("Running direct SQL query...");

foreach (User user in userQueryInSql)

{

Console.WriteLine("\tRead {0}", user);

}

Console.WriteLine("Press any key to continue ...");

Console.ReadKey();

}

1. Copy and paste the following code to your BasicOperations method, before the **await this.DeleteUserDocument("Users", "WebCustomers", yanhe);** line.

this.ExecuteSimpleQuery("Users", "WebCustomers");

1. In the integrated terminal, again, copy and paste the following command to run the program to ensure it runs.

dotnet run

1. The following result should be displayed

Database and collection validation complete

Created User 1

Press any key to continue ...

User 2 already exists in the database

Press any key to continue ...

Replaced last name for Suh

Press any key to continue ...

Read user 1

Press any key to continue ...

Running LINQ query...

Read {"id":"2","userId":"nelapin","lastName":"Pindakova","firstName":"Nela","email":"nelapin@contoso.com","dividend":"8.50","OrderHistory":[{"OrderId":"1001","DateShipped":"08/17/2018","Total":"105.89"}],"ShippingPreference":[{"Priority":1,"AddressLine1":"505 NW 5th St","AddressLine2":null,"City":"New York","State":"NY","ZipCode":"10001","Country":"USA"},{"Priority":2,"AddressLine1":"505 NW 5th St","AddressLine2":null,"City":"New York","State":"NY","ZipCode":"10001","Country":"USA"}],"CouponsUsed":[{"CouponCode":"Fall2018"}]}

Running direct SQL query...

Read {"id":"2","userId":"nelapin","lastName":"Pindakova","firstName":"Nela","email":"nelapin@contoso.com","dividend":"8.50","OrderHistory":[{"OrderId":"1001","DateShipped":"08/17/2018","Total":"105.89"}],"ShippingPreference":[{"Priority":1,"AddressLine1":"505 NW 5th St","AddressLine2":null,"City":"New York","State":"NY","ZipCode":"10001","Country":"USA"},{"Priority":2,"AddressLine1":"505 NW 5th St","AddressLine2":null,"City":"New York","State":"NY","ZipCode":"10001","Country":"USA"}],"CouponsUsed":[{"CouponCode":"Fall2018"}]}

Press any key to continue ...

Deleted user 1

End of demo, press any key to exit.

**Task 5: Create and run stored procedures from your application**

1. In Visual Studio Code, in the **Azure: Cosmos DB tab**, expand your **Azure Cosmos DB account**, **Users**, and then **WebCustomers** and then right-click **Stored Procedures** and then click **Create Stored Procedure**.
2. In the text box at the top of the screen, type **UpdateOrderTotal** and press Enter to give the stored procedure a name.

**Note**: By default, a stored procedure that retrieves the first item is provided. if not perform the following steps:

1. In the **Azure: Cosmos DB tab**, expand **Stored Procedures** and click **UpdateOrderTotal**.
2. To run this default stored procedure from your application, add the following code to the **Program.cs** file as the first entry after the **class Program**.

public async Task RunStoredProcedure(string databaseName, string collectionName, User user)

{

await client.ExecuteStoredProcedureAsync<string>(UriFactory.CreateStoredProcedureUri(databaseName, collectionName, "UpdateOrderTotal"), new RequestOptions { PartitionKey = new PartitionKey(user.UserId) });

Console.WriteLine("Stored procedure complete");

}

1. Copy the following code and paste it before the await this.DeleteUserDocument("Users", "WebCustomers", yanhe); line in the BasicOperations method.

await this.RunStoredProcedure("Users", "WebCustomers", yanhe);

1. In the integrated terminal, again, copy and paste the following command to run the program to ensure it runs.

dotnet run

1. The following result should be displayed

Database and collection validation complete

Created User 1

Press any key to continue ...

User 2 already exists in the database

Press any key to continue ...

Replaced last name for Suh

Press any key to continue ...

Read user 1

Press any key to continue ...

Running LINQ query...

Read {"id":"2","userId":"nelapin","lastName":"Pindakova","firstName":"Nela","email":"nelapin@contoso.com","dividend":"8.50","OrderHistory":[{"OrderId":"1001","DateShipped":"08/17/2018","Total":"105.89"}],"ShippingPreference":[{"Priority":1,"AddressLine1":"505 NW 5th St","AddressLine2":null,"City":"New York","State":"NY","ZipCode":"10001","Country":"USA"},{"Priority":2,"AddressLine1":"505 NW 5th St","AddressLine2":null,"City":"New York","State":"NY","ZipCode":"10001","Country":"USA"}],"CouponsUsed":[{"CouponCode":"Fall2018"}]}

Running direct SQL query...

Read {"id":"2","userId":"nelapin","lastName":"Pindakova","firstName":"Nela","email":"nelapin@contoso.com","dividend":"8.50","OrderHistory":[{"OrderId":"1001","DateShipped":"08/17/2018","Total":"105.89"}],"ShippingPreference":[{"Priority":1,"AddressLine1":"505 NW 5th St","AddressLine2":null,"City":"New York","State":"NY","ZipCode":"10001","Country":"USA"},{"Priority":2,"AddressLine1":"505 NW 5th St","AddressLine2":null,"City":"New York","State":"NY","ZipCode":"10001","Country":"USA"}],"CouponsUsed":[{"CouponCode":"Fall2018"}]}

Press any key to continue ...

Stored procedure complete

Deleted user 1

End of demo, press any key to exit.

**Result** In this exercise, you have built a console application from scratch by setting up Visual Studio code to connect with Azure Cosmos DB. You have then created .Net code to create, read, update, and delete NoSQL data programmatically. You then added code to query the Cosmos DB and learned how to create programming objects such as Stored Procedures to work against Cosmos DB

**Exercise 4: Distribute your data globally with Azure Cosmos DB**

Estimated Time: 15 minutes

Individual exercise

The main tasks for this exercise are as follows:

1. Replicate Data to Multiple Regions
2. Managing Failover

**Task 1: Replicate Data to Multiple Regions**

1. In Microsoft Edge, click on the tab that states **Data Explorer - Microsoft..**.
2. If a message appears that states "Connection error", click on the button **Refresh**.
3. In the **awcdbstudxx - Data Explorer** window, click on **Replicate data globally**
4. On the world map, single click a data center location within the continent you reside, and click on **Save**.

**Note** The provisioning of the additional data centers will take approximately 7 minutes

**Task 2: Managing Failover.**

1. In the **awcdbstudxx - Replicate data globally** window, click on **Manual Failover**.
2. Click on the **Read Region** datacenter location and click on **OK**.

**Note** The Manual Failover will take approximately 3 minutes.

1. In the **awcdbstudxx - Replicate data globally** window, click on **Automatic Failover**
2. In the "Automatic Failover" screen, click on the **ON** button, and then click on **OK**.

**Note** The provisioning of the Automatic Failover will take approximately 3 minutes.