Developing Solutions for Microsoft Azure

Exam Ref AZ-204

Santiago Fernández Muñoz

Microsoft

Exam Ref AZ-204 Developing Solutions for Microsoft Azure

Santiago Fernández Muñoz

Container or Database exists in your Cosmos DB account; if they don't exist, the method automatically creates the Container or the Database. When you create a new container in your database, notice that in this example, you have provided the PartitionKey using the appropriate constructor overload.

However, when you need to create a new document in the database, you don't have available this type of IfNotExists method. In this situation, you have two options:

1. Use the method UpsertItemAsync, which creates a new document if the document doesn't exist or updates an existing document.
2. Implement your own version of the IfNotExists method, so you need to check whether the document already exists in the container. If the document doesn't exist, then you create the actual document, as shown in the following fragment from Listing 2-2. (The code in bold shows the methods that you need to use for creating a document.)

```
try
{

    await this?.container.ReadItemAsync<Person>(person.Id, new PartitionKey
(person.LastName));

    this.SendMessageToConsoleAndWait($"Document {person.Id} already exists in
collection {collection}");
}
catch (CosmosException dce)
{
    if (dce.StatusCode == HttpStatusCode.NotFound)
    {

        await this?.container.CreateItemAsync<Person>(person,
new PartitionKey(person.LastName));

        this.SendMessageToConsoleAndWait($"Created new document {person.Id} in
collection {collection}");
    }
}
```

When you create the document using the CreateItemAsync method, notice that you can provide the value for the partition key by using the following code snippet new PartitionKey(person.LastName). If you don't provide the value for the partition key, the correct value is inferred from the document that you are trying to insert into the database.

You need to do this verification because you get a CosmosException with StatusCode 409 (Conflict) if you try to create a document with the same Id of an already existing document in the collection. Similarly, you get a CosmosException with StatusCode 404 (Not Found) if you try to delete a document that doesn't exist in the container using the DeleteItemAsync method or if you try to replace a document that doesn't exist in the container using the ReplaceItemAsync method. Notice that these two methods also accept a partition key parameter.

When you create a document, you need to provide an Id property of type string to your document. This property needs to identify your document inside the collection uniquely. If you don't provide this property, Cosmos DB automatically adds it to the document for you, using a GUID string.

As you can see in the example code in Listing 2-2, you can query your documents using LINQ or SQL sentences. In this example, I have used a pretty simple SQL query for getting documents that represent a person with the male gender. However, you can construct more complex sentences like a query that returns all people who live in a specific country, using the WHERE Address.Country = 'Spain' expression, or people that have an Android device using the WHERE ARRAY_CONTAINS(Person.Devices, { 'OperatingSystem': 'Android'}, true) expression.

NEED MORE REVIEW? **SQL QUERIES WITH COSMOS DB**

You can review all the capabilities and features of the SQL language that Cosmos DB implements by reviewing this article:

- **SQL Language Reference for Azure Cosmos DB** ***https://docs.microsoft.com/en-us/azure/cosmos-db/sql-api-query-reference***

Once you have modified the Program.cs file, you need to create some additional classes that you use in the main program for managing documents. You can find these new classes in Listings 2-3 to 2-5.

1. In the Visual Studio Code window, create a new folder named **Model** in the project folder.
2. Create a new C# class file in the Model folder and name it **Person.cs**.
3. Replace the content of the Person.cs file with the content of Listing 2-3. Change the namespace as needed for your project.
4. Create a new C# class file in the Model folder and name it **Device.cs**.
5. Replace the content of the Device.cs file with the content of Listing 2-4. Change the namespace as needed for your project.
6. Create a new C# class file in the Model folder and name it **Address.cs**.
7. Replace the content of the Address.cs file with the content of Listing 2-5. Change the namespace as needed for your project.
8. At this point, you can run the project by pressing F5 in the Visual Studio Code window. Check to see how your code is creating and modifying the different databases, document collections, and documents in your Cosmos DB account. You can review the changes in your Cosmos DB account using the Data Explorer tool in your Cosmos DB account in the Azure portal.

LISTING 2-3 Cosmos DB SQL API example: Person.cs

```
//C# .NET Core.
using Newtonsoft.Json;

namespace ch2_1_3_SQL.Model
{
```

```
    public class Person
    {
        [JsonProperty(PropertyName="id")]
        public string Id { get; set; }
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public Device[] Devices { get; set; }
        public Address Address { get; set; }
        public string Gender { get; set; }
        public bool IsRegistered { get; set; }
        public override string ToString()
        {
            return JsonConvert.SerializeObject(this);
        }
    }
}
```

LISTING 2-4 Cosmos DB SQL API example: Device.cs

```
//C# .NET Core.
namespace ch2_1_3_SQL.Model
{
    public class Device
    {
        public int Ram { get; set; }
        public string OperatingSystem { get; set; }
        public int CameraMegaPixels { get; set; }
        public string Usage { get; set; }
    }
}
```

LISTING 2-5 Cosmos DB SQL API example: Address.cs

```
//C# .NET Core.
namespace ch2_1_3_SQL.Model
{
    public class Address
    {
        public string City { get; set; }
        public string State { get; set; }
        public string PostalCode { get; set; }
        public string Country { get; set; }
        public string Street { get; set; }
    }
}
```

At this point, you can press F5 in your Visual Studio Code window to execute the code. The code stops on each step for you to be able to view the result of the operation directly on the Azure portal. Use the following steps for viewing the modifications in your Cosmos DB account:

1. Sign in to the Azure portal (*http://portal.azure.com*).
2. In the Search box at the top of the Azure portal, type the name of your Cosmos DB account and click the name of the account.
3. On your Cosmos DB Account blade, click Data Explorer.
4. On the Data Explorer blade, on the left side of the panel, under the label SQL API, you should be able to see the list of databases created in your Cosmos DB account.

Working with the MongoDB API for Cosmos DB is as easy as working with any other Mongo DB library. You only need to use the connection string that you can find in the Connection String panel under the Settings section in your Azure Cosmos DB account.

The following example shows how to use Cosmos DB in your MongoDB project. For this example, you are going to use MERN (MongoDB, Express, React, and Node), which is a full-stack framework for working with MongoDB and NodeJS. Also, you need to meet the following requirements:

- You must have the latest version of NodeJS installed on your computer.
- You must have an Azure Cosmos DB account configured for using MongoDB API. Remember that you can use the same procedure used earlier for creating a Cosmos DB with the SQL API to create an Azure Cosmos DB account with the MongoDB API. You only need to select the correct API when you are creating your Cosmos DB account.
- You need one of the connection strings that you can find in the Connection String panel in your Azure Cosmos DB account in the Azure portal. You need to copy one of these connection strings because you need to use it later in the code.

Use the following steps to connect a MERN project with Cosmos DB using the MongoDB API:

1. Create a new folder for your project.
2. Open the terminal and run the following commands:

```
git clone https://github.com/Hashnode/mern-starter.git
cd mern-starter
npm install
```

3. Open your preferred editor and open the mern-starter folder. Don't close the terminal window that you opened before.
4. In the mern-starter folder, in the server subfolder, open the config.js file and replace the content of the file with the following code:

```
const config = {
  mongoURL: process.env.MONGO_URL || '<YOUR_COSMOSDB_CONNECTION_STRING>',
  port: process.env.PORT || 8000,
};
export default config;
```

5. On the terminal window, run the command npm start. This command starts the NodeJS project and creates a Node server listening on port 8000.
6. Open a web browser and navigate to *http://localhost:8000*. This opens the MERN web project.
7. Open a new browser window, navigate to the Azure portal, and open the Data Explorer browser in your Azure Cosmos DB account.
8. In the MERN project, create, modify, or delete some posts. Review how the document is created, modified, and deleted from your Cosmos DB account.

NEED MORE REVIEW? **GREMLIN AND CASSANDRA EXAMPLES**

As you can see in the previous examples, integrating your existing code with Cosmos DB doesn't require too much effort or many changes to your code. For the sake of brevity, we decided to omit the examples of how to connect your Cassandra or Gremlin applications with Cosmos DB. You can learn how to do these integrations by reviewing the following articles:

- **Quickstart: Build a .NET Framework or Core application Using the Azure Cosmos DB Gremlin API account** ***https://docs.microsoft.com/en-us/azure/cosmos-db/create-graph-dotnet***
- **Quickstart: Build a Cassandra App with .NET SDK and Azure Cosmos DB** ***https://docs.microsoft.com/en-us/azure/cosmos-db/create-cassandra-dotnet***

Set the appropriate consistency level for operations

One of the main benefits offered by Cosmos DB is the ability to have your data distributed across the globe with low latency when accessing the data. This means that you can configure Cosmos DB for replicating your data between any of the available Azure regions while achieving minimal latency when your application accesses the data from the nearest region. If you need to replicate your data to an additional region, you only need to add to the list of regions in which your data should be available.

This replication across the different regions has a drawback—the consistency of your data. To avoid corruption, your data needs to be consistent between all copies of your database. Fortunately, the Cosmos DB protocol offers five levels of consistency replication. Going from consistency to performance, you can select how the replication protocol behaves when copying your data between all the replicas that are configured across the globe. These consistency levels are region agnostic, which means the region that started the read or write operation or the number of regions associated with your Cosmos DB account doesn't matter, even if you configured a single region for your account. You configure this consistency level at the Cosmos DB level, and it applies to all databases, collections, and documents stored inside the same

account. You can choose among the consistency levels shown in Figure 2-2. Use the following procedure to select the consistency level:

1. Sign in to the Azure portal (*http://portal.azure.com*).
2. In the Search box at the top of the Azure portal, type the name of your Cosmos DB account and click the name of the account.
3. On your Cosmos DB account blade, click Default Consistency in the Settings section.
4. On the Default Consistency blade, select the desired consistency level. Your choices are Strong, Bounded Staleness, Session, Consistent Prefix, and Eventual.
5. Click the Save icon in the top-left corner of the Default Consistency blade.

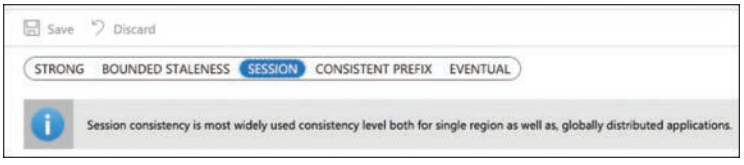

FIGURE 2-2 Selecting the consistency level

- **Strong** The read operations are guaranteed to return the most recently committed version of an element; that is, the user always reads the latest committed write. This consistency level is the only one that offers a linearizability guarantee. This guarantee comes at a price. It has higher latency because of the time needed to write operation confirmations, and the availability can be affected during failures.
- **Bounded Staleness** The reads are guaranteed to be consistent within a preconfigured lag. This lag can consist of a number of the most recent (K) versions or a time interval (T). This means that if you make write operations, the read of these operations happens in the same order but with a maximum delay of K versions of the written data or T seconds since you wrote the data in the database. For reading operations that happen within a region that accepts writes, the consistency level is identical to the Strong consistency level. This level is also known as "time-delayed linearizability guarantee."
- **Session** Scoped to a client session, this consistency level offers the best balance between a strong consistency level and the performance provided by the eventual consistency level. It best fits applications in which write operations occur in the context of a user session.
- **Consistent Prefix** This level guarantees that you always read data in the same order that you wrote the data, but there's no guarantee that you can read all the data. This means that if you write "A, B, C" you can read "A", "A, B" or "A, B, C" but never "A, C" or "B, A, C."
- **Eventual** There is no guarantee for the order in which you read the data. In the absence of a write operation, the replicas eventually converge. This consistency level offers better performance at the cost of the complexity of the programming. Use this consistency level if the order of the data is not essential for your application.