Microsoft

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

Exam Ref AZ-204 Developing Solutions for Microsoft Azure

Published with the authorization of Microsoft Corporation by:
Pearson Education, Inc.

ISBN-13: 978-0-13-679833-0
ISBN-10: 0-13-679833-0

Library of Congress Control Number: 2020942404

ScoutAutomatedPrintCode

CREDITS

EDITOR-IN-CHIEF
Brett Bartow

EXECUTIVE EDITOR
Loretta Yates

ASSOCIATE EDITOR
Charvi Arora

DEVELOPMENT EDITORS
Songlin Qiu, Charvi Arora

MANAGING EDITOR
Sandra Schroeder

SENIOR PROJECT EDITOR
Tracey Croom

COPY EDITOR
Charlotte Kughen

INDEXER
Cheryl Ann Lenser

PROOFREADER
Abigail Manheim

TECHNICAL EDITOR
Dave McCollough

EDITORIAL ASSISTANT
Cindy Teeters

COVER DESIGNER
Twist Creative, Seattle

COMPOSITOR
codeMantra

To my wonderful wife, because of her support and inspiration, especially in the hard times.

— Santiago Fernández Muñoz

This page intentionally left blank

Contents at a glance

This page intentionally left blank

Contents

Acknowledgments

I want to say thank you to the people who gave me the opportunity to write this book and who also helped me during the entire process. Without their support, this book would not be a reality.

I also want to say thank you to my friend Rafa Hueso for his support and guidance during the last years of my professional career.

This page intentionally left blank

About the author

I started my career as a Linux and Windows instructor. At the same time, I also started to learn scripting programming languages such as bash and VBS that were useful for my work. During that period of my career, I realized scripting languages were helpful, but they were not enough to meet all my needs, so I started learning other languages like Java, PHP, and finally C#.

I've been working as a Microsoft technologies consultant for the last 14 years, and over the last 6 years, I've consulted on Azure-related technologies. I've participated in different types of projects, serving in a variety of capacities from .NET developer to solution architect. Now I'm focused on developing custom industrial IoT solutions for my company and clients.

This page intentionally left blank

Introduction

Most books take a very low-level approach, teaching you how to use individual classes and accomplish fine-grained tasks. Through this book, we review the main technologies that Microsoft offers for deploying different kinds of solutions into Azure. From the most classical and conservative approaches using Azure virtual machines to the latest technologies, implementing event-based or message-based patterns with Azure Event Grid or Azure Service Bus, this book reviews the basics for developing most types of solutions using Azure services. The book also provides code examples for illustrating how to implement most of the concepts covered through the different sections.

This book is intended for those professionals who are planning to pass the exam AZ-204. This book covers every major topic area found on the exam, but it does not cover every exam question. Only the Microsoft exam team has access to the exam questions, and Microsoft regularly adds new questions to the exam, making it impossible to cover specific questions. You should consider this book a supplement to your relevant real-world experience and other study materials. If you encounter a topic in this book that you do not feel completely comfortable with, use the "Need more review?" links in the text to find more information and take the time to research and study the topic. Great information is available on MSDN and TechNet and in blogs and forums.

Organization of this book

This book is organized by the "Skills measured" list published for the exam. The "Skills measured" list is available for each exam on the Microsoft Learn website: *http://aka.ms/examlist*. Each chapter in this book corresponds to a major topic area in the list, and the technical tasks in each topic area determine a chapter's organization. If an exam covers six major topic areas, for example, the book contains six chapters.

Microsoft certifications

Microsoft certifications distinguish you by proving your command of a broad set of skills and experience with current Microsoft products and technologies. The exams and corresponding certifications are developed to validate your mastery of critical competencies as you design and develop, or implement and support, solutions with Microsoft products and technologies both on premises and in the cloud. Certification brings a variety of benefits to the individual and to employers and organizations.

MORE INFO **ALL MICROSOFT CERTIFICATIONS**

For information about Microsoft certifications, including a full list of available certifications, go to *http://www.microsoft.com/learn*.

Check back often to see what is new!

Errata, updates, and book support

We've made every effort to ensure the accuracy of this book and its companion content. You can access updates to this book—in the form of a list of submitted errata and their related corrections—at

MicrosoftPressStore.com/ExamRefAZ204/errata.

If you discover an error that is not already listed, please submit it to us at the same page.

For additional book support and information, please visit

http://www.MicrosoftPressStore.com/Support.

Please note that product support for Microsoft software and hardware is not offered through the previous addresses. For help with Microsoft software or hardware, go to *http://support.microsoft.com*.

Stay in touch

Let's keep the conversation going! We're on Twitter:

http://twitter.com/MicrosoftPress

Important: How to use this book to study for the exam

Certification exams validate your on-the-job experience and product knowledge. To gauge your readiness to take an exam, use this Exam Ref to help you check your understanding of the skills tested by the exam. Determine the topics you know well and the areas in which you need more experience. To help you refresh your skills in specific areas, we have also provided "Need more review?" pointers, which direct you to more in-depth information outside the book.

The Exam Ref is not a substitute for hands-on experience. This book is *not* designed to teach you new skills.

We recommend that you round out your exam preparation by using a combination of available study materials and courses. Learn more about available classroom training and find free online courses and live events at *http://microsoft.com/learn*. Microsoft Official Practice Tests are available for many exams at *http://aka.ms/practicetests*.

This book is organized by the "Skills measured" list published for the exam.
The "Skills measured" list for each exam is available on the Microsoft Learn website: *http://aka.ms/examlist*.

Note that this Exam Ref is based on this publicly available information and the author's experience. To safeguard the integrity of the exam, authors do not have access to the exam questions.

This page intentionally left blank

CHAPTER 2

Develop for Azure storage

All applications work with information or data. Applications create, transform, model, or operate with that information. Regardless of the type or volume of the data that your application uses, sooner or later, you need to save it persistently so that it can be used later.

Storing data is not a simple task, and designing storage systems for that purpose is even more complicated. Perhaps your application needs to deal with terabytes of information, or you may work with an application that needs to be accessed from different countries, and you need to minimize the time required to access it. Also, cost efficiency is a requirement in any project. In general, many requirements make designing and maintaining storage systems difficult.

Microsoft Azure offers different storage solutions in the cloud to satisfy your application storage requirements. Azure offers solutions for making your storage cost-effective and minimizing latency.

Skills covered in this chapter:

- Skill 2.1: Develop solutions that use Cosmos DB storage
- Skill 2.2: Develop solutions that use Blob Storage

Skill 2.1: Develop solutions that use Cosmos DB storage

Cosmos DB is a premium storage service that Azure provides for satisfying your need for a globally distributed, low-latency, highly responsive, and always-online database service. Cosmos DB has been designed with scalability and throughput in mind. One of the most significant differences between Cosmos DB and other storage services offered by Azure is how easily you can scale your Cosmos DB solution across the globe by merely clicking a button and adding a new region to your database.

Another essential feature that you should consider when evaluating this type of storage service is how you can access this service from your code and how hard it would be to migrate your existing code to a Cosmos DB–based storage solution. The good news is that Cosmos DB offers different APIs for accessing the service. The best API for you depends on the type of data that you want to store in your Cosmos DB database. You store your data using Key-Value, Column-Family, Documents, or Graph approaches. Each of the different APIs that Cosmos DB offers allows you to store your data with different schemas. Currently, you can access Cosmos DB using SQL, Cassandra, Table, Gremlin, and MongoDB APIs.

This skill covers how to

- Select the appropriate API for your solution
- Implement partitioning schemes
- Interact with data using the appropriate SDK
- Set the appropriate consistency level for operations
- Create Cosmos DB containers
- Implement server-side programming including stored procedures, triggers, and change feed notifications

Select the appropriate API for your solution

When you are planning how to store the information that your application needs to work, you need to consider the structure that you need to use for storing that information. You may find that some parts of your application need to store information using a Key-Value structure. In contrast, others may need a more flexible, schemaless structure in which you need to save the information into documents. Maybe one fundamental characteristic of your application is that you need to store the relationship between entities, and you need to use a graph structure for storing your data.

Cosmos DB offers a variety of APIs for storing and accessing your data, depending on the requirements that your application has:

- **SQL** This is the core and default API for accessing your data in your Cosmos DB account. This core API allows you to query JSON objects using SQL syntax, which means you don't need to learn another query language. Under the hood, the SQL API uses the JavaScript programming model for expression evaluation, function invocations, and typing system. You use this API when you need to use a data structure based on documents.
- **Table** You can think of the Table API as the evolution of the Azure Table Storage service. This API benefits from the high-performance, low-latency, and high-scalability features of Cosmos DB. You can migrate from your current Azure Table Storage service with no code modification in your application. Another critical difference between Table API for Cosmos DB and Azure Table Storage is that you can define your own indexes in your tables. In the same way that you can do with the Table Storage service, Table API allows you to store information in your Cosmos DB account using a data structure based on documents.
- **Cassandra** Cosmos DB implements the wire protocol for the Apache Cassandra database into the options for storing and accessing data in the Cosmos DB database. This allows you to forget about operations and performance-management tasks related to managing Cassandra databases. In most situations, you can migrate your application from your current Cassandra database to Cosmos DB using the Cassandra API by merely

changing the connection string. Azure Cosmos DB Cassandra API is compatible with the CQLv4 wire protocol. Cassandra is a column-based database that stores information using a key-value approach.

- **MongoDB** You can access your Cosmos DB account by using the MongoDB API. This NoSQL database allows you to store the information for your application in a document-based structure. Cosmos DB implements the wire protocol compatible with MongoDB 3.2. This means that any MongoDB 3.2 client driver that implements and understands this protocol definition can connect seamlessly with your Cosmos DB database using the MongoDB API.
- **Gremlin** Based on the Apache TinkerPop graph transversal language or Gremlin, this API allows you to store information in Cosmos DB using a graph structure. This means that instead of storing only entities, you store
  - **Vertices** You can think of a vertex as an entity in other information structures. In a typical graph structure, a vertex could be a person, a device, or an event.
  - **Edges** These are the relationships between vertices. A person can know another person, a person might own a type of device, or a person may attend an event.
  - **Properties** These are each of the attributes that you can assign to a vertex or an edge.

Beware that you cannot mix these APIs in a single Cosmos DB account. You need to define the API that you want to use for accessing your Cosmos DB account when you are creating the account. Once you have created the account, you won't be able to change the API for accessing it.

Azure offers SDKs for working with the different APIs that you can use for connecting to Cosmos DB. Supported languages are .NET, Java, Node.js, and Python. Depending on the API that you want to use for working with Cosmos DB, you can also use other languages like Xamarin, Golang, or PHP. In this section, you can review an example of each API and learn how to create, read, update, and delete data using the different APIs.

Before starting with the examples, you need to create a Cosmos DB account for storing your data. The following procedure shows how to create a Cosmos DB free account with the SQL API. You can use this same procedure for creating accounts with the other APIs we have reviewed in this skill:

1. Sign in to the Azure portal (*http://portal.azure.com*).
2. In the top-left corner in the Azure portal, click the menu icon represented by three horizontal bars, and then click Create A Resource.
3. On the New panel, under the Azure Marketplace column, click Databases. On the Featured column, click Azure Cosmos DB.
4. On the Create Azure Cosmos DB Account blade, in the Resource Group drop-down menu, click the Create New link below the drop-down menu. In the pop-up dialog box, type a name for the new Resource Group. Alternatively, you can select an existing Resource Group from the drop-down menu.

5. In the Instance Details section, type an Account Name.
6. In the API drop-down menu, ensure that you have selected the option Core (SQL), as shown in Figure 2-1.

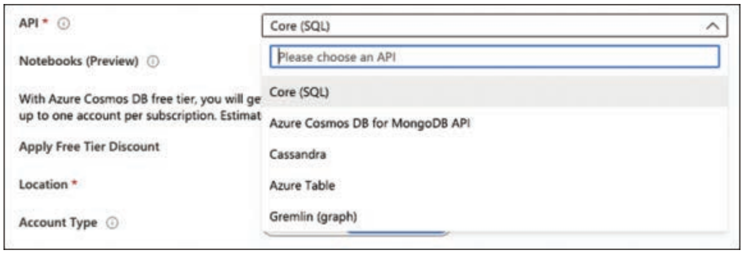

API * Core (SQL)
Notebooks (Preview)
With Azure Cosmos DB free tier, you will ge
up to one account per subscription. Estimat
Apply Free Tier Discount
Location *
Account Type

FIGURE 2-1 Selecting a Cosmos DB API

7. Ensure that the Notebooks switch is set to Off.
8. Ensure that the Apply Free Tier Discount switch is set to Apply.
9. On the Location drop-down menu, select the region most appropriate for you. If you are using App Services or virtual machines, you should select the same region in which you deployed those services.
10. In the Account Type, set the value Non-Production.
11. Leave Geo-Redundancy and Multi-Region Write disabled.
12. In the bottom-left corner of the Create Azure Cosmos DB Account blade, click the Review + Create button.
13. In the bottom-left corner of the Review + Create tab, click the Create button to start the deployment of your Cosmos DB account.

NOTE **AZURE COSMOS DB EMULATOR**

You can use the Azure Cosmos DB emulator during the development stage of your application. You should bear in mind that there are some limitations when working with the emulator instead of a real Cosmos DB account. The emulator is only supported on Windows platforms or Docker for Windows. You can review all characteristics of the Cosmos DB emulator at *https://docs.microsoft.com/en-us/azure/cosmos-db/local-emulator*.

EXAM TIP

You can use different APIs for accessing your Cosmos DB database. Each API offers different feature depending on the way you need to represent your data. Remember that you cannot change the API once you have created your Cosmos DB database.

Implement partitioning schemes

When you save data to your Cosmos DB account—independently of the API that you decide to use for accessing your data—Azure places the data in different servers to accommodate the performance and throughput that you require from a premium storage service like Cosmos DB. The storage services use partitions to distribute the data. Cosmos DB slices your data into smaller pieces called partitions that are placed on the storage server. There are two different types of partitions when working with Cosmos DB:

- **Logical** You can divide a Cosmos DB container into smaller pieces based on your criteria. Each of these smaller pieces is a logical partition. All items stored in a logical partition share the same partition key.
- **Physical** These partitions are a group of replicas of your data that is physically stored on the servers. Azure automatically manages this group of replicas or replica sets. A physical partition can contain one or more logical partitions.

NEED MORE REVIEW? **PHYSICAL PARTITION**

The only control that you have on how the data is distributed across physical partitions is setting the partition keys. If you want to review how the logical partitions and physical partitions are related to each other, consult the following article: *https://docs.microsoft.com/en-us/azure/cosmos-db/partition-data#physical-partitions*.

By default, any logical partition has a limit of 20 GB for storing data. When you are configuring a new collection, you need to decide whether you want your collection to be stored in a single logical partition and keep it under the limit of 20 GB or allow it to grow over that limit and span across different logical partitions. If you need your container to split over several partitions, Cosmos DB needs some way to know how to distribute your data across the different logical partitions. This is where the partition key comes into play. Bear in mind that this partition key is immutable, which means you cannot change the property that you want to use as the partition key once you have selected it.

Choosing the correct partition key is critical for achieving the best performance. The reason choosing the proper partition key is so important is because Azure creates a logical partition for each distinct value of your partition key. Listing 2-1 shows an example of a JSON document.

LISTING 2-1 Example JSON document

```
{
    "id": "1",
    "firstName": "Santiago",
    "lastName": "Fernández",
    "city": "Sevilla",
    "country": "Spain"
}
```

Depending on your data, city or country properties would be the right choice for the partition key. You can find in your data that some documents have the same value for the country property, so they are stored together in the same logical partition. Using the id property as the partition key means that you end with a logical partition with a single document on each partition. This configuration can be beneficial when your application usually performs read workloads and uses parallelization techniques for getting the data.

On the other hand, if you select a partition key with just a few possible values, you can end with "hot" partitions. A "hot" partition is a partition that receives most of the requests when working with your data. The main implication for these "hot" partitions is that they usually reach the throughput limit for the partition, which means you need to provision more throughput. Another potential drawback is that you can reach the limit of 20 GB for a single logical partition. Because a logical partition is the scope for efficient multidocument transactions, selecting a partition key with a few possible values allows you to execute transactions on many documents inside the same partition.

Use the following guidelines when selecting your partition key:

- The storage limit for a single logical partition is 20 GB. If you foresee that your data would require more space for each value of the partition, you should select another partition key.
- The requests to a single logical partition cannot exceed the throughput limit for that partition. If your requests reach that limit, they are throttled to avoid exceeding the limit. If you reach this limit frequently, you should select another partition key because there is a good chance that you have a "hot" partition. The minimum throughput limit is different from databases to containers. The minimum throughput for databases is 100 request units per second (RU/s). The minimum throughput for containers is 400 RU/s.
- Choose partition keys with a wide range of values and access patterns that can evenly distribute requests across logical partitions. This allows you to achieve the right balance between being able to execute cross-document transactions and scalability. Using timestamp-based partition keys is usually a lousy choice for a partition key.
- Review your workload requirements. The partition key that you choose should allow your application to perform well on reading and writing workloads.
- The parameters that you usually use on your requests and filtering queries are good candidates for a partition key.

NEED MORE REVIEW? **PARTITIONING**

You can review more information about how partitioning works reviewing the following article: *https://docs.microsoft.com/en-us/azure/cosmos-db/partitioning-overview*

There could be situations where none of the properties of your items are appropriate for the partition keys. In those situations, you can create synthetic partition keys. A synthetic partition key is a key compound of two concatenated properties. In our previous document example shown in Listing 2-1, you created a new property named partitionKey containing a string that concatenates the values of city and country. For the example document, the value of the partitionKey should be *Sevilla-Spain***.**

EXAM TIP

Remember that your data is distributed across the different logic partitions by using the partition key. For this reason, once you have chosen a partition key, you cannot change it.

Interact with data using the appropriate SDK

Cosmos DB allows you to access data using different types of APIs. Once you have your Cosmos DB account ready, you can start creating your databases and containers for working with data. Remember that once you choose the API for your Cosmos DB account, you cannot change it.

The following example shows how to create a console application using .NET Core. The first example uses Cosmos DB SQL API for creating, updating, and deleting some elements in the Cosmos DB account:

1. Open Visual Studio Code and create a directory for storing the example project.
2. Open the Terminal, switch to the project's directory, and type the following command:

   ```
   dotnet new console
   ```

3. Install the NuGet package for interacting with your Cosmos DB account using the SQL API. Type the following command in the Terminal:

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

4. Change the content of the Program.cs file using the content provided in Listing 2-2. You need to change the namespace according to your project's name.
5. Sign in to the Azure portal (*http://portal.azure.com*).
6. 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.
7. On your Cosmos DB Account blade, in the Settings section, click Keys.
8. On the Keys panel, copy the URI and Primary Keys values from the Read-Write Keys tab. You need to provide these values to the EndpointUri and Key Constants in the code shown in Listing 2-2. (The most important parts of the code are shown with bold format.)

LISTING 2-2 Cosmos DB SQL API example

```
//C# .NET Core. Program.cs
using System.Collections.Immutable;
using System.Xml.Linq;
using System.Diagnostics;
using System.Runtime.CompilerServices;
```

```
using System;

using System.Linq;
using Microsoft.Azure.Cosmos;
using System.Threading.Tasks;
using ch2_1_3_SQL.Model;
using System.Net;

namespace ch2_1_3_SQL
{
    class Program
    {
        private const string EndpointUri = "<PUT YOUR ENDPOINT URL HERE>";
        private const string Key = "<PUT YOUR COSMOS DB KEY HERE>";
        private CosmosClient client;
        private Database database;
        private Container container;

        static void Main(string[] args)
        {

            try
            {
                Program demo = new Program();
                demo.StartDemo().Wait();
            }
            catch (CosmosException ce)
            {
                Exception baseException = ce.GetBaseException();
                System.Console.WriteLine($"{ce.StatusCode} error ocurred:
                {ce.Message}, Message: {baseException.Message}");
            }
            catch (Exception ex)
            {
                Exception baseException = ex.GetBaseException();
                System.Console.WriteLine($"Error ocurred: {ex.Message}, Message:
                {baseException.Message}");
            }

        }

        private async Task StartDemo()
        {
            Console.WriteLine("Starting Cosmos DB SQL API Demo!");

            //Create a new demo database
            string databaseName = "demoDB_" + Guid.NewGuid().ToString().
Substring(0, 5);

            this.SendMessageToConsoleAndWait($"Creating database {databaseName}...");

            this.client = new CosmosClient(EndpointUri, Key);
            this.database = await this.client.CreateDatabaseIfNotExistsAsync
(databaseName);

            //Create a new demo collection inside the demo database.
            //This creates a collection with a reserved throughput. You can customize
the options using a ContainerProperties object
            //This operation has pricing implications.
            string containerName = "collection_" + Guid.NewGuid().ToString().
Substring(0, 5);


            this.SendMessageToConsoleAndWait($"Creating collection demo
{containerName}...");

            this.container = await this.database.CreateContainerIfNotExistsAsync
(containerName, "/LastName");

            //Create some documents in the collection
            Person person1 = new Person
            {
                Id = "Person.1",
                FirstName = "Santiago",
                LastName = "Fernandez",
                Devices = new Device[]
                {

                     new Device { OperatingSystem = "iOS", CameraMegaPixels = 7,
                     Ram = 16, Usage = "Personal"},
                     new Device { OperatingSystem = "Android", CameraMegaPixels = 12,
                     Ram = 64, Usage = "Work"}
                },
                Gender = "Male",
                Address = new Address
                {
                    City = "Seville",
                    Country = "Spain",
                    PostalCode = "28973",
                    Street = "Diagonal",
                    State = "Andalucia"
                },
                IsRegistered = true
            };


            await this.CreateDocumentIfNotExistsAsync(databaseName, containerName,
person1);

            Person person2 = new Person
            {
                Id = "Person.2",
                FirstName = "Agatha",
                LastName = "Smith",
                Devices = new Device[]
                {

                     new Device { OperatingSystem = "iOS", CameraMegaPixels = 12,
                     Ram = 32, Usage = "Work"},
                     new Device { OperatingSystem = "Windows", CameraMegaPixels = 12,
                     Ram = 64, Usage = "Personal"}
                },
                Gender = "Female",
                Address = new Address
                {
                    City = "Laguna Beach",
                    Country = "United States",
                    PostalCode = "12345",
                    Street = "Main",
                    State = "CA"
                },
                IsRegistered = true
            };


            await this.CreateDocumentIfNotExistsAsync(databaseName, containerName,
person2);

            //Make some queries to the collection
            this.SendMessageToConsoleAndWait($"Getting documents from the collection
{containerName}...");

            //Find documents using LINQ

            IQueryable<Person> queryablePeople = this.container.GetItemLinqQueryable
<Person>(true)
                .Where(p => p.Gender == "Male");

            System.Console.WriteLine("Running LINQ query for finding men...");
            foreach (Person foundPerson in  queryablePeople)
            {
                System.Console.WriteLine($"\tPerson: {foundPerson}");
            }

            //Find documents using SQL

            var sqlQuery = "SELECT * FROM Person WHERE Person.Gender = 'Female'";

            QueryDefinition queryDefinition = new QueryDefinition(sqlQuery);
            FeedIterator<Person> peopleResultSetIterator = this.container.GetItemQuery
Iterator<Person>(queryDefinition);

            System.Console.WriteLine("Running SQL query for finding women...");
            while (peopleResultSetIterator.HasMoreResults)
            {
                FeedResponse<Person> currentResultSet = await peopleResultSetIterator.
ReadNextAsync();
                foreach (Person foundPerson in currentResultSet)
                {
                    System.Console.WriteLine($"\tPerson: {foundPerson}");
                }
            }

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

            //Update documents in a collection
            this.SendMessageToConsoleAndWait($"Updating documents in the collection
{containerName}...");
            person2.FirstName = "Mathew";
            person2.Gender = "Male";

            await this.container.UpsertItemAsync(person2);
            this.SendMessageToConsoleAndWait($"Document modified {person2}");

            //Delete a single document from the collection
            this.SendMessageToConsoleAndWait($"Deleting documents from the collection
{containerName}...");

            PartitionKey partitionKey = new PartitionKey(person1.LastName);
            await this.container.DeleteItemAsync<Person>(person1.Id, partitionKey);
            this.SendMessageToConsoleAndWait($"Document deleted {person1}");

            //Delete created demo database and all its children elements
            this.SendMessageToConsoleAndWait("Cleaning-up your Cosmos DB account...");
            await this.database.DeleteAsync();
        }
        private void SendMessageToConsoleAndWait(string message)
        {
            Console.WriteLine(message);
            Console.WriteLine("Press any key to continue...");
            Console.ReadKey();
        }


        private async Task CreateDocumentIfNotExistsAsync(string database,
string collection, Person person)
        {
            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 work with the SQL API, the Azure Cosmos DB SDK provides you with the appropriate classes for working with the different elements of the account. In the example shown in Listing 2-2, you need to create a CosmosClient object before you can access your Azure Cosmos DB account. The Azure Cosmos DB SDK also provides you with the classes Database and Container for working with these elements. When you need to create a Database or a Container, you can use CreateDatabaseIfNotExistsAsync or CreateContainerIfNotExistsAsync, respectively. These IfNotExists methods automatically check to determine whether the

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:

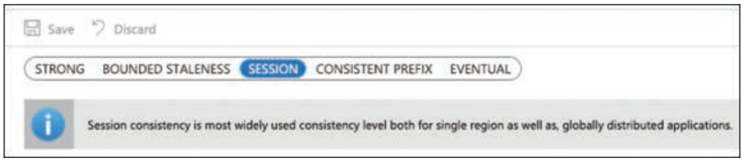
- **Quickstart: Build a .NET Framework or Core application Using the Azure Cosmos DB Gremlin API account** ***https://docs.microsoft.com/en-us/azure/cosmos-db/create-graph-dotnet***
- **Quickstart: Build a Cassandra App with .NET SDK and Azure Cosmos DB** ***https://docs.microsoft.com/en-us/azure/cosmos-db/create-cassandra-dotnet***

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.



Session consistency is most widely used consistency level both for single region as well as, globally distributed applications.

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.

NOTE **CONSISTENCY, AVAILABILITY, AND PERFORMANCE TRADEOFFS**

Every consistency level shown in this section has its implications in terms of data consistency, data availability, and application performanace. You can review the implications of choosing each of the consistency levels by reviewing the following article: *https://docs.microsoft.com/en-us/azure/cosmos-db/consistency-levels-tradeoffs*.

The best consistency level choice depends on your application and the API that you want to use to store data. As you can see in the different consistency levels, your application's requirements regarding data read consistency versus availability, latency, and throughput are critical factors that you need to consider when making your selection.

You should consider the following points when you use SQL or Table API for your Cosmos DB account:

- The recommended option for most applications is the level of session consistency.
- If you are considering the strong consistency level, we recommend that you use the bonded staleness consistency level because it provides a linearizability guarantee with a configurable delay.
- If you are considering the eventual consistency level, we recommend that you use the consistent prefix consistency level because it provides comparable levels of availability and latency with the advantage of guaranteed read orders.
- Carefully evaluate the strong and eventual consistency levels because they are the most extreme options. In most situations, other consistency levels can provide a better balance between performance, latency, and data consistency.

NEED MORE REVIEW? **CONSISTENCY LEVELS TRADE-OFF**

Each consistency level comes at a price. You can review the implications of choosing each consistency level by reading the article "Consistency, Availability, and Performance Tradeoffs" at *https://docs.microsoft.com/en-us/azure/cosmos-db/consistency-levels-tradeoffs*.

When you use Cassandra or MongoDB APIs, Cosmos DB maps the consistency levels offered by Cassandra and MongoDB to the consistency level offered by Cosmos DB. The reason for doing this is because when you use these APIs, neither Cassandra nor MongoDB offers a well-defined consistency level. Instead, Cassandra provides write or read consistency levels that map to the Cosmos DB consistency level in the following ways:

- **Cassandra write consistency level** This level maps to the default Cosmos DB account consistency level.
- **Cassandra read consistency level** Cosmos DB dynamically maps the consistency level specified by the Cassandra driver client to one of the Cosmos DB consistency levels.

On the other hand, MongoDB allows you to configure the following consistency levels: Write Concern, Read Concern, and Master Directive. Similar to the mapping of Cassandra consistency levels, Cosmos DB consistency levels map to MongoDB consistency levels in the following ways:

- **MongoDB write concern consistency level** This level maps to the default Cosmos DB account consistency level.
- **MongoDB read concern consistency level** Cosmos DB dynamically maps the consistency level specified by the MongoDB driver client to one of the Cosmos DB consistency levels.
- **Configuring a master region** You can configure a region as the MongoDB "master" by configuring the region as the first writable region.

NEED MORE REVIEW? **CASSANDRA AND MONGODB CONSISTENCY LEVEL MAPPINGS**

You can review how the different consistency levels map between Cassandra and MongoDB and Cosmos DB consistency levels in the article "Consistency Levels and Azure Cosmos DB APIs" at *https://docs.microsoft.com/en-us/azure/cosmos-db/consistency-levels-across-apis*.

EXAM TIP

The consistency level impacts the latency and availability of the data. In general terms, you should avoid the most extreme levels as they have a more significant impact on your program that should be carefully evaluated. If you are unsure of which level of consistency should use, you should use the session level, as this is the best-balanced level.

Create Cosmos DB containers

When you are working with Cosmos DB, you have several layers in the hierarchy of entities managed by the Cosmos DB account. The first layer is the Azure Cosmos DB account, where you choose the API that you want to use for accessing your data. Remember that this API has implications about how the data is stored in the databases.

The second layer in the hierarchy is the database. You can create as many databases as you need in your Cosmos DB account. Databases are a way of grouping containers, and you can think in databases like in namespaces. At this level, you can configure the throughput associated to the containers included in the database. Depending on the API that you are using, the database has a different name:

- **SQL API** Database.
- **Cassandra API** Keyspace.
- **MongoDB API** Database.

- **Gremlin API** Database.
- **Table API** This concept does not apply to Table API, although under the hood when you create your first Table, Cosmos DB creates a default database for you.

A container in an Azure Cosmos DB account is the unit of scalability for throughput and storage. When you create a new container, you need to set the partition key for establishing the way that the items that are going to be stored in the container are distributed across the different logical and physical partitions. As we reviewed in the "Implement partitioning schemes" section earlier in this chapter, the throughput is distributed across the logical partitions defined by the partition key.

When you create a new container, you can decide if the throughput for the container is one of the two following modes:

- **Dedicated** All the throughput is provisioned for a container. In this mode, Azure makes a reservation of resources for the container that is backed by SLAs.
- **Shared** The throughput is shared between all the containers configured in the database, excluding those containers that have been configured as dedicated throughput mode. The shared throughput is configured at the database level.

When you create a Cosmos DB container, there are a set of properties that you can configure. These properties affect different aspects of the container or the way the items are stored or managed. The following list shows those properties of a container that can be configured. Bear in mind that not all properties are available for all APIs:

- **IndexingPolicy** When you add an item to a container, by default, all the properties of the item are automatically indexed. It doesn't matter if all the items in the collection share the same schema, or each item has its own schema. This property allows you to configure how to index the items in the container. You can configure different types of indexes and include or exclude some properties from the indexes.
- **TimeToLive (TTL)** You can configure your container to delete items after a period of time automatically. TimeToLive is expressed in seconds. You can configure the TTL value at the container or item level. If you configure the TTL at the container level, all items in the container have the same TTL, except if you configure a TTL for a specific item. A value of -1 in the TTL means that the item does not expire. If you set a TTL value to an item where its container does not have a TTL value configured, then the TTL at item level has no effect.
- **ChangeFeedPolicy** You can read the changes made to an item in a container. The change feed provides you with the original and modified values of an item. Because the changes are persisted, you can process the changes asynchronously. You can use this feature for triggering notifications or calling APIs when a new item is inserted or an existing item is modified.
- **UniqueKeyPolicy** You can configure which property of the item is used as the unique key. Using unique keys, you ensure that you cannot insert two items with the same value for the same item. Bear in mind that the uniqueness is scoped to the logical partition.

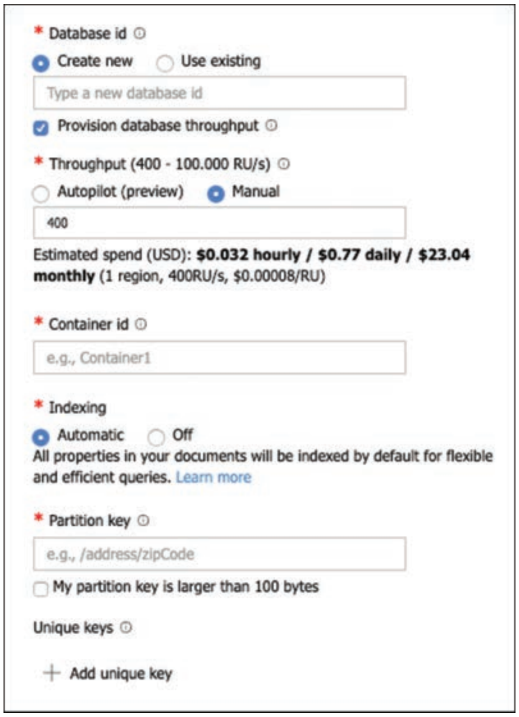
For example, if your item has the properties email, firstname, lastname, and company, and you define email as the unique key and company as the partition key, you cannot insert an item with the same email and company values. You can also create compound unique keys, like email and firstname. Once you have created a unique key, you cannot change it. You can only define the unique key during the creation process of the container.

NOTE **CONTAINERS' PROPERTIES**

The properties available to the containers depends on the API that you configured for your Azure Cosmos DB account. For a complete list of properties available for each API please review the article at *https://docs.microsoft.com/en-us/azure/cosmos-db/databases-containers-items#azure-cosmos-containers*.

Use the following procedure to create a new collection in your Cosmos DB account. This procedure could be slightly different depending on the API that you use for your Cosmos DB account. In this procedure, you use a Cosmos DB account configured with the SQL API:

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, click the New Container icon in the top-left corner of the blade.
5. On the Add Container panel, shown in Figure 2-3, provide a name for the new database. If you want to add a container to an existing database, you can select the database by clicking the Use Existing radio button.
6. Ensure that the Provision database throughput check is selected. Using this option, you are configuring this container as a shared throughput container. If you want to create a dedicated throughput container, uncheck this option.
7. Leave the Throughput value set to 400. This is the value for the database throughput if the previous option is checked. Otherwise, this value represents the dedicated throughput reserved for the container.
8. In the Container Id text box, type a name for the container.
9. In the Partition Key text box, type a partition key, starting with the slash character.
10. If you want to create a unique key for this container, click the Add Unique Key button.
11. Click the OK button at the bottom of the panel.



* Database id ⓘ

[✓ Create new] Use existing

Type a new database id

[✓] Provision database throughput ⓘ

* Throughput (400 - 100.000 RU/s) ⓘ

Autopilot (preview) [✓ Manual]

400

Estimated spend (USD): **$0.032 hourly / $0.77 daily / $23.04 monthly** (1 region, 400RU/s, $0.00008/RU)

* Container id ⓘ

e.g., Container1

* Indexing

[✓ Automatic] Off

All properties in your documents will be indexed by default for flexible and efficient queries. Learn more

* Partition key ⓘ

e.g., /address/zipCode

[ ] My partition key is larger than 100 bytes

Unique keys ⓘ

FIGURE 2-3 Creating a new collection

NEED MORE REVIEW? **TIME TO LIVE, INDEXES, AND CHANGES FEED**

You can review the details of how to configure the Time To Live, Index Policies, and Changes Feed by reading the following articles:

- **Configure Time to Live in Azure Cosmos DB** *https://docs.microsoft.com/en-us/azure/cosmos-db/how-to-time-to-live*
- **Unique Key Constraints in Azure Cosmos DB** *https://docs.microsoft.com/en-us/azure/cosmos-db/unique-keys*
- **Change Feed Design Patterns in Azure Cosmos DB** *https://docs.microsoft.com/en-us/azure/cosmos-db/change-feed-design-patterns*

EXAM TIP

You need to plan carefully how to create a new container in Azure Cosmos DB. You can set some of the properties that you can configure only during the creation process. Once you have created the container if you need to modify those properties, you need to create a new container with the needed values and migrate the data to the new container.

Implement server-side programming including stored procedures, triggers, and change feed notifications

When you work with Cosmos DB API, Azure allows you to write your triggers, stored procedures, and user-defined functions. You can write these procedures and functions using JavaScript. Before you can call a stored procedure, trigger, or user-defined function, you need to register it. You can use the Azure portal, the JavaScript language integrated query API in Cosmos DB, or the Cosmos DB SQL API client SDK for creating and calling your stored procedures, triggers, and user-defined functions.

Any stored procedure, trigger, or user-defined function that you write is registered on a container basis. That means that you need to register the stored procedure on each container where you want to execute your stored procedure. You also need to consider that stored procedures and triggers are scoped to partitions. Any item with a partition key value different from the partition key of the item that fired the trigger or the stored procedure is not visible.

When you are writing a stored procedure, trigger, or user-defined function, you need to create a reference to the execution context. This context gives you access to the requests that fired the stored procedure or trigger and allows you to work with the responses and items that you want to insert into the database. In general terms, the context gives you access to all the operations that you can perform in the Azure Cosmos DB database. The following procedure shows how to create a stored procedure in an Azure Cosmos DB SQL API account:

1. Open the Azure portal (*https://portal.azure.com*).
2. In the Search text box in the top area of the portal, type the name of your Cosmos DB account. Remember that this needs to be an SQL API Cosmos DB account.
3. On your Cosmos DB SQL API account, click Data Explorer.
4. Click an existing database. If you don't have any database, create a new one for testing purposes.
5. Click an existing container, or you can create a testing container following the procedure that we reviewed in a previous section.
6. Click the New Stored Procedure button. This button creates a new sample stored procedure that you can use as a template for your stored procedures.
7. In the Stored Procedure Id text box, provide a name for the stored procedure.
8. Replace the content of the New Stored Procedure tab with the content of Listing 2-6.

LISTING 2-6 Cosmos DB SQL API stored procedure

```
//JavaScript
function createNewDocument(docToCreate) {
    var context = getContext();
    var container = context.getCollection();
    var response = context.getResponse();

    console.log(docToCreate);
    var accepted = container.createDocument(container.getSelfLink(),
        docToCreate,
        function (err, docCreated) {
            if (err) throw new Error('Error creating a new document: ' + err.message);
            response.setBody(docCreated);
        });

    if (!accepted) return;

}
```

9. Click the Save button.
10. Click the Execute button. This button opens the Input Parameters blade.
11. In the Input Parameters blade, in the Partition Key Value section, change the type from Custom to String.
12. In the Partition Key Value section, in the Value text box, type a value to the partition key. Remember that this partition key is the one that you have defined for the container where you are creating this stored procedure.
13. In the Type drop-down menu in the Enter Input Parameters section, ensure that the value String is selected.
14. In the Param text box, type the document in JSON format that you want to insert. For the sake of simplicity, use a string with a structure similar to {"id": "12345", "key": "value"}.
15. Click the Execute button at the bottom of the Input Parameters panel.
16. In the Data Explorer navigation tree, click on the Items leaf below the container where you are creating the stored procedure.
17. Ensure that the new document has been correctly inserted in your container.

NOTE **BADREQUEST ERROR**

If you get a BadRequest error when you execute the previous example, review the values of the input parameters. Remember that you cannot insert a document in a different partition from the one that you select in the Partition Key Value. For example, if your partition key is the field "city" and the value you provide is "Seville", you need to include this value in the Enter Input Parameters section. For this example, your document should look similar to { "country": "Spain", "city": "Seville"}.

Although the previous example is quite simple, there are some interesting points that we should review. One of the essential points that you need to consider when programming your stored procedures, user-defined functions or trigger, is the fact that the input parameters always have the string type. This means that if you need to pass an object to the stored procedure, you need to stringify the object, and then convert back to a JSON object by using the JSON.parse() method.

As you can see, we use the global getContext() method for getting a reference to the context. That context gives us access to the features of the Cosmos DB account. Then we got a reference to the current container by using the getContainer() method in the context. We also use the getResponse() method from the context for sending back information to the client.

Because we are going to create a new document in the container, we need to use the createDocument() method in the container. This method requires a link to the container where we are going to insert the document, and the document itself. Because the methods require a JSON document, if the value of the input parameter is not a valid JSON string, you get a JSON parse error here. We also provided an optional anonymous function for managing any error that may arise during the creation of the document. If you don't provide a callback function, any error is thrown as an exception.

Creating a trigger is quite similar to create a stored procedure. The concepts are equivalent, but you need to consider when you need to execute the action of your trigger. If you need to make an operation before the item is inserted into the container, you need to use a **pre-trigger**. If you need to make an action after the item has been successfully inserted in the container, you need to use a **post-trigger**.

Pre-triggers cannot have input parameters. Because the item is not actually in the database, you need to work with the request that fired the trigger. This request contains the information needed for inserting the new item into the collection. You can get a reference to the request by using the getRequest() method from the context object. Once you have made your modifications to the original item, you can send the modified item to the database by using the request.setBody() method.

NEED MORE REVIEW? **MORE SAMPLES**

Although the sample that we reviewed in this section could seem simplistic, it covers some important points that you need to be aware of when programming your server-side items. The following articles provide more detailed examples of how to create and register stored procedures, user-defined functions, or triggers using JavaScript or C#:

- **How to write stored procedures, triggers, and user-defined functions in Azure Cosmos DB** *https://docs.microsoft.com/en-us/azure/cosmos-db/how-to-write-stored-procedures-triggers-udfs*
- **How to write stored procedures and triggers in Azure Cosmos DB by using the JavaScript query API** *https://docs.microsoft.com/en-us/azure/cosmos-db/how-to-write-javascript-query-api*
- **How to register and use stored procedures, triggers, and user-defined functions in Azure Cosmos DB** *https://docs.microsoft.com/en-us/azure/cosmos-db/how-to-use-stored-procedures-triggers-udfs*

Index