Azure Cosmos DB and DocumentDB

Succinctly®

by Ed Freitas

Syncfusion | Technology Resource Portal

Azure Cosmos DB and DocumentDB Succinctly

By
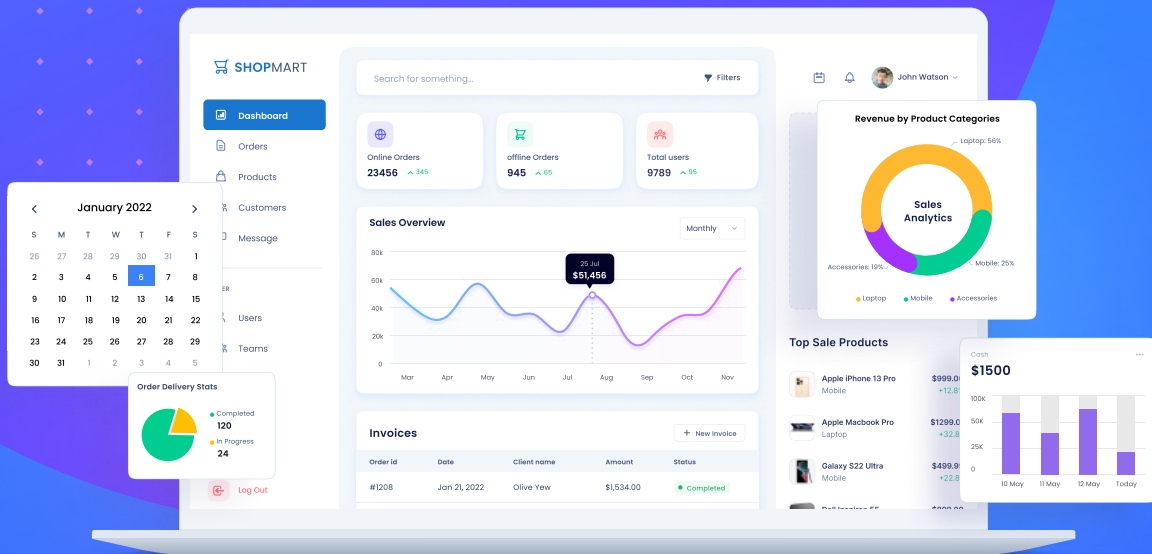**Ed Freitas**

Foreword by Daniel Jebaraj

Technical Reviewer: James McCaffrey

Copy Editor: Jacqueline Bieringer

Acquisitions Coordinator: Morgan Weston, social media marketing manager, Syncfusion, Inc.

Proofreader: Graham High, senior content producer, Syncfusion, Inc.

Table of Contents

The Story behind the *Succinctly* Series of Books

Daniel Jebaraj, Vice President
Syncfusion, Inc.

Staying on the cutting edge

As many of you may know, Syncfusion is a provider of software components for the Microsoft platform. This puts us in the exciting but challenging position of always being on the cutting edge.

Whenever platforms or tools are shipping out of Microsoft, which seems to be about every other week these days, we have to educate ourselves, quickly.

Information is plentiful but harder to digest

In reality, this translates into a lot of book orders, blog searches, and Twitter scans.

While more information is becoming available on the Internet and more and more books are being published, even on topics that are relatively new, one aspect that continues to inhibit us is the inability to find concise technology overview books.

We are usually faced with two options: read several 500+ page books or scour the web for relevant blog posts and other articles. Just as everyone else who has a job to do and customers to serve, we find this quite frustrating.

The *Succinctly* series

This frustration translated into a deep desire to produce a series of concise technical books that would be targeted at developers working on the Microsoft platform.

We firmly believe, given the background knowledge such developers have, that most topics can be translated into books that are between 50 and 100 pages.

This is exactly what we resolved to accomplish with the *Succinctly* series. Isn't everything wonderful born out of a deep desire to change things for the better?

The best authors, the best content

Each author was carefully chosen from a pool of talented experts who shared our vision. The book you now hold in your hands, and the others available in this series, are a result of the authors' tireless work. You will find original content that is guaranteed to get you up and running in about the time it takes to drink a few cups of coffee.

Free forever

Syncfusion will be working to produce books on several topics. The books will always be free. Any updates we publish will also be free.

Free? What is the catch?

There is no catch here. Syncfusion has a vested interest in this effort.

As a component vendor, our unique claim has always been that we offer deeper and broader frameworks than anyone else on the market. Developer education greatly helps us market and sell against competing vendors who promise to "enable AJAX support with one click," or "turn the moon to cheese!"

Let us know what you think

If you have any topics of interest, thoughts, or feedback, please feel free to send them to us at succinctly-series@syncfusion.com.

We sincerely hope you enjoy reading this book and that it helps you better understand the topic of study. Thank you for reading.

Please follow us on Twitter and "Like" us on Facebook to help us spread the word about the *Succinctly* series!

About the Author

Ed Freitas currently works as a consultant. He was recently involved in analyzing 1.6 billion rows of data using Redshift (Amazon Web Services) in order to gather valuable insights on client patterns. He holds an M.S. in Computer Science. He enjoys soccer, running, traveling, and life hacking. You can reach him at http://bit.ly/1UtECxw.

Acknowledgments

My thanks to all the people who contributed to this book, especially Tres Watkins, Morgan Weston, and Darren West, the Syncfusion team that helped this book become a reality. Darren West, the manuscript manager, and James McCaffrey, the technical editor, thoroughly reviewed the book's organization, code quality, and overall accuracy. Thank you all.

Introduction

Azure Cosmos DB from Microsoft is a globally-distributed, multi-model data service that lets you elastically scale throughput and storage across any number of geographical regions using low latency, high availability, and consistency. Azure Cosmos DB's database engine natively supports DocumentDB's SQL dialect, MongoDB API, Gremlin (graph) API, and Azure Table storage APIs.

In this book, we'll focus on the DocumentDB part of Cosmos DB, which is Microsoft's NoSQL document database platform that runs on Azure. SQL stands for Structured Query Language and it's an acronym associated with relational databases, as it is the standard and most widely used language for working with relational databases.

By using the term NoSQL, what we really want to express is that we are referring to nonrelational databases. Nonrelational databases abandon many of the concepts that traditional relational databases use, in particular how data is structured and organized into a tabular format where each column has a specific data type. Tables contain rows, all with the same number of columns, each representing a record.

There are several types of nonrelational databases, such as key-value stores, column stores, and graph and document databases. Azure Table storage is a key-value store, Cassandra is a column store, and Neo4j is a graph database. MongoDB and DocumentDB are document databases.

Although there are key differences among the various types of NoSQL databases, they also have traits in common.

NoSQL databases are designed to scale out and not just up, meaning that while relational databases can scale up by simply adding additional hardware resources, it is more difficult to scale a relational database horizontally. It is not easy to spread the data among various partitions once you start hitting the ceiling on CPU, disk, and memory, and you can no longer scale upwards.

Contrary to relational databases, NoSQL databases are architected to scale out as far as needed, making it much easier to achieve Internet scale. They are not bound to a rigid tabular and columnar structure, so the data can be organized in a schema-free way.

Schema-free means that we no longer need what we would traditionally think of as records on the database. We are free to store information in a way that might not be the same for each record, so we could have different numbers and types of columns, not only with different values, but also different data types.

By design, NoSQL databases were built for simplicity, in terms of what types of information can be written. They are not nearly as robust as traditional relational databases like SQL Server or Oracle, and they don't even attempt to mirror or provide the complete functionality of these. This is why they are able to outperform relational databases on a large scale, by easily scaling out and not just up.

Because NoSQL databases are schema-free, scalable, and easy to use, they give us more choices for our applications than we had before. Traditional relational databases are definitely here to stay. However, they no longer hold a firm monopoly as the default choice when we need to choose a database for our application development needs.

DocumentDB is a document nonrelational database that has been designed for JavaScript Object Notation (JSON) objects (documents) with tagged elements that can be both serialized and deserialized, typically schema-free.

A document in NoSQL terms refers to a JSON object and should not be confused with the typical meaning of the word "document," which traditionally refers to PDF, Word, and other similar document file formats.

In this e-book, we will explore DocumentDB's features, how it compares to traditional relational databases, how it can be used to create scalable applications, and how using many of Azure's built-in functionalities allows it to be a great option for Internet applications.

The examples in this e-book will be mostly done with Visual Studio 2017 using C#; however, some parts will be covered using DocumentDB's built-in query language DocumentDB SQL, which is very similar to standard SQL, and JSON will be also used. On the server side, JavaScript will be used.

In order to maximize your learning potential, it is assumed that you have existing knowledge of relational databases and SQL, as well as proficiency with C#, and are familiar with JSON and JavaScript. The Visual Studio code samples can be found here.

Throughout this book the terms Cosmos DB and DocumentDB may be used interchangeably, as Cosmos DB originated from DocumentDB.

The journey should be fun to follow, the examples easy to implement, and the skills useful for helping you understand what is possible with this amazing NoSQL database. Enjoy!

Chapter 1 DocumentDB Basics

Introduction

DocumentDB, as its name implies, stores data as "documents," which are actually JSON objects.

In a relational database, records are stored as rows on a table with specific columns using a defined schema. However, on a document store, records are documents and each one contains specific properties, schema-free.

To understand the differences between a relational and a document database, let's consider the following table.

Table 1-a: Main Differences between a Relational and Document Database

Relational Database	Document Database
Rows	Documents
Columns and data types	Properties
Strong-typed and defined schemas	Schema-free
Highly normalized	Mostly denormalized
Robust and mature	Simple and lean
Scales vertically (more hardware)	Scales out, horizontally

A relational database table has a fixed structure, and in order to make changes to a table, such as adding a new column, changing a specific column to allow NULL values, or changing a data type, it is necessary to modify the table's schema, which can create side effects for the existing data already stored in the table.

Because DocumentDB is schema-free, it is not subject to the constraints that relational databases are. In today's world, where application and software development are associated with frequently evolving data schemas, document databases are a great match.

Let's look at how some example data would be stored in a traditional relational table and how it would be stored in a document database.

Table 1-b: Example Data Stored in a Relational Database Table

FirstName	LastName	Age	Active
John	Doe	44	Active
Cristiano	Ronaldo	31	NULL
Peter	Pan	67	Retired

As we can see, the data is structured in a tabular format with all records having the same number of columns. Note that I use an Age column for simplicity; in a real database I'd use a Date-of-Birth column.

However, consider the record highlighted in yellow. This record has values in the FirstName, LastName, and Age columns, but the column Active doesn't have a value; therefore, it is set to NULL. The other records, contrary to the one in yellow, have values in all their columns.

Although the tabular format can accommodate records that do not necessarily have values in the same number of columns (by using NULL values in columns that do not have a value), it still requires those records to have that column, since the schema enforces it.

If we were to accommodate this same data in a nonrelational document database such as DocumentDB, it would look as follows.

Table 1-c: Example Data Stored in a Nonrelational Document Database

Document	Document Data

1	{
	    "FirstName": "John",
	    "LastName": "Doe",
	    "Age": 44,
	    "Active": "Active"
	}

2	{
	    "FirstName": "Cristiano",
	    "LastName": "Ronaldo",
	    "Age": 31
	}

3	{
	    "FirstName": "Peter",
	    "LastName": "Pan",
	    "Age": 67,
	    "Active": "Retired"
	}

For illustrative purposes only, the data in the previous table is shown in a tabular format, which resembles the format a relational database table uses. However, a document database does not store the data in a tabular format, but instead as a binary representation of the JSON data.

What is known as a table in a relational database context is called a collection of documents in a document database or, simply put, a collection.

Notice that the main differences between relational and nonrelational document databases are that in nonrelational document databases records are called documents and they only store values for properties being used (which would correspond to columns in a relational table). Therefore, NULL properties do not need to be represented within documents, as they are not enforced by a schema.

Furthermore, a nonrelational document database allows documents to store subproperties, which cannot be stored explicitly within a relational table. An example of this would be replacing the FirstName and LastName properties with a FullName property with two subproperties called First and Last.

Table 1-d: A Document with Subproperties

Document	Document Data
2	```{ "FullName": { "First": "Cristiano", "Last": "Ronaldo" }, "Age": 31 }```

This example shows how, by having subproperties, the document is not normalized in regards to other documents present in the same collection, but a document database can perfectly accommodate this.

This means that each document can have properties and subproperties that might not necessarily be present on other documents, which allows for a high degree of flexibility and denormalization of the data stored.

Master-detail nested properties

In nonrelational databases, master-detail relationships are defined by having nested properties. In a relational database, the only way to achieve this is by having a master-detail relationship between two tables, sharing a common field between them.

Let's consider this example of a master-detail relationship within a relational database.

Table 1-e: A Typical Master-Detail Relational Database Relationship

FirstName	LastName	Age	Active	PlayerId
John	Doe	44	Active	1
Cristiano	Ronaldo	31	NULL	2
Peter	Pan	67	Retired	3

PlayerId	TeamName
2	Real Madrid
2	Manchester United
1	Mars Galactic Soccer
3	Fantasy FC

In a relational database, like the example above, the only way to indicate that Cristiano Ronaldo has played with both Real Madrid and Manchester United is by having a second (detail) table that contains the names of both teams and adding an extra column to the master table with a unique PlayerId that identifies the record.

This PlayerId column will also exist on the detail table and it will be responsible for linking the detail records with the corresponding record on the master table. This is a standard master-detail relational database relationship.

In a document database, this master-detail relationship is represented as a property with nested properties. Let's explore how this would look for the record of the player Cristiano Ronaldo.

Table 1-f: A Document with Master-Detail Nested Properties

Document	Document Data

2	```
	{
	    "FirstName: "Cristiano",
	    "LastName": "Ronaldo",
	    "Age": 31,
	    "Teams": [
	        {
	            "Team": "Real Madrid"
	        },
	        {
	            "Team": "Manchester United"
	        }
	    ]
	}
	``

We can see that the teams that would be stored in the detail table in a traditional relational database are now stored within an array called Teams, in which each team is a nested JSON object that contains a subproperty called Team.

Basically, the master-detail relationship is now described as an array of JSON objects. However, just because the master-detail relationship can be easily described as an array of JSON objects in this case, doesn't mean it always has to be described as such.

When to use a document database

The power and beauty of nonrelational databases is that you are free to implement and represent the relation between master-detail records in any way that best suits your application and business requirements.

It is also not uncommon in document databases to repeat some data so that each document has the data it needs without having to locate other documents. If the data repeats itself too much, then you may choose to organize the repeating data in different documents, similar to a traditional relational database. In any case, you are free to organize the structure of your JSON documents to what works best for your requirements and applications.

In essence, a document database gives you the freedom to model your data in a way that best fits your needs.

Despite this flexibility, it's important to understand that document databases are most suitable when working with data that can be organized into rich hierarchical documents that can be almost entirely self-contained.

If you find yourself modelling a database containing many related documents or documents with a flat structure, this is a sign that a document database is probably not the best option for your application.

When you need a database that scales, a document database is a great option. The primary reasons document databases can scale out are that they don't impose complex or rigid rules on the data and they are simple and lean by design.

On the other hand, relational databases are better suited for handling complex requirements that do not necessarily have to scale out.

Why DocumentDB?

DocumentDB is Microsoft's highly scalable document database API that runs on Azure Cosmos DB. Although it has all the characteristics of a typical document database, it also has features that are not available on any other document database. Let's explore these features.

With DocumentDB, unlike other document databases where you explicitly need to define indexes, all properties are automatically indexed as soon as the document has been added to the database. This allows you to search any property within the document's hierarchy, however deeply nested it might be.

Furthermore, documents are searchable using a special flavor of SQL that anyone with SQL experience can easily grasp and relate to in an intuitive way—this is DocumentDB's very own SQL dialect.

Because DocumentDB runs on Azure Cosmos DB, it provides a server-side environment on which you can run JavaScript code that can update multiple documents with full transactional processing. This is a great and easy way to ensure data consistency among multiple documents.

Also, DocumentDB allows tunable performance for your application's requirements, such as enhancing throughput, indexing, and consistency. Throughput can be scaled up or down instantly by changing the performance tier through the Azure Portal.

Although DocumentDB automatically indexes every property, you can still fine tune the system to exclude any properties or documents that do not need to be indexed, which could even help to improve performance in very specific scenarios.

Even though DocumentDB supports traditional strong as well as eventual consistency (a slightly different form associated with distributed data systems), it also provides two additional options to give you greater control over the tradeoffs between performance and consistency.

All this functionality is nicely packaged as a fully managed and massively scalable Platform-as-a-Service (PaaS) solution that is very easy to setup and get started with. There's nothing to install, no operating system or updates to manage, and no replicas to set up.

Through the Azure Portal, you can get up and running with DocumentDB in a matter of minutes by using a browser and having an Azure subscription. Take the necessary time to set up your Azure subscription properly.

Sounds exciting, so let's explore some of these features with some additional details.

Rich SQL flavored queries

One of the best features of DocumentDB is that its native querying language is very similar to SQL. For those doing development with .NET, there's a LINQ provider as well.

Although DocumentDB's flavored queries are written in SQL, they are deeply rooted in JSON and JavaScript semantics. They allow you to query over hierarchical nested data and arrays within documents and also share custom projections from the results of your queries. Let's have a look at an example.

Code Listing 1-a: A DocumentDB SQL Flavored Query

```sql
-- SQL flavored query.
SELECT ch.First, ch.Last
FROM Players AS p
JOIN ch IN p.children
WHERE p.Age = 31
```

In this flavored SQL query example, we are doing a few things. The `JOIN` clause basically allows DocumentDB to iterate through all the `children` (properties) nested within the Players document.

The `WHERE` clause filters by checking the documents that have an `Age` property value equal to 31. Notice that dotted notation is used to refer to the properties within the document. The dotted notation can nest as far down as the document's hierarchy goes.

Because on the `SELECT` clause we are selecting two properties instead of a `SELECT *`, DocumentDB projects a new JSON object that only contains the properties being queried, instead of the whole document as the result.

Once a document is inserted into DocumentDB, it is pretty much searchable instantly, as it is automatically indexed. This indexing behavior can be tuned. However, there's usually no need.

Client-side development

DocumentDB provides various SDKs, which allow easy integration from your preferred development platform. There are SDKs for several of the most common development platforms, such as .NET, Node.js, JavaScript, Java, and Python.

It is no different than other platforms, as it also provides a REST/HTTP API that can be used as long as the headers of the HTTP request contain valid authentication information and the request points to the right HTTP DocumentDB resource.

Working with the REST/HTTP API is the most primitive way to interact with DocumentDB; it can become very tedious when you need to focus most of your attention on the logic of your application. Therefore, we will focus on using the .NET SDK in this e-book when doing client-side development.

If an SDK isn't available for your development platform, you can use the REST/HTTP API. However, you should also contact Azure support, let them know what platform you are using, and provide them your feedback. Hopefully, they can take your opinion into account and include an SDK for your development platform in their roadmap. The Azure team is very active and committed to the platform, so if there isn't an SDK for your platform, it is likely because nobody has asked for it yet.

Server-side development

DocumentDB is a server sandbox environment that gives you the ability to execute logic inside, where the data resides. Server-side logic in DocumentDB can be wrapped up into stored procedures and triggers, and user-defined functions (UDFs), which is strikingly familiar from working with relational database systems like Oracle and SQL Server.

However, there's a subtle difference between server-side logic written in DocumentDB and server-side logic written using traditional relational databases. In DocumentDB, server-side logic is written in JavaScript instead of SQL, which makes it a perfect companion to deal with JSON objects.

DocumentDB embraces JavaScript as a sort of modern-day SQL by supporting the transactional script execution natively inside the database engine.

Because DocumentDB is a fully hosted service, it cannot allow scripts that perform poorly to run indefinitely, as this could risk the integrity of the whole service. Therefore, it enforces a paradigm known as bounded execution, which basically determines how much time your logic can run before timing out.

All server-side logic execution is fully transactional, which means that if you update some documents and an error occurs, or one of your scripts times out due to bounded execution before it actually completes, then all updates up to that moment are rolled back automatically. If your server-side code completes successfully, then all updates are guaranteed to commit together. This makes DocumentDB an even more compelling option.

Scalability

With NoSQL databases, scalability is key to success and DocumentDB delivers. It is already the back-end of choice for services like Xbox and Office OneNote, which rely on DocumentDB for databases containing tens of terabytes of JSON documents, over a million users, and an uptime availability of 99.9%.

To give you a sense of how much DocumentDB can scale: it can grow as much as you can afford or to the end of the available hardware in Azure's data centers, whichever limit you reach first. This is a profound statement and a clear indication of what DocumentDB can handle and how much it can scale. It also serves as the foundation used for the creation of Cosmos DB.

In simple terms, DocumentDB can massively scale to hundreds of terabytes and even petabytes through thousands of nodes. It can scale up and down and also out.

It is able to scale up and down seamlessly, which consists of a combination of computing power and storage capacity.

DocumentDB is also able to scale out by adding more collections. A collection of documents can essentially be seen as a scale unit. If your database grows beyond 10 GB, then you can scale out by simply adding more collections and then partitioning your data across multiple collections.

Consistency

Another important feature that makes DocumentDB great is its ability to tune consistency. There's usually a tradeoff between performance and consistency.

Basically, strong consistency slows down reads and writes and eventual consistency doesn't always return the most current data. With strong consistency, you get consistent query results as writers make changes to the database, but you pay a price in performance, as all queries must wait until all replicas have been updated with the latest changes, which obviously slows things down a bit.

Conversely, eventual consistency gives you the best performance, but you cannot fully rely on query results. They might return data that is not entirely consistent with what other users might be updating, given that not all replicas are necessarily up to date.

DocumentDB supports both of these consistency methods and three additional methods that fall somewhat in the middle of strong and eventual consistencies. These three additional methods are called bounded staleness, session, and consistent prefix.

Bounded staleness lets you tolerate inconsistent query results by guaranteeing that those results are at least consistent enough within a specified period of time.

Session consistency, which is actually the default consistency method used within DocumentDB, can be thought of as a hybrid experience. Writers are guaranteed to have strong consistency for data that they have themselves written, while everyone else operates with eventual consistency.

Consistent prefix guarantees that in absence of any further writes, the replicas within the group eventually converge. Azure Cosmos DB accounts that are configured with consistent prefix consistency can associate any number of Azure regions with their Azure Cosmos DB account.

DocumentDB allows consistency to be tuned and changed, giving you the flexibility to work with the approach that best suits your business requirements and needs.

More information about tunable data consistency levels in Azure Cosmos DB using the DocumentDB API can be found [here](#).

Costs

Before the introduction of Cosmos DB, when DocumentDB was offered as a standalone service, costs were based on pre-set tiers and collections.

In DocumentDB prior to Cosmos DB, a collection was not only a unit of scale, but also directly related to costs and pricing. You would pay per collection, and each collection had a storage capacity of up to 10 GBs.

With the introduction of Azure Cosmos DB, the old pricing schema based on the combination of collections and tiers became irrelevant and not fully elastic.

The old DocumentDB S1, S2, and S3 performance levels did not offer the flexibility that DocumentDB API collections now offer going forward with Cosmos DB. This was because in the S1, S2, and S3 performance levels, both the throughput and storage capacity were pre-set and did not offer elasticity.

Azure Cosmos DB now offers the ability to customize your throughput and storage, offering you much more flexibility to scale as your application needs change. More information on this topic can be found [here](#).

Summary

In this chapter we've quickly learned about some of the most prominent NoSQL document database characteristics and specifically about DocumentDB and how it compares to traditional relational databases.

In particular, we focused on describing how DocumentDB is designed from the ground up to scale out and work with hierarchical JSON documents schema-free, unlike traditional tabular database tables that require a schema and complex joins to piece together.

We also quickly described how DocumentDB enables instant searching across documents and how every property is automatically indexed when a document is added to a collection, allowing you to quickly and easily query them using a familiar SQL-like syntax.

Furthermore, we discussed how client and server-side programming are possible through various platform-specific SDKs and using JavaScript with full transactional processing.

Finally, we described how DocumentDB permits tunable consistency and supports elastic scaling. We also talked briefly about costs.

This sets the stage for the following chapters and gives us a solid starting point and a high level understanding of what DocumentDB can allow us to accomplish. In the following chapters, we'll focus on diving deeper into each of DocumentDB's features and exploring each one of them by writing some code. Thanks for reading!

Chapter 2 First Steps with DocumentDB

Introduction

In the previous chapter, we had a bird's-eye view of DocumentDB, its main features, and why it's a great option when it comes to choosing a NoSQL database, given its capacity to massively, seamlessly scale.

In this chapter, we'll explore how we can get DocumentDB set up and ready to use, specifically how to create the DocumentDB account, database, and then our first collection. After this initial setup, we'll look at collections more in-depth and add documents to them.

In order to have data to later query and work with, we'll also look at using the DocumentDB Data Migration Tool to import data from other sources into DocumentDB. This is a great way to see how data can transition away from more traditional sources to a NoSQL JSON document database.

Before getting started, it is important to note that you'll need a Microsoft account (typically associated with a Hotmail or Live email account) and also an Azure subscription. Sounds exciting, so let's not wait any longer to get started setting up DocumentDB.

DocumentDB setup

The first step to getting DocumentDB up and running is to create an instance on Azure. If you don't have an Azure account, you'll need to sign up for Azure with a Microsoft account. You can do that by visiting azure.microsoft.com and then clicking the **PORTAL** option at the top of the page.
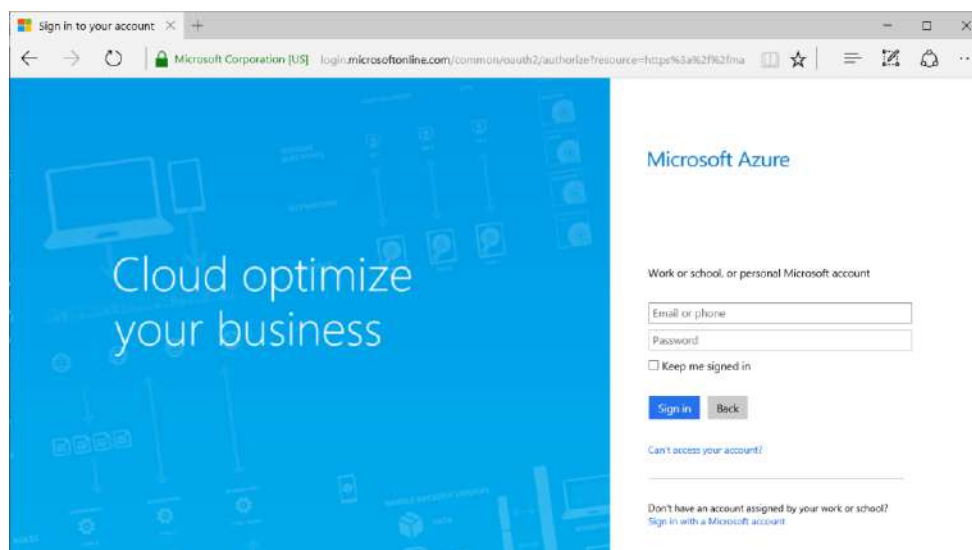

Figure 2-a: Microsoft Azure Sign-in Screen

Once you've signed up or signed in to the Azure Portal, you can browse through the list of Azure services and select the **Azure Cosmos DB** option.
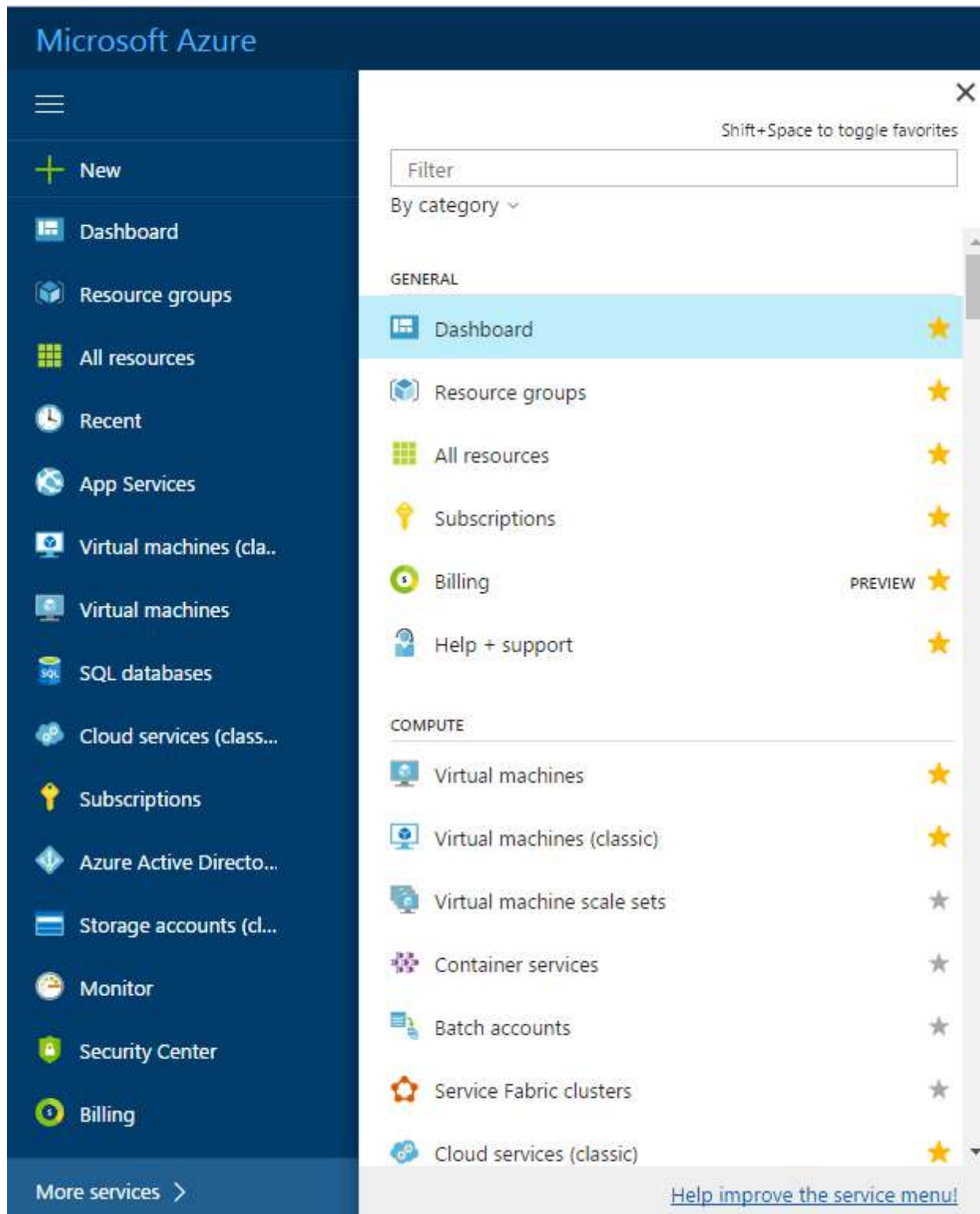
Figure 2-b: Browsing for Cosmos DB within the List of Azure Services

After selecting the **Azure Cosmos DB** option, you have to create a new Cosmos DB account by clicking **Add**.
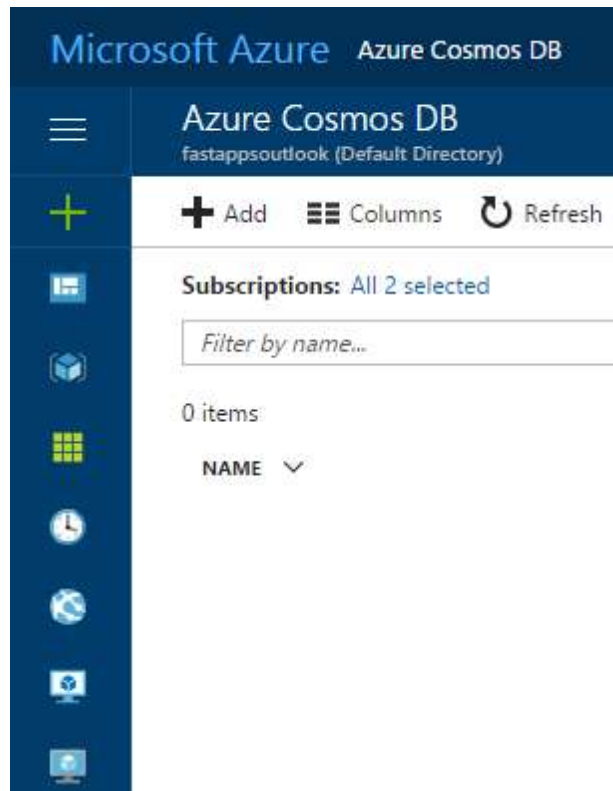
Figure 2-c: Screen to Add a Cosmos DB Account

This leads to a screen that allows you to enter the details of the Cosmos DB account: **ID**, **API**, **Subscription**, **Resource Group**, and **Location**. You can select which Azure region your account will be hosted on.

The ID is a unique global identifier within Microsoft Azure for the Cosmos DB account. Make sure to select **SQL (DocumentDB)** as the API. To finalize the creation of the account, click **Create**.
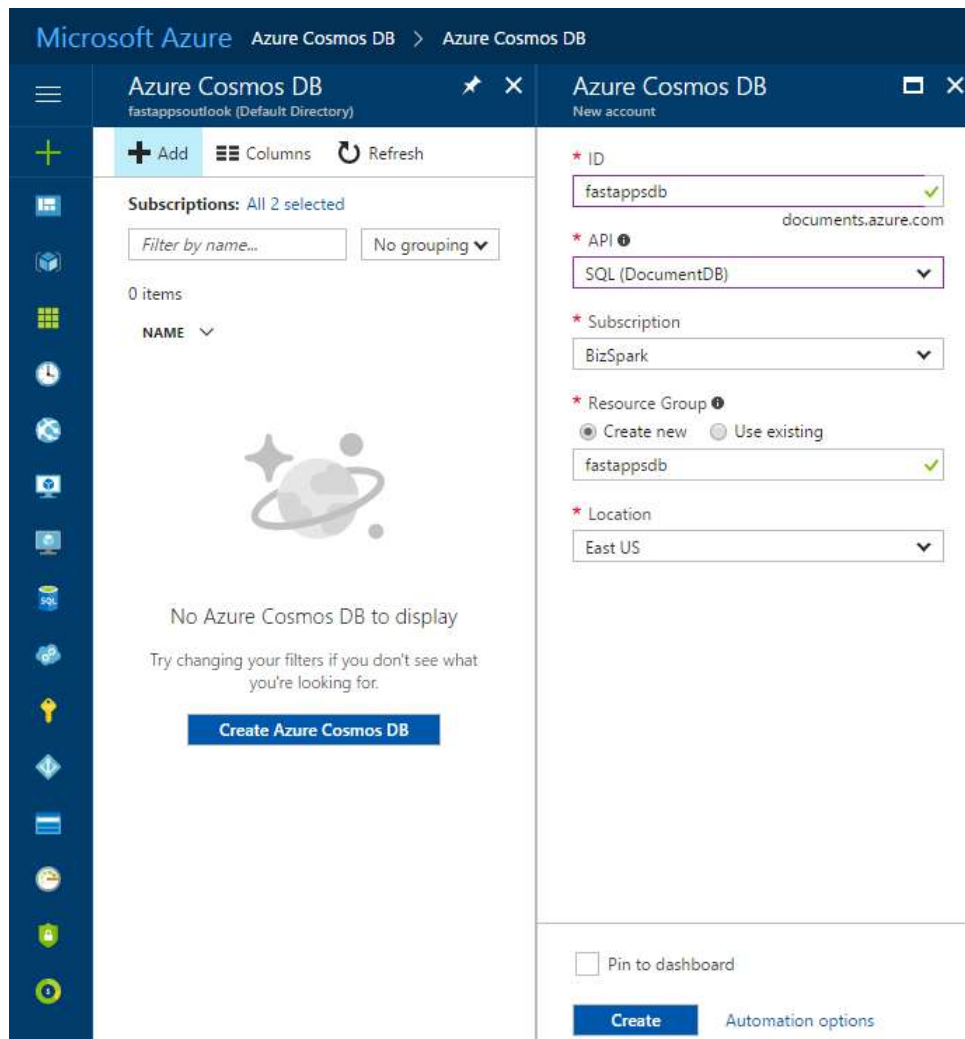
Figure 2-d: Final Cosmos DB Account Creation Screen

Once the Cosmos DB account has been created, it can be seen as follows.
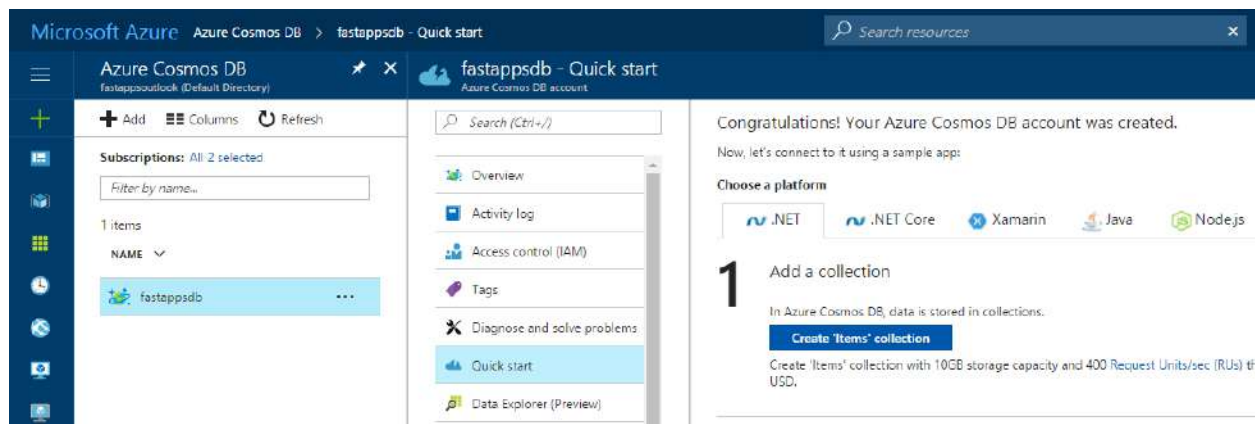
Figure 2-e: Cosmos DB Account Dashboard

In Cosmos DB, data is stored in collections. We can choose our platform and then click the **Create 'Items' collection** button to get started. In my case, I'll be using .NET code later on so this is the platform I'll choose.

Once the **Items** collection has been created, you'll be presented with the following screen.
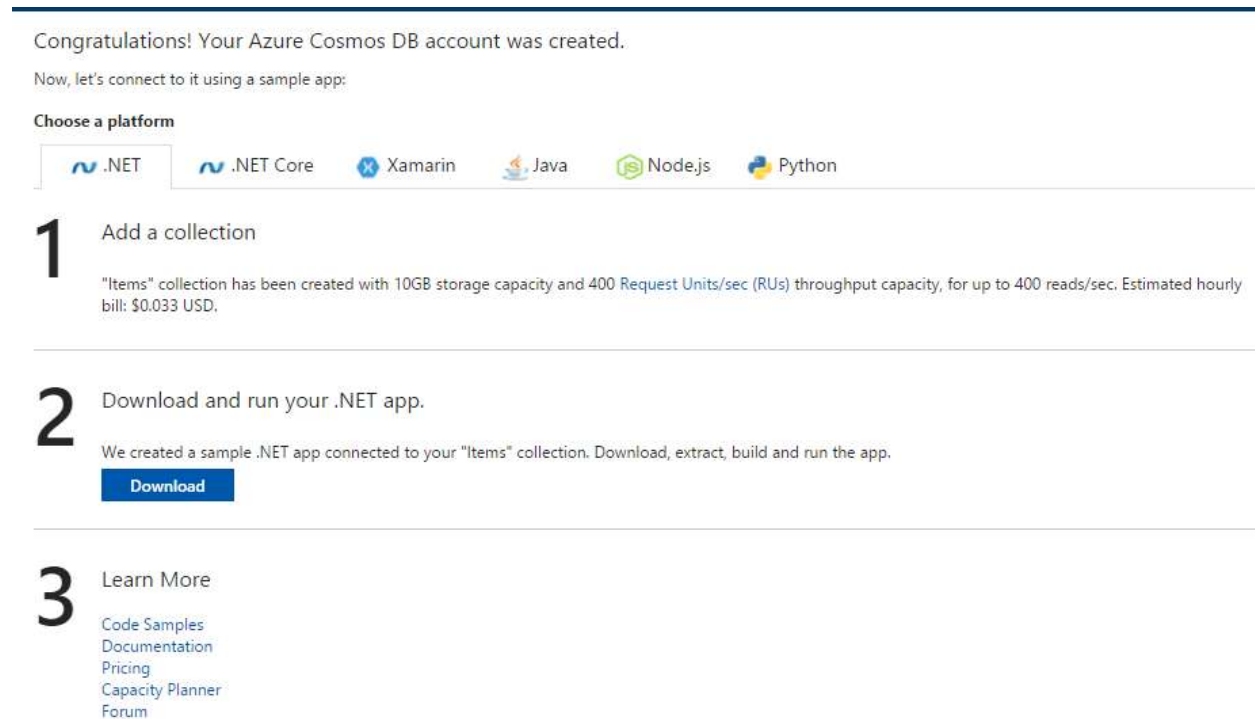
Figure 2-f: Items Collection Created Screen

You can choose to download the sample application provided, but I'll skip this for now. Let's instead click on the **Data Explorer (Preview)** option to view our Items collection.

Figure 2-g: The Cosmos DB Data Explorer (Preview) Pane

We'll be spending a bit of time in the **Data Explorer (Preview)** blade, so it will be our main starting point.

One of the downsides to using Azure services is that the UI is in constant flux. Therefore, by the time you read this, the UI will likely be slightly different from the figures in this e-book. However, changes will be minor and you'll have no trouble matching the figures to what you see.

But before we start adding documents and interacting with DocumentDB, let's review quickly how it is internally structured.
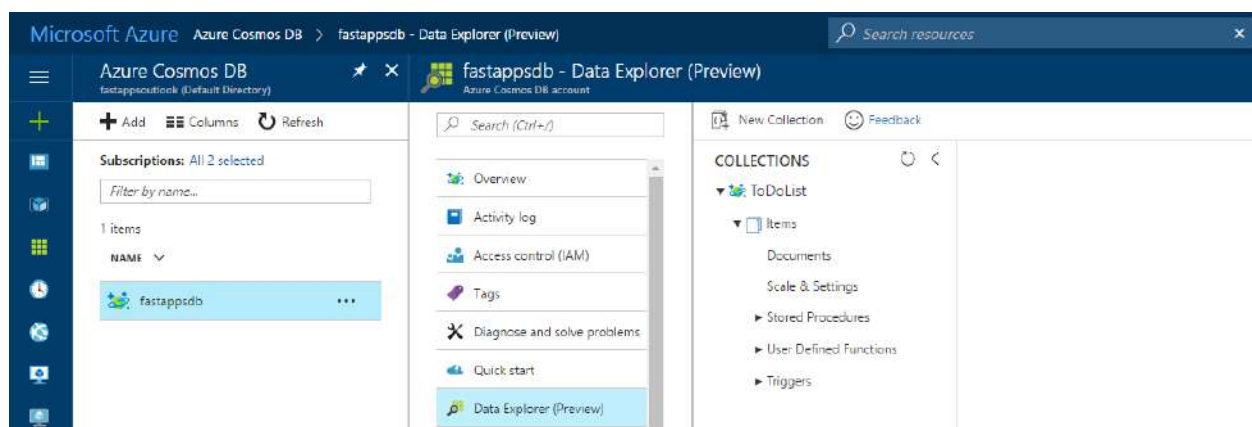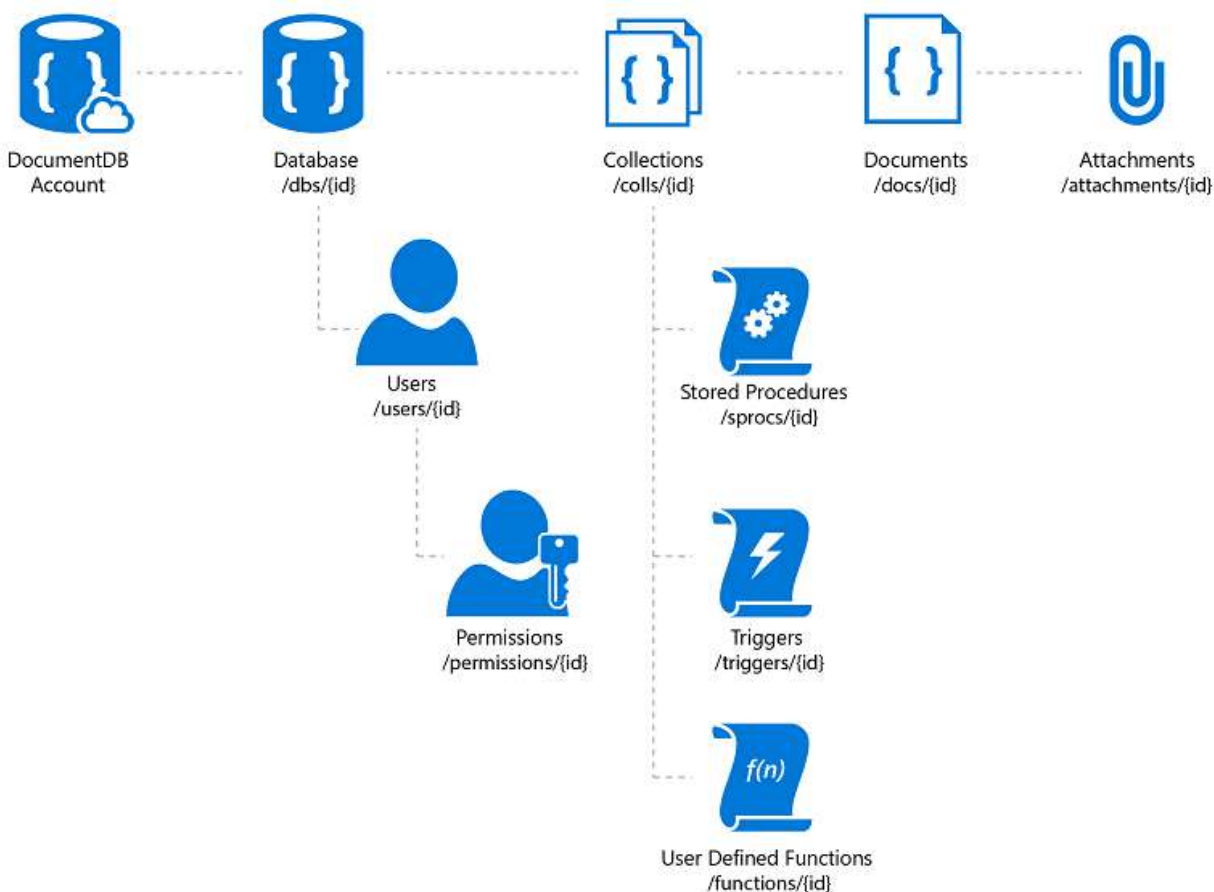

Figure 2-h: DocumentDB Internal Structure (Image Courtesy of Microsoft Azure)

To understand this diagram better, note that a Cosmos DB account consists of a set of databases, each containing multiple collections, each of which can contain stored procedures, triggers, UDFs, documents, and related attachments.

A database also has associated users, each with a set of permissions to access various other collections, stored procedures, triggers, UDFs, documents, or attachments.

While databases, users, permissions, and collections are system-defined resources with well-known schemas, on the other hand stored procedures, triggers, UDFs, and attachments contain arbitrary and user-defined JSON content.

In my case, **fastappsdb** corresponds to the Cosmos DB account, **ToDoList** corresponds to the automatically created DocumentDB database that was created when I previously clicked the **Create 'Items' collection** button, and **Items** is the collection that was created.
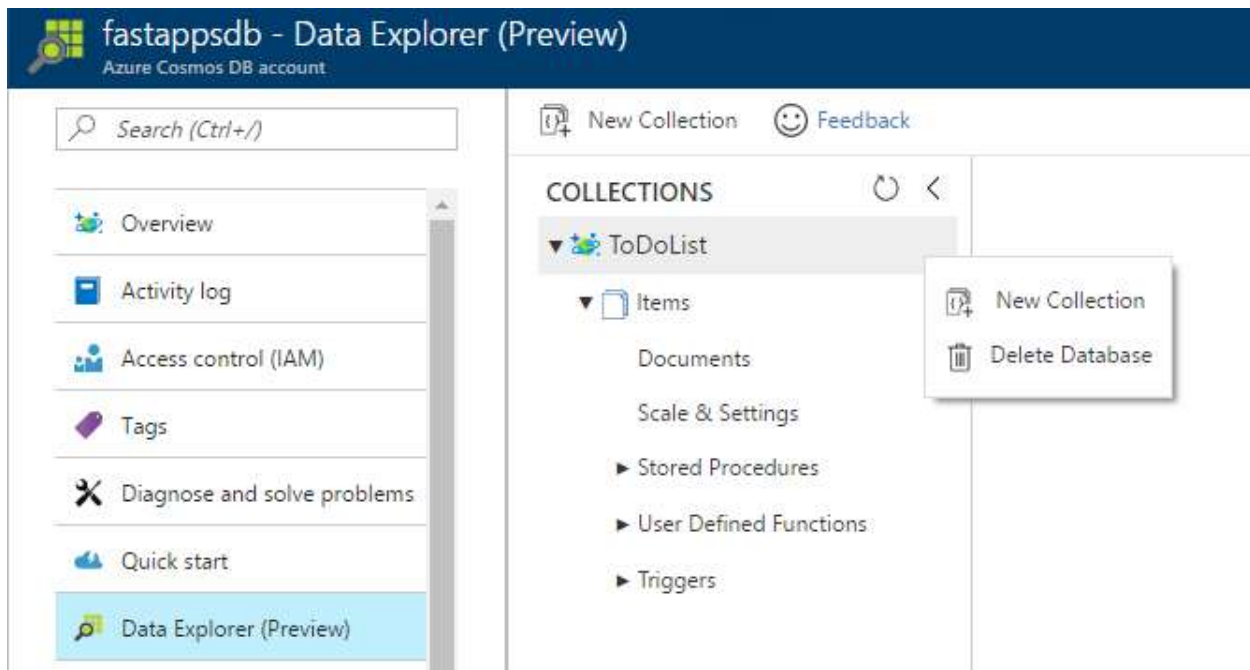
Figure 2-i: Our DocumentDB Database Containing the Items Collection

Notice that if you want you can choose to create a new collection on the existing DocumentDB database or delete it and later create a new one. For now, let's leave things are they are.

Creating a new collection

Although we don't need a new collection now, it is important to know how you can create one.

This can be done by simply clicking the **New Collection** button at the top of the **Data Explorer (Preview)** blade.

Alternatively, this can also be done at the database level by clicking the **ellipsis** (**…**) button, which will make a small menu appear, then clicking the **New Collection** option.

If you do this, a pane on the right side of the screen will appear where you can enter the name of your new collection.
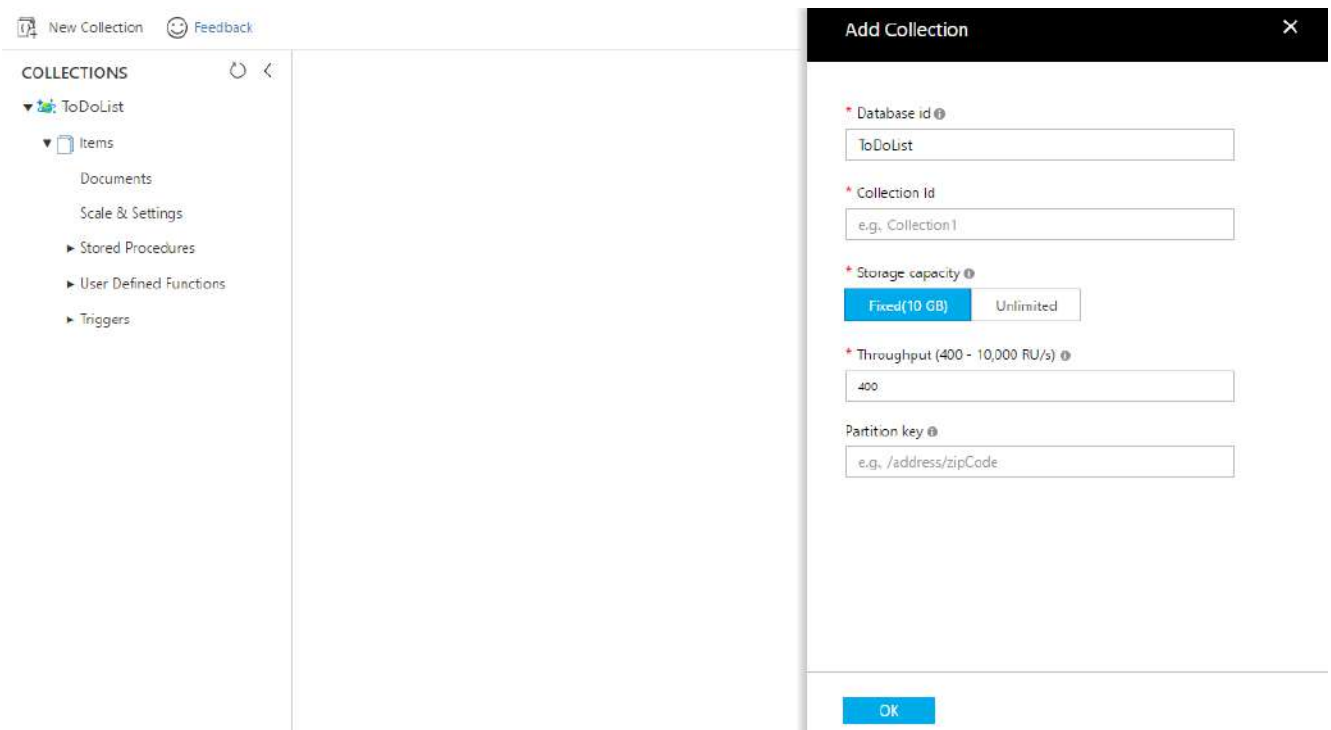
Figure 2-j: Add Collection Pane

Once there, specify a unique name that will be used for the **Collection Id** and then change the **Throughput** field if desired. You may also add a **Partition key**. You are now ready to create your collection. Let's explore how we can add documents to an existing collection.

Adding a document

Within the **Data Explorer (Preview)** blade, expand the **Items** collection and then select the **Documents** option.
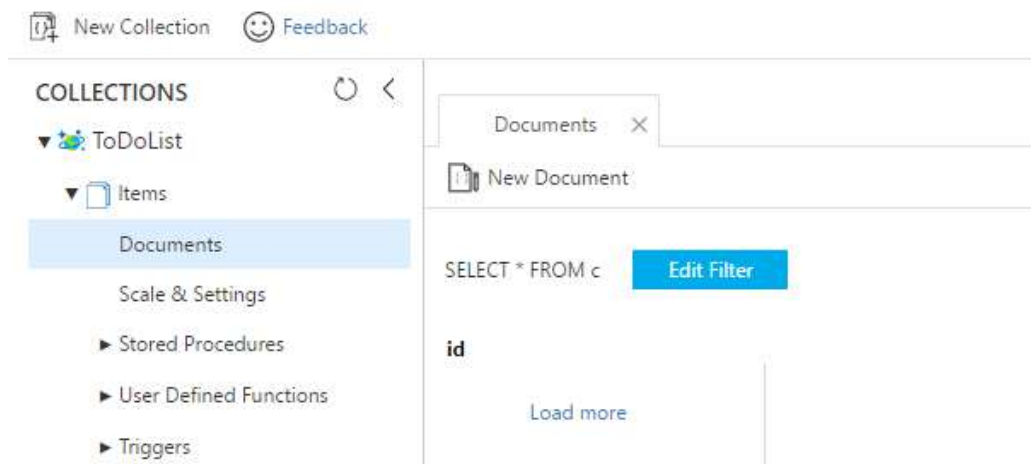
Figure 2-k: The Collection's Document Explorer

Click the **New Document** button and the **Document Explorer** window will then change to edit mode and a template JSON document will be displayed. Remove the default JSON data and enter the following information.
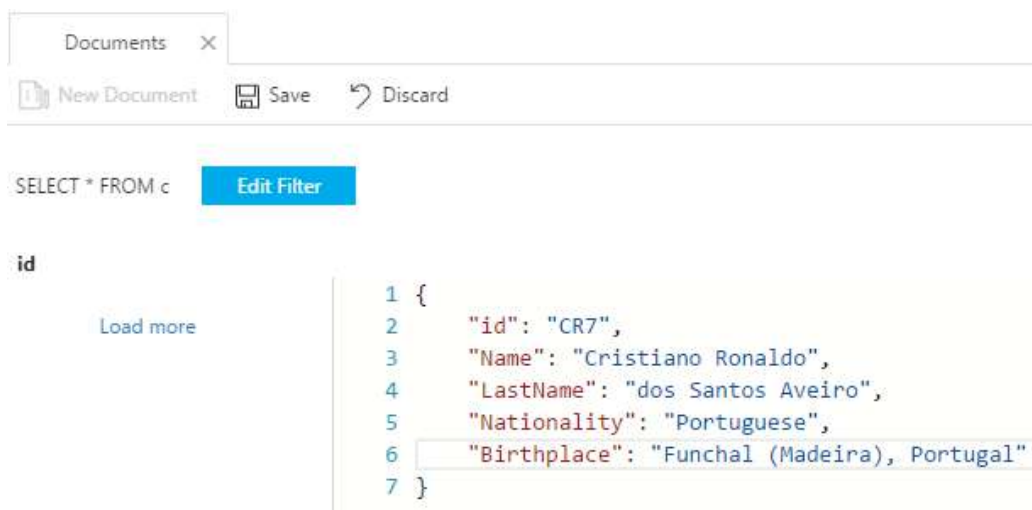
Figure 2-I: Creating a New Document with Document Explorer

This is a simple web-based text editor that enforces no rules and allows you to add pretty much any raw data to it.

We are creating a JSON document with basically any properties we wish. Once done, all we have to do is click **Save** and our document will be stored inside the collection.

Notice that a unique **id** has been provided and this must be unique across all documents within a collection. If you don't provide a unique **id**, DocumentDB will automatically create one using a GUID.

The unique **id** property must be always a **String** no longer than 255 characters. It cannot be a **Number**, **Date**, **Time**, **Boolean**, or another **object**.

That's all there is to creating JSON documents within the Azure Portal for a DocumentDB collection. Pretty easy.

Data migration into DocumentDB

A great way to get data into DocumentDB is to import it from a traditional relational data source such as SQL Server. However, that's not the only way.

The open source project DocumentDB Data Migration Tool allows you to precisely achieve that. The binaries can be downloaded from here.

Furthermore, this tool allows you to import data not only from SQL Server, but also from various other sources such as Azure Table storage, JSON files, MongoDB, CSV files, RavenDB, Amazon DynamoDB, HBase, and even from other DocumentDB databases. It's pretty handy.

Let's go ahead and download this tool and then run it. The binaries are packaged as a zip file. Once downloaded, simply unzip to any folder.
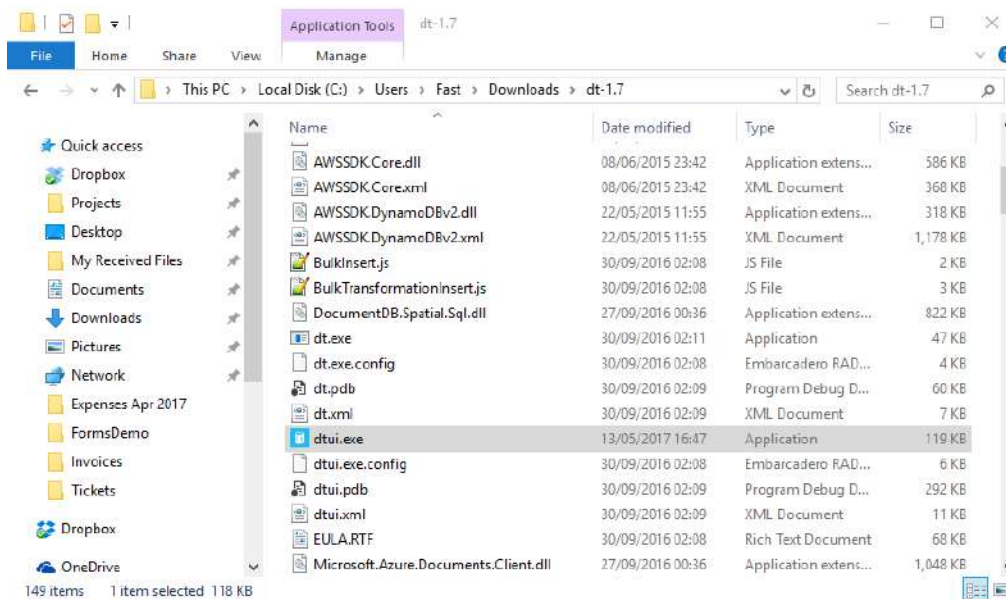
Figure 2-m: The DocumentDB Data Migration Tool Binaries

Once unzipped, locate the file called **dtui.exe** and run it. You'll then be presented with the following screen.
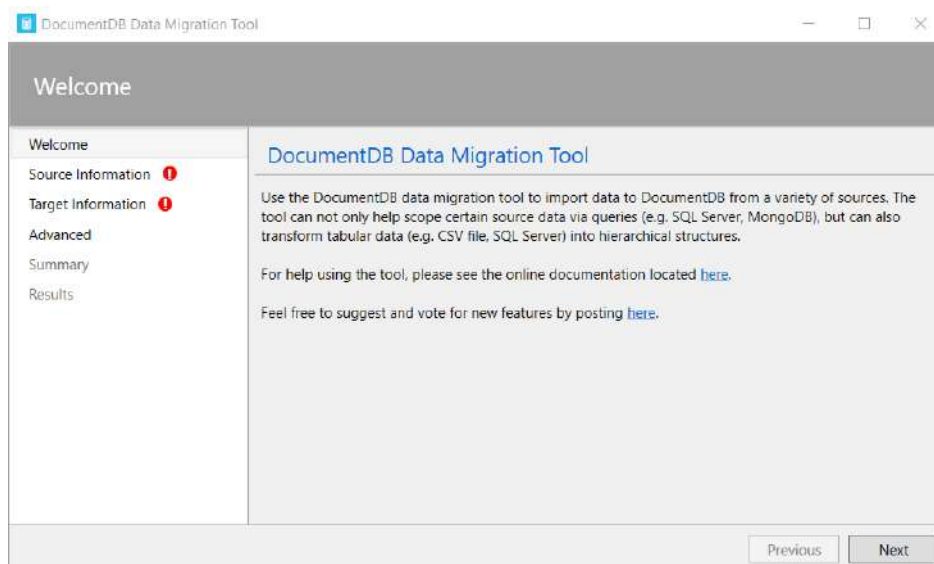
Figure 2-n: The DocumentDB Data Migration Tool Welcome Screen

The DocumentDB Data Migration Tool is quite intuitive and easy to use. Click **Next** and you'll be asked to select the source from which you want to import. Then indicate the connection parameters required in order to retrieve the data.

In this example we'll be importing data from an Azure Table.

Figure 2-o: Selecting an Import Source in the DocumentDB Data Migration Tool

Fill in the **Connection String** to the Azure Storage Table (which you can copy from the Azure management dashboard), the Table Name of the Azure table, and then, if you do not wish to fill in any of the other fields, click **Next**.

Figure 2-p: Target Information for the DocumentDB Data Migration Tool

Once this has been done, click **Next** to proceed. You'll review the overall migration information and finalize the data migration process.

Given that there are multiple possibilities of importing data from various data sources, it is important to double check that the source and target information is correct so that the Data Migration Tool can correctly perform its job.

The Data Migration Tool is a handy way to get data into DocumentDB and get started. It is also a great way to get data from a standard SQL Server database system into DocumentDB. It's a very easy to use tool that requires very little experimentation to get right. Use it to your advantage.

Summary

In this short chapter, we've gone through some of the basic steps required to get DocumentDB up and running and import data.

We've seen how to create a DocumentDB database and a collection, and add data to it. We've also looked at the different pricing tiers associated with performance levels when creating a collection.

In the next chapter we'll look at using DocumentDB for running data queries using a very familiar SQL flavored syntax. Following that, we'll explore building client applications and program directly on the DocumentDB server.

The real fun is about to begin. Thanks for reading!

Chapter 3 Queries with DocumentDB

Introduction

In the previous chapters, I've explained the benefits of using Cosmos DB with the DocumentDB API and we've learned about some of its most prominent features.

We described how DocumentDB is designed from the ground up to scale out and work with schema-free hierarchical JSON documents instead of traditional database tables.

We also explored how to get up and running with Cosmos DB (DocumentDB API) using the Azure Portal by creating a database and a collection. We also learned how to import data using the Data Migration Tool.

With this wrapped up, we are in a position to fully start using DocumentDB, putting some data in it and executing queries.

In this chapter we'll explore how to query DocumentDB using a very familiar SQL-like grammar and syntax for retrieving documents, with roots in JSON and JavaScript semantics.

The Microsoft Azure Cosmos DB team has made a great decision to use a SQL flavored syntax for querying documents within DocumentDB, as it is expressive, easy-to-use, and standard (most developers know it), rather than coming up with a new language or way of querying documents.

What is important to understand is that SQL is actually a relational database querying language. However, Microsoft has adapted it to the NoSQL world, so it is capable of querying JSON documents and establishing relations among them, just like it does with a relational database.

The language still reads as traditional SQL, but the semantics are all based on JSON and it works with JavaScript, but not T-SQL datatypes. Also, expressions are evaluated as JavaScript expressions rather than T-SQL ones.

It is also important to understand that the results returned by running DocumentDB queries using the SQL flavored syntax will be schema-free JSON objects, which could have nested properties or not. We will not be dealing with results organized into rows and columns using a tabular format. Nested properties are accessed using dotted notation, which allows you to descend into any subsection of a document.

The beauty of the SQL flavored syntax in Cosmos DB using the DocumentDB API is that it also allows you to iterate through nested arrays among documents and project the results as a custom JSON document to fit your needs or simply return the results as-is.

The usage of common operators is also allowed, such as arithmetic operators (**+**, **-**, *****, **/**, **%**), bitwise operators (**|**, **&**, **^**, **>>**, **>>>**), logical operators (**AND**, **OR**), comparison operators (**=**, **!=**, **>**, **>=**, **<**, **<=**, **<>**), and the string operator for concatenation (**||**).

Furthermore, the usage of clauses such as **SELECT**, **FROM**, **WHERE**, **JOIN**, **IN**, **BETWEEN**, and **ORDER BY** are permitted.

There are many built-in functions for:

- Math: **ABS**, **CEILING**, **EXP**, **FLOOR**, **LOG**, **LOG10**, **POWER**, **ROUND**, **SIGN**, **SQRT**, **SQUARE**, **TRUNC**, **ACOS**, **ASIN**, **ATAN**, **ATN2**, **COS**, **COT**, **DEGREES**, **PI**, **RADIANS**, **SIN**, and **TAN**.

- Type checking: **IS_ARRAY**, **IS_BOOL**, **IS_NULL**, **IS_NUMBER**, **IS_OBJECT**, **IS_STRING**, **IS_DEFINED**, and **IS_PRIMITIVE**.

- String: **CONCAT**, **CONTAINS**, **ENDSWITH**, **INDEX_OF**, **LEFT**, **LENGTH**, **LOWER**, **LTRIM**, **REPLACE**, **REPLICATE**, **REVERSE**, **RIGHT**, **RTRIM**, **STARTSWITH**, **SUBSTRING**, and **UPPER**.

- Array manipulation: **ARRAY_CONCAT**, **ARRAY_CONTAINS**, **ARRAY_LENGTH**, and **ARRAY_SLICE**.

Although we won't explore each and every one of these built-in functions, it's useful to know that the DocumentDB SQL flavored syntax supports all these options, which makes it convenient for handling documents with various data types. The functions will also be useful when projecting custom JSON results.

By the end of this chapter you should have a good understanding of how to write and execute queries using DocumentDB to retrieve documents and project custom results. The examples should be both intuitive and easy to follow and implement.

I encourage you to explore the online [documentation](), which contains a wealth of information about DocumentDB's query language.

Now, let's get started and explore how we can query DocumentDB using its very own SQL flavored syntax. Let the fun begin!

Our first query

The Azure Portal has a Query Explorer that allows us to run queries against DocumentDB collections.

We'll be actively working with the Query Explorer to showcase the various possibilities of querying DocumentDB.

To get started, within the **Data Explorer (Preview)** blade, select the **Items** collection and click the **ellipsis** (**…**) button. A small menu will appear with a few options. Then click the **New SQL Query** option. This will open a query tab as shown in the following figure.
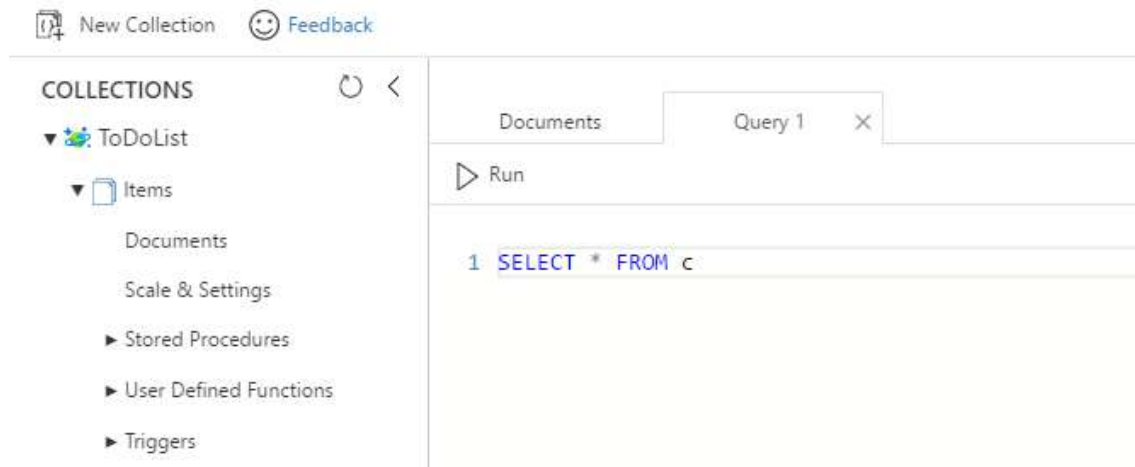
Figure 3-a: A Query Explorer Tab

As soon as the Query Explorer tab is opened, a default query is presented. This query is a simple **SELECT** * from the active collection.

Code Listing 3-a: The Default DocumentDB Query

```
-- DocumentDB default SQL query.

SELECT * FROM c

-- c is the alias for the Active Collection
```

To execute the query, you must click the **Run** button at the top of the **Query** tab. Notice that in the Query tab, queries are not terminated using a semicolon (**;**).

Based on the collection we previously created containing information about Cristiano Ronaldo, this is what the result looks like after running the query.
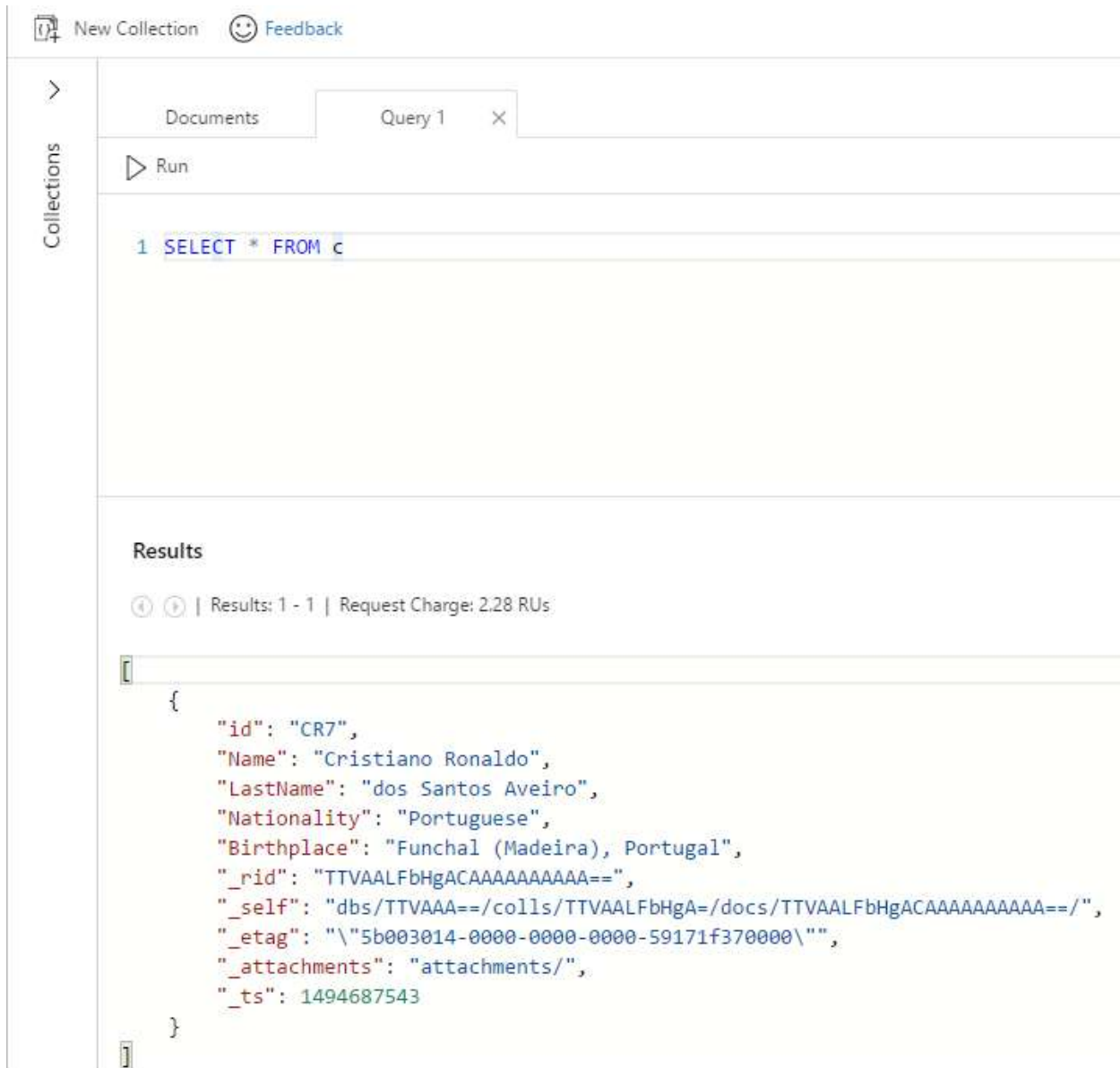
Figure 3-b: The Query Result

Notice that the resulting document contains additional properties that were not originally visible when the document was added to the collection.

These properties are the document's resource ID (**_rid**); time stamp (**_ts**), which represents a raw integer value for when the document was last inserted or updated; self-link (**_self**), which drills down on **_rid** from the database to the collection and to this specific document; **_etag**, used for concurrency; and the **_attachments** link (in case there are any).

Although this is a very simple query, there are some important differences from the world of relational databases. In a relational database, ***** means all the columns. In DocumentDB, *****
means return the document results as they are stored.

Unlike with relational databases, using `*` doesn't have any negative impact on performance. If you specifically indicate the name of the properties you want to include in your results, then you are projecting a new shape for each document that will be returned within those results, therefore providing a consistent schema for those returned documents. In DocumentDB, this is known as projection.

The `FROM` clause is used to indicate the collection the query will be executed against. It is not mandatory to use it. However, if it is not used, then you will not retrieve documents stored within a collection, but instead be executing queries with expressions that return scalar values. We'll explore that shortly.

At the top of the **Results** area, DocumentDB provides some valuable information about the RUs involved when executing this query.

Results

◁ ▷ | Results: 1 - 1 | Request Charge: 2.28 RUs

Figure 3-c: Query Execution RU Details

This wraps up our first query. Now that we know how to create a new query, let's move on to more advanced examples and interesting things we can do. Scalar queries are up next.

Basic scalar queries

Scalar queries are queries that simply evaluate expressions and do not return any document results from a collection. These queries are a great way to learn the basics of DocumentDB querying and explore what is possible. They are also very useful when building custom projections.

Let's have a look at some basic scalar expression queries.

Code Listing 3-b: Basic Scalar Queries

```
-- DocumentDB 3 basic scalar queries

SELECT "Welcome"
SELECT "DocumentDB" AS Word
SELECT VALUE "Test"
```

Let's now execute each one of these queries separately with our opened **Query** tab.

If you copy and paste these three queries directly into the **Query** tab, you won't be able to execute them all at the same time.

In order to execute these queries one at a time, copy, paste, and then run each one separately.

Let's start off by running the first one. This query returns the following result.

Results

① ⑨ | Results: 1 - 1 | Request Charge: 2.15 RUs

```
[
    {
        "$1": "Welcome"
    }
]
```

Figure 3-d: Scalar Query 1 Execution Result

Notice how DocumentDB created a JSON document using the expression passed as its result. However, look at what happens when we execute the second query.

Results

① ⑨ | Results: 1 - 1 | Request Charge: 2.16 RUs

```
[
    {
        "Word": "DocumentDB"
    }
]
```

Figure 3-e: Scalar Query 2 Execution Result

In this case, the name of the property is **Word** instead of **$1**. This is because in the second query we used the **AS** clause to indicate how we want to represent the result.

In the first query, because the **AS** clause was not used, DocumentDB automatically assigned the name **$1** to the resultant property of the query.

When we execute the third query, we get the following.

Results

① ⑨ | Results: 1 - 1 | Request Charge: 2.15 RUs

```
[
    "Test"
]
```

Figure 3-f: Scalar Query 3 Execution Result

In this query, because we are using the keyword **VALUE**, we are telling DocumentDB to simply return the value of the expression without a property name.

Now let's consider these other examples that show a very cool feature of DocumentDB's SQL flavored querying syntax: the ability to include JSON expressions.

Code Listing 3-c: Scalar Queries with JSON Syntax

```
-- DocumentDB scalar queries. The second one has JSON syntax embedded.

SELECT "This is", "DocumentDB" AS Word
SELECT ["This is", "DocumentDB"] AS Words
```

If we copy and paste the first query and click **Run**, we'll get the following result.

Figure 3-g: Another Scalar Query Execution Result

As you can see, the result is pretty much what we were expecting, given that the second string "DocumentDB" was projected into the property **Word** using the **AS** clause, whereas for the first string "This is", DocumentDB created an automatic property called **$1**.

Now, copy, paste, and execute the second query. Running the query provides the following result.

Figure 3-h: JSON Syntax Scalar Query Execution Result

Notice in this query that DocumentDB is returning the result containing the two words as an array, just like it was represented as an expression using JSON syntax.

Being able to include JSON syntax in the SQL expression itself is a very handy and powerful feature, which can be quite valuable when creating custom projected results.

Let's explore other types of scalar expression queries we could use and run with DocumentDB.

Advanced scalar queries with JSON syntax

So far we've seen some simple and interesting scalar queries we can write using DocumentDB, but we've just scratched the surface of what kind of scalar queries are possible. Let's have a look at the following examples.

Code Listing 3-d: More Scalar Queries with JSON Syntax

```
-- DocumentDB scalar queries with JSON syntax.
SELECT {"Sentence 1": "DocumentDB", "Sentence 2": "is cool"} AS Sentences
SELECT {"Sentence 1":["Document", "DB"], "Sentence 2":["is", "cool"]} AS
Sentences
```

If we copy, paste, and then run the first example on the **Query** tab, we get the following result.

Figure 3-i: JSON Syntax Scalar Query Execution Result

If we copy, paste, and run the second example, we get the following result instead.

Figure 3-j: JSON Syntax Scalar Query Execution Result

Notice that main difference between these scalar queries is that in the second example **Sentence 1** and **Sentence 2** are actually array properties, whereas in the first example they are simply string properties.

You can let your imagination go wild. Basically, any kind of projection can be shaped. That's the expressiveness and beauty of what can be achieved with JSON syntax inside a scalar query.

Scalar queries using operators

Scalar queries are flexible and very versatile. One of their great features is that they can be used with operators. Let's have a look at what is possible.

Code Listing 3-e: Scalar Query Using Operators

```
-- DocumentDB scalar queries using operators.
SELECT 12 + ((7 * 8) / 7) AS Res, ("1" = "2" AND "Blue" = "Red" OR 10 =
10) AS Logic, (8 > 7 ? "8" : "7") AS Ternary, ("me" ?? undefined) AS
Coalesce, ("This" || " is " || "DocumentDB") AS ConcatStr
```

This scalar query includes a few interesting things. It includes both mathematical and logical operations. It also includes a ternary and coalesce expression, as well as string concatenation.

When running this example, the following result is projected.

Results

◉ ⊙ | Results: 1 - 1 | Request Charge: 2.16 RUs

```
[
    {
        "Res": 20,
        "Logic": true,
        "Ternary": "8",
        "Coalesce": "me",
        "ConcatStr": "This is DocumentDB"
    }
]
```

Figure 3-k: Result of the Scalar Query Using Operators

We can combine all sorts of operations together as long as they make sense in order to project custom fields. The ternary and coalesce operators can be used to build conditional expressions similar to other popular programming languages.

The ternary (?) operator can be very handy when constructing new JSON properties on the fly. On the other hand, the coalesce (??) operator can be used to efficiently check for the presence of a property defined in a document. This is useful when querying against semi-structured data or data of mixed types.

We can use parentheses to determine the correct order in which we want expressions to be evaluated, just like we would do with JavaScript expressions. The same expression evaluation rules as in JavaScript apply here as well.

Furthermore, coalesce expressions can be chained to one another, and even string concatenations are possible. The possibilities are endless, limited only by the operators available and your imagination.

More information about DocumentDB query operators can be found here. Up next are scalar queries using built-in functions. Let's have a look.

Scalar queries using built-in functions

Beyond using operators, we can also write scalar queries using many of the useful built-in functions provided by DocumentDB. Let's explore some examples.

Code Listing 3-f: Scalar Query Using Built-In Math Functions

```
-- DocumentDB scalar queries using built-in math functions.
SELECT PI() AS Pi, LOG(15) AS Log, TAN(85) AS Tan, COS(52) AS Cos, SIN
(61) AS Sin, FLOOR(13.5) AS Floor, ABS(-25) AS Abs, CEILING (22.2) AS
Ceiling, ROUND(6.7) AS Round
```

If we copy and paste this text and execute this query, we get the following result.
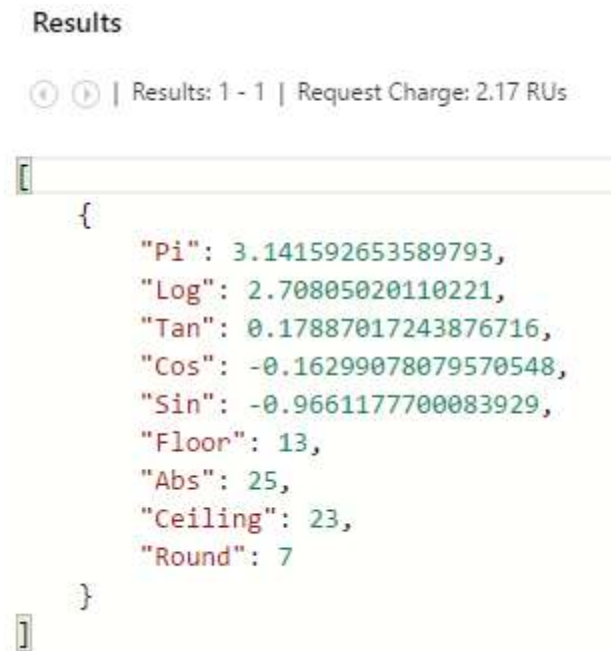
Figure 3-l: Result of the Scalar Query Using Built-In Functions

DocumentDB's built-in math functions work just as great and in the same way as math functions from other platforms and programming languages, using the same intuitive and well-known names. Details about each math function can be found here.

Let's now look at running a query using built-in functions to perform type checking on specific data types. Type checking functions are indispensable when you have one or more properties with varying data types.

Code Listing 3-g: Scalar Query Using Built-In Type Checking Functions

```
-- DocumentDB scalar queries using built-in type checking functions.
SELECT IS_OBJECT("DocDB") AS Obj1, IS_OBJECT ({"Word": "DocDB"}) AS Obj2,
IS_NULL (null) AS Null1, IS_NULL (52) AS Null2, IS_BOOL (61) AS Bool1,
IS_ARRAY ([13.5]) AS Array1
```

If we execute this query, we get this result.

Results

| Results: 1 - 1 | Request Charge: 2.16 RUs

```
[
    {
        "Obj1": false,
        "Obj2": true,
        "Null1": true,
        "Null2": false,
        "Bool1": false,
        "Array1": true
    }
]
```

Figure 3-m: Result of the Scalar Query Using Type Checking Functions

Type checking functions are really useful because you may know that a property exists in some documents within a collection, but you may not know what data type that property actually is. Even properties with the same name throughout different documents might have different data types. Type checking is a very useful way of ensuring you use the right data types for the results you need to project.

Now that we've quickly explored type checking functions, let's move our attention to built-in string manipulation functions. These include features such as string extraction, parsing, case conversion, concatenation, and search. Let's explore an example.

Code Listing 3-h: Scalar Query Using Built-In String Manipulation Functions

```
-- DocumentDB scalar queries using string manipulation functions.
SELECT UPPER ("upper") AS Upper1, LOWER ("LOWER") AS Lower1, LENGTH
("length") AS Length1, SUBSTRING ("abcdef", 2, 2) AS Substring1, LEFT
("left", 2) AS Left1, RIGHT ("right", 2) AS Right1, INDEX_OF ("index",
"ex") AS Index1, ENDSWITH ("abcdef", "cdef") AS Endswith1, STARTSWITH
("abcdef", "ab") AS Startswith1, CONCAT ("this", " is ", "cool") AS
Concat1, CONTAINS ("this", "i") AS Contains1
```

String manipulation functions are self-descriptive and have been designed to work in DocumentDB in the same way they have been designed to work in other programming languages.

If we execute this query, we get this result.

Results

Results: 1 - 1 | Request Charge: 2.17 RUs

```
{
    "Upper1": "UPPER",
    "Lower1": "lower",
    "Length1": 6,
    "Substring1": "cd",
    "Left1": "le",
    "Right1": "ht",
    "Index1": 3,
    "Endswith1": true,
    "Startswith1": true,
    "Concat1": "this is cool",
    "Contains1": true
}
```

Figure 3-n: Result of the Scalar Query Using String Manipulation Functions

The results are also very much self-explanatory and easy to understand just by looking at them. String manipulation functions provide a great way to project custom query results.

More details about each individual string manipulation function can be found here. Have a look to understand what each one does.

Now that we've explored string manipulation, let's move our attention to arrays and how we can use some of the available built-in functions to bend and twist them.

When working with arrays, you can concatenate multiple arrays into one using **ARRAY_CONCAT**, check if a particular array exists within another using **ARRAY_CONTAINS**, check the length of an array with **ARRAY_LENGTH**, and extract a portion of an existing array using **ARRAY_SLICE**.

Let's explore a scalar query example using these built-in array functions.

Code Listing 3-i: Scalar Query Using Built-In Array Manipulation Functions

```
-- DocumentDB scalar queries using array manipulation functions.
SELECT ARRAY_SLICE (["1", "2", "3", "4"], 1, 2) AS Slice1, ARRAY_LENGTH
(["1", "2", "3", "4"]) AS Len1, ARRAY_CONTAINS (["1", "2", "3", "4"],
"5") AS Contains1, ARRAY_CONCAT (["1"], ["2"], ["3"], ["4"])
```

If we execute this query, we get the following result.

Results

(<) (>) | Results: 1 - 1 | Request Charge: 2.16 RUs

```
[
    {
        "Slice1": [
            "2",
            "3"
        ],
        "Len1": 4,
        "Contains1": false,
        "$1": [
            "1",
            "2",
            "3",
            "4"
        ]
    }
]
```

Figure 3-o: Result of the Scalar Query Using Array Manipulation Functions

Notice that **Slice1** contains the values of the second and third values of the original array since we instructed the slice to occur on the item with index **1** (second item of the array because arrays use zero-based indexing) and we also instructed **ARRAY_SLICE** to fetch two items.

Also notice how **Contains1** returns false, as the array doesn't contain the value "5" and the result of **ARRAY_CONCAT** is assigned to an automatically generated property called **$1**, since no **AS** clause was provided for the last part of the scalar query.

Further documentation about these functions can be found [here](#).

We've now seen what we can achieve with scalar queries. They provide a great way to manipulate results and create custom projections. Moving on, we'll focus on querying real data that we'll store in our collection.

Querying documents in a collection

We can really appreciate the value of DocumentDB's SQL flavored query language when we have to use it to get data and project custom results from documents contained within a collection.

Once you include a **FROM** clause on a query, it is no longer a scalar query but instead a query that runs through all the documents within the current collection.

The name of the variable that goes after the **FROM** clause is merely an alias for the name of the current collection. Usually that alias is **c**, indicating the current collection, but it can be any other name.

Using the **Data Explorer (Preview)** blade, let's add the following document to our current collection, which is available on Cosmos DB's demo website. For a quick reminder on how to add a document, refer to the section "Adding a Document" from Chapter 2.

Code Listing 3-j: Sample Document

```
{
  "id": "19015",
  "description": "Snacks, granola bars, hard, plain",
  "tags": [
    {
      "name": "snacks"
    },
    {
      "name": "granola bars"
    },
    {
      "name": "hard"
    },
    {
      "name": "plain"
    }
  ],
  "version": 1,
  "isFromSurvey": false,
  "foodGroup": "Snacks",
  "servings": [
    {
      "amount": 1,
      "description": "bar",
      "weightInGrams": 21
    },
    {
      "amount": 1,
      "description": "bar (1 oz)",
      "weightInGrams": 28
    },
    {
      "amount": 1,
      "description": "bar",
      "weightInGrams": 25
    }
  ]
}
```

Now let's run this next query.

```
SELECT food.id,
       food.description,
       food.tags,
       food.foodGroup
FROM food
WHERE food.foodGroup = "Snacks" AND food.id = "19015"
```

After executing this query, we get the following result.

Code Listing 3-l: Custom Projection Query Results

```
[
  {
    "id": "19015",
    "description": "Snacks, granola bars, hard, plain",
    "tags": [
      {
        "name": "snacks"
      },
      {
        "name": "granola bars"
      },
      {
        "name": "hard"
      },
      {
        "name": "plain"
      }
    ],
    "foodGroup": "Snacks"
  }
]
```

There are a couple of things to notice. The first is that we were able to add a totally unrelated document to the one we had previously added, the one with data about Cristiano Ronaldo.

The document that we've just added has a structure, names, and values of properties that are totally unrelated to the original document we added to the current collection. This means we can add any type of document to a collection, with any kind of properties and values.

The second thing to notice is that in the query mentioned in Code Listing 3-k, we are using the alias **Food** to refer to the current collection, even though the collection might contain other documents that are unrelated to **Food**, such as the original one that contains data about Cristiano Ronaldo.

When we write a document query, it is recommended that we use an alias that represents the type of documents we are querying within the current collection. In this case, because we are querying about food, we have chosen to use the alias **Food**.

We could have chosen to use a generic alias like **c**, but by using the alias **Food**, we are simply making the query more readable and easier for us to understand in the future.

Also notice how the dotted notation is used to indicate the names of the properties that will be returned in the result and used for filtering (used inside the **WHERE** clause).

Using the dotted notation is required for a document query to work.

Querying using ORDER BY

Let's move on and walk through the example queries listed on Cosmos DB's demo website.

Go ahead and add this document to the current collection using the **Data Explorer (Preview)** blade within the Azure Portal.

Code Listing 3-m: Another Sample Document

```
{
  "id": "09132",
  "description": "Grapes, red or green (European type, such as Thompson
seedless), raw",
  "tags": [
    {
      "name": "grapes"
    },
    {
      "name": "red or green (european type"
    },
    {
      "name": "such as thompson seedless)"
    },
    {
      "name": "raw"
    }
  ],
  "foodGroup": "Fruits and Fruit Juices",
  "servings": [
    {
      "amount": 1,
      "description": "cup",
      "weightInGrams": 151
    },
    {
      "amount": 10,
```

```
        "description": "grapes",
        "weightInGrams": 49
    },
    {
        "amount": 1,
        "description": "NLEA serving",
        "weightInGrams": 126
    }
  ]
}
```

Now let's run this next query for the document that we've just added, using the **Query** tab.

Code Listing 3-n: Another Query for the Sample Document

```
SELECT food.description,
       food.foodGroup,
       food.servings[0].description AS servingDescription,
       food.servings[0].weightInGrams AS servingWeight
FROM food
WHERE food.foodGroup = "Fruits and Fruit Juices"
      AND food.servings[0].description = "cup"
ORDER BY food.servings[0].weightInGrams DESC
```

Executing this query returns the following result.

Code Listing 3-o: Custom Projection Query Results

```
[
  {
    "description": "Grapes, red or green (European type, such as Thompson
     seedless), raw",
    "foodGroup": "Fruits and Fruit Juices",
    "servingDescription": "cup",
    "servingWeight": 151
  }
]
```

Because the **servings** property is an array with 3 items, by selecting
food.servings[0].description, the query is retrieving the description subproperty of the first
item within the **servings** array. This property is then projected as **servingDescription**.

It is also interesting to observe how DocumentDB has built-in support for **ORDER BY** and string
range queries. Detailed information about using **ORDER BY** and how this relates to the indexing
policy can be found here.

The TOP keyword

When there are many documents within a collection, it might be useful to limit the results to a certain number of documents. For instance, we might be interested in showing the first ten documents. This is where the **TOP** keyword comes in handy.

Take, for instance, the following example.

Code Listing 3-p: Query Using the TOP Keyword

```
SELECT TOP 10 food.id,
       food.description,
       food.tags,
       food.foodGroup
FROM food
WHERE food.foodGroup = "Snacks"
```

In this query, we are instructing DocumentDB to return the top ten documents in the collection that match the filtering criteria of the query.

Since, at the moment, we only have one document that matches these criteria, when we execute this query, we get the same result as described in Code Listing 3-l.


```
Results

    | Results: 1 - 1 | Request Charge: 2.93 RUs

[
    {
        "id": "19015",
        "description": "Snacks, granola bars, hard, plain",
        "tags": [
            {
                "name": "snacks"
            },
            {
                "name": "granola bars"
            },
            {
                "name": "hard"
            },
            {
                "name": "plain"
            }
        ],
        "foodGroup": "Snacks"
    }
]
```

Figure 3-p: Result of the Query Using the TOP Keyword

However, that result could have been different if we had had more documents of that same type.

The IN and BETWEEN keywords

As mentioned on DocumentDB's demo website, the **IN** keyword can be used to check whether a specified value matches any element in a given list. **BETWEEN** can be used to run queries against a range of values.

In order to understand this better, let's add the following document to the current collection using the **Data Explorer (Preview)** blade.

Code Listing 3-q: Another Sample Document

```
{
  "id": "05740",
  "description": "Chicken, Turkey and Duck",
  "tags": [
    {
      "name": "Chicken"
    },
    {
      "name": "Turkey"
    },
    {
      "name": "Duck"
    }
  ],
  "version": 1,
  "isFromSurvey": false,
  "foodGroup": "Poultry Products",
  "servings": [
    {
      "amount": 1,
      "description": "Yummy chicken",
      "weightInGrams": 200
    },
    {
      "amount": 1,
      "description": "Yummy Turkey",
      "weightInGrams": 300
    },
    {
      "amount": 1,
      "description": "Yummy Duck",
      "weightInGrams": 250
    }
  ]
}
```

Then add this one as well.

```json
{
  "id": "07050",
  "description": "Sausages and Luncheon Meats",
  "tags": [
    {
      "name": "German Sausage"
    },
    {
      "name": "Hot Dog"
    },
    {
      "name": "Spam"
    }
  ],
  "version": 1,
  "isFromSurvey": false,
  "foodGroup": "Sausages and Luncheon Meats",
  "servings": [
    {
      "amount": 1,
      "description": "Yummy sausage",
      "weightInGrams": 10
    },
    {
      "amount": 1,
      "description": "Yummy Hot Dog",
      "weightInGrams": 15
    },
    {
      "amount": 1,
      "description": "Yummy Spam",
      "weightInGrams": 20
    }
  ]
}
```

To see the **IN** and **BETWEEN** keywords in action, let's look at the following query.

Code Listing 3-s: Query Using the IN and BETWEEN Keywords

```sql
SELECT food.id,
       food.description,
       food.tags,
       food.foodGroup,
       food.version
FROM food
WHERE food.foodGroup
```

```
        IN ("Poultry Products", "Sausages and Luncheon Meats")
        AND (food.id BETWEEN "05740" AND "07050")
```

Based on the two documents we just added to the current collection using **Data Explorer**, running this query returns the following result.

```
[
    {
        "id": "05740",
        "description": "Chicken, Turkey and Duck",
        "tags": [
            {
                "name": "Chicken"
            },
            {
                "name": "Turkey"
            },
            {
                "name": "Duck"
            }
        ],
        "foodGroup": "Poultry Products",
        "version": 1
    },
    {
        "id": "07050",
        "description": "Sausages and Luncheon Meats",
        "tags": [
            {
                "name": "German Sausage"
            },
            {
                "name": "Hot Dog"
            },
            {
                "name": "Spam"
            }
        ],
        "foodGroup": "Sausages and Luncheon Meats",
        "version": 1
    }
]
```

Figure 3-q: Result of the Query Using the IN and BETWEEN Keywords

The usage of the **IN** and **BETWEEN** keywords has allowed us to filter out the document mentioned in Code Listing 3-j, even though this document has the exact same properties as the documents mentioned in Code Listings 3-q and 3-r.

Using these keywords and narrowing the query down to any **id** property that is between a range of **"05740"** and **"07050"** inclusive have allowed us to specifically return the documents that matched the string criteria chosen for the **foodGroup** property.

If there had been other documents with a **foodGroup** of **"Poultry Products"** or **"Sausages and Luncheon Meats**," with **id** values between **"05740"** and **"07050**," these would have also been returned as part of the result.

We can quickly verify that this is true by adding the following sample document to the current collection using **Data Explorer** and afterwards running the same query again.

The following document is pretty much a clone of the document mentioned in Code Listing 3-r, except that we've modified its **id** to be **"07049**."

Code Listing 3-t: Another Sample Document

```
{
  "id": "07049",
  "description": "Sausages and Luncheon Meats",
  "tags": [
    {
      "name": "German Sausage"
    },
    {
      "name": "Hot Dog"
    },
    {
      "name": "Spam"
    }
  ],
  "version": 1,
  "isFromSurvey": false,
  "foodGroup": "Sausages and Luncheon Meats",
  "servings": [
    {
      "amount": 1,
      "description": "Yummy sausage",
      "weightInGrams": 10
    },
    {
      "amount": 1,
      "description": "Yummy Hot Dog",
      "weightInGrams": 15
    },
    {
      "amount": 1,
```

```
      "description": "Yummy Spam",
      "weightInGrams": 20
    }
  ]
}
```

If we now run the query mentioned in Code Listing 3-s in the **Query** tab, we should also have as part of the result the document we've just added (mentioned in Code Listing 3-t). However, we still have filtered out the document mentioned in Code Listing 3-j. The result of the query would be as follows.

Code Listing 3-u: Result of the Query Using the IN and BETWEEN Keywords

```
[
  {
    "id": "07050",
    "description": "Sausages and Luncheon Meats",
    "tags": [
      {
        "name": "German Sausage"
      },
      {
        "name": "Hot Dog"
      },
      {
        "name": "Spam"
      }
    ],
    "foodGroup": "Sausages and Luncheon Meats",
    "version": 1
  },
  {
    "id": "05740",
    "description": "Chicken, Turkey and Duck",
    "tags": [
      {
        "name": "Chicken"
      },
      {
        "name": "Turkey"
      },
      {
        "name": "Duck"
      }
    ],
    "foodGroup": "Poultry Products",
    "version": 1
  },
```

```
  {
    "id": "07049",
    "description": "Sausages and Luncheon Meats",
    "tags": [
      {
        "name": "German Sausage"
      },
      {
        "name": "Hot Dog"
      },
      {
        "name": "Spam"
      }
    ],
    "foodGroup": "Sausages and Luncheon Meats",
    "version": 1
  }
]
```

Querying documents with projections

As mentioned, DocumentDB supports JSON projections within its queries. To see this in action, let's add the following document to the current collection using **Data Explorer**.
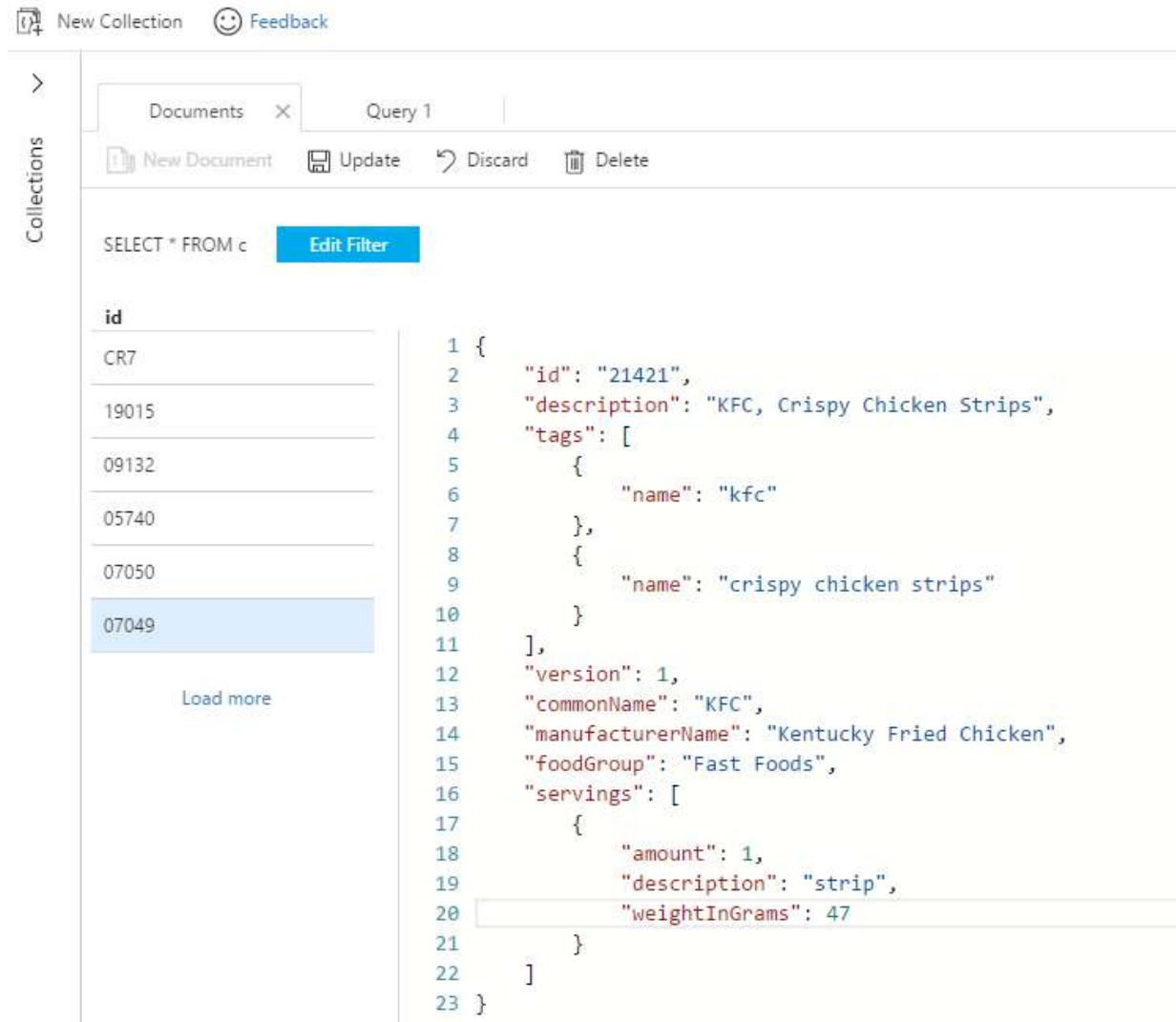
Figure 3-r: Adding a New Document Using Document Explorer

Considering the following query, let's see how we can project some custom properties as part of the result.

Code Listing 3-v: Query Using JSON Projections

```
SELECT {
    "Company": food.manufacturerName,
    "Brand": food.commonName,
    "Serving Description": food.servings[0].description,
    "Serving in Grams": food.servings[0].weightInGrams,
    "Food Group": food.foodGroup
    } AS Food
FROM food
WHERE food.id = "21421"
```

Executing this query in the **Query** tab would return the following result.

Figure 3-s: Result of the JSON Projections Query

The JSON object included in the **SELECT** clause of the query has allowed DocumentDB to project how the value of each property is returned in a different way, using alternative (more user-friendly) aliases.

JSON projections are a great way to customize results and make them more intuitive, user-friendly, and easier to read. They are especially convenient when writing queries that will be used to create reports.

Querying documents with JOIN

DocumentDB's **JOIN** supports intradocument (within a document) joins but does not support interdocument (between different documents) joins.

Instead, the **JOIN** keyword is used to perform intradocument (nested) queries. In order to understand this better, let's add the following document to the current collection using **Data Explorer**.

Code Listing 3-w: Another Sample Document

```
{
  "id": "09052",
  "description": "Blueberries, canned, heavy syrup, solids and liquids",
  "tags": [
    {
      "name": "blueberries"
    },
```

```
    {
      "name": "canned"
    },
    {
      "name": "heavy syrup"
    },
    {
      "name": "solids and liquids"
    }
  ],
  "version": 1,
  "isFromSurvey": false,
  "foodGroup": "Fruits and Fruit Juices",
  "nutrients": [
    {
      "id": "221",
      "description": "Alcohol, ethyl",
      "nutritionValue": 0,
      "units": "g"
    },
    {
      "id": "303",
      "description": "Iron, Fe",
      "nutritionValue": 0.33,
      "units": "mg"
    }
  ],
  "servings": [
    {
      "amount": 1,
      "description": "cup",
      "weightInGrams": 256
    }
  ]
}
```

Having added this document to the current collection, let's execute this next query that uses the **JOIN** keyword.

Code Listing 3-x: Query Using the JOIN Keyword

```
SELECT tag.name
FROM food
JOIN tag IN food.tags
WHERE food.id = "09052"
```

Running this query produces the following result.

Results

| Results: 1 - 4 | Request Charge: 2.79 RUs

```
[
    {
        "name": "blueberries"
    },
    {
        "name": "canned"
    },
    {
        "name": "heavy syrup"
    },
    {
        "name": "solids and liquids"
    }
]
```

Figure 3-t: Result of the JOIN Query

If we take a moment to analyze this query, we can see that the **JOIN** focuses on the **food.tags**. In other words, the **JOIN** is narrowing down the scope of the query to the **food.tags** array and the rest of the document is ignored except for the **food.id** property, which is used in the **WHERE** clause of the query.

So, in essence, the **JOIN** allows the **SELECT** to focus on a specific section of the document.

As you can see, the **JOIN** in DocumentDB behaves in a totally different way than what you would expect from traditional T-SQL in a relational database.

This is likely the biggest difference between traditional SQL and DocumentDB databases.

Other interesting bits

DocumentDB has some other tricks up its sleeve. One of them is the ability to create an alias of an alias. This may sound redundant, but it's a way to make queries shorter and write less.

Let's look at an example query, which is a slight modification of the previous query that used the **JOIN** keyword.

```
SELECT tag.name
FROM food AS f
JOIN tag IN f.tags
WHERE f.id = "09052"
```

The trick is simply to re-alias the collection's alias (**food**) to a shorter variable name, in this case **f**, using the **AS** keyword.

In this particular example, there's no real added value to doing this. However, in a larger query with many more variables, re-aliasing is a great way to shorten the query by typing less.

In any case, it's a nice trick. It's good to know and use in particular situations, especially in long queries.

DocumentDB has another nice trick that allows us to retrieve only subsections of documents. The source can also be reduced to a smaller subset.

For instance, enumerating only a subtree in each document, the subroot could then become the source. Let's explore an example.

Code Listing 3-z: Subroot Query

```
SELECT *
FROM f.servings
```

Any valid JSON value (not undefined) that can be found in the source will be considered for inclusion in the result of the query.

Executing this query will produce the following result.

Code Listing 3-aa: Execution of the Previous Query

```
[
  [
    {
      "amount": 1,
      "description": "bar",
      "weightInGrams": 21
    },
    {
      "amount": 1,
      "description": "bar (1 oz)",
      "weightInGrams": 28
    },
    {
      "amount": 1,
      "description": "bar",
```

```json
      "weightInGrams": 25
    }
  ],
  [
    {
      "amount": 1,
      "description": "cup",
      "weightInGrams": 151
    },
    {
      "amount": 10,
      "description": "grapes",
      "weightInGrams": 49
    },
    {
      "amount": 1,
      "description": "NLEA serving",
      "weightInGrams": 126
    }
  ],
  [
    {
      "amount": 1,
      "description": "Yummy sausage",
      "weightInGrams": 10
    },
    {
      "amount": 1,
      "description": "Yummy Hot Dog",
      "weightInGrams": 15
    },
    {
      "amount": 1,
      "description": "Yummy Spam",
      "weightInGrams": 20
    }
  ],
  [
    {
      "amount": 1,
      "description": "Yummy chicken",
      "weightInGrams": 200
    },
    {
      "amount": 1,
      "description": "Yummy Turkey",
      "weightInGrams": 300
    },
    {
```

```
      "amount": 1,
      "description": "Yummy Duck",
      "weightInGrams": 250
    }
  ],
  [
    {
      "amount": 1,
      "description": "Yummy sausage",
      "weightInGrams": 10
    },
    {
      "amount": 1,
      "description": "Yummy Hot Dog",
      "weightInGrams": 15
    },
    {
      "amount": 1,
      "description": "Yummy Spam",
      "weightInGrams": 20
    }
  ],
  [
    {
      "amount": 1,
      "description": "strip",
      "weightInGrams": 47
    }
  ],
  [
    {
      "amount": 1,
      "description": "cup",
      "weightInGrams": 256
    }
  ]
]
```

Notice how the results returned correspond only to the **servings** subsections of the documents within the current collection and no other properties are even considered as the source of the information.

Documents within the collection that do not have a **servings** subsection are not even taken into account for the results.

Summary

Throughout this chapter, we've explored the various ways we can query DocumentDB directly from the Azure Portal using the built-in Query Explorer blade and a familiar SQL flavored syntax.

We've learned the basics of DocumentDB querying, explored scalar queries, and seen how to query documents independent of their structure, shape, and properties.

Even though the SQL flavored syntax looks very similar to what we are used to from the relational database world, we've also seen some differences and aspects that are unique to DocumentDB and give it a unique character.

DocumentDB's querying capabilities are not only intuitive, but also rich and powerful. They provide a straightforward way of interacting with collections and are easy to implement and execute.

I highly recommend you keep exploring all the querying capabilities that DocumentDB has to offer. The official documentation website is a great resource to look at and read thoroughly.

So far it's been an interesting journey and there's more to explore. Up next, we'll delve into the details of using DocumentDB from client code. This should also be fun and interesting to explore. Thanks for reading!
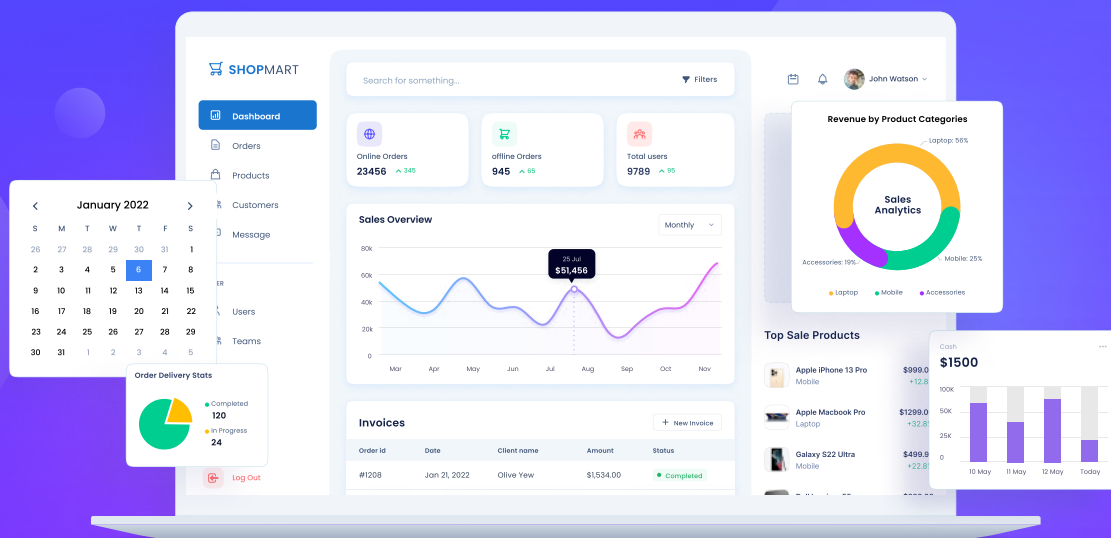
Chapter 4 Client-Side Development

Introduction

In the previous chapters, we had a great introduction to the DocumentDB API. We created a database and a collection, added documents, and queried data using a familiar SQL flavored syntax. We also explored briefly how to migrate existing data using the Data Migration Tool.

In this chapter, we'll focus our attention on writing client-side code using the .NET Framework and C# to interact with DocumentDB.

Although it is possible to write client-side code using DocumentDB's REST API, Microsoft provides SDKs for the most popular programming languages, making it much easier to interact with DocumentDB. So we won't be exploring the REST API, but instead using the .NET SDK and writing code with C#.

We'll start by looking at how to connect to DocumentDB from client-side code, talking about master keys and resources, and writing code examples that work with databases, collections, and documents.

The same principles that will be applied and demonstrated with the .NET SDK should be equally applicable to any other programming language SDK provided by Microsoft for DocumentDB, but syntactical differences will exist. We won't cover any other SDKs in this chapter, though.

I highly encourage you to keep exploring the site's excellent [documentation](#) to learn how to use any of the other SDKs that exist. There are native SDKs for Node.js, Python, and Java.

Let's not wait any longer to get started. Have fun!

Master keys

In order to be able to do anything with DocumentDB, it is essential to connect to it. Establishing a connection is the first and main step before we can perform any operation on DocumentDB.

Creating a connection to DocumentDB requires an endpoint and a key. The endpoint is the URL to the DocumentDB account, which has the form of *https://accountname.documents.azure.com*.

The key contains your credentials to DocumentDB. There are two types of keys: a master key and resource tokens.

The master key grants total access to DocumentDB and you can do pretty much anything with it. You can think of it as the *sa* account in the SQL Server world. It provides full access to the entire DocumentDB account and should not be distributed to end user machines. There are actually two types of master keys: a primary and secondary key.

Having a primary and secondary master key is a great way to enable key rotation and change the master key without affecting any running application.

Furthermore, DocumentDB provides keys for read-only applications. Let's have a quick look on the Azure Portal to see where we can find the keys.
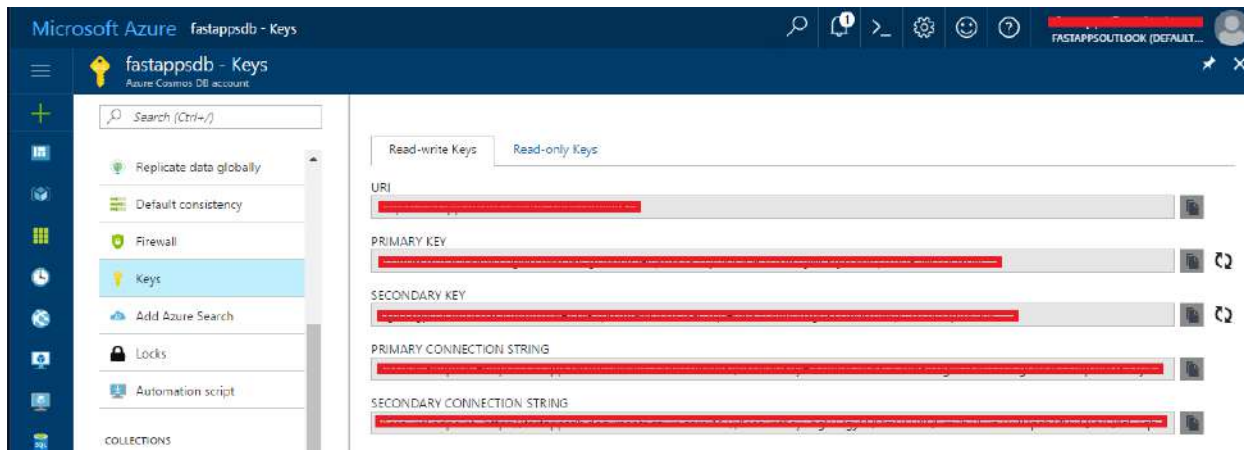

Figure 4-a: Keys for a Cosmos DB Account

Both the read-write and read-only keys can be found by clicking **Keys** on the main blade of the Cosmos DB account within the Azure Portal.

Granular access

In addition to master keys that provide full access to DocumentDB, resource tokens also let you connect. This provides granular control over security. Resource tokens can only access specific resources within DocumentDB, not all of them.

Resource tokens are based on user permissions. First it is necessary to create one or more users for the database, and then permissions are added for each user. Each permission has a resource token for all read or full access to a single user resource, which may apply to a collection, document, attachment, stored procedure, user-defined function, or trigger. Access to administrative resources (database account, databases, and users and permissions) is not allowed using resource tokens, but only when using a master key.

Resource tokens are a great way to allow granular access to specific resources based on permissions granted to specific users for certain things.

Introduction to the .NET SDK

Working with the DocumentDB .NET SDK is very straightforward. You get started by creating a `DocumentClient` instance and then supplying the connection information: an endpoint and a key.

Once you have an instance, you can invoke methods to access DocumentDB resources. At this point, you can create, modify, and delete databases, collections, documents, and other resources.

Let's go ahead and install the .NET SDK from NuGet so we can start coding. Create a new Visual Studio 2017 Console Application project and add the NuGet package.
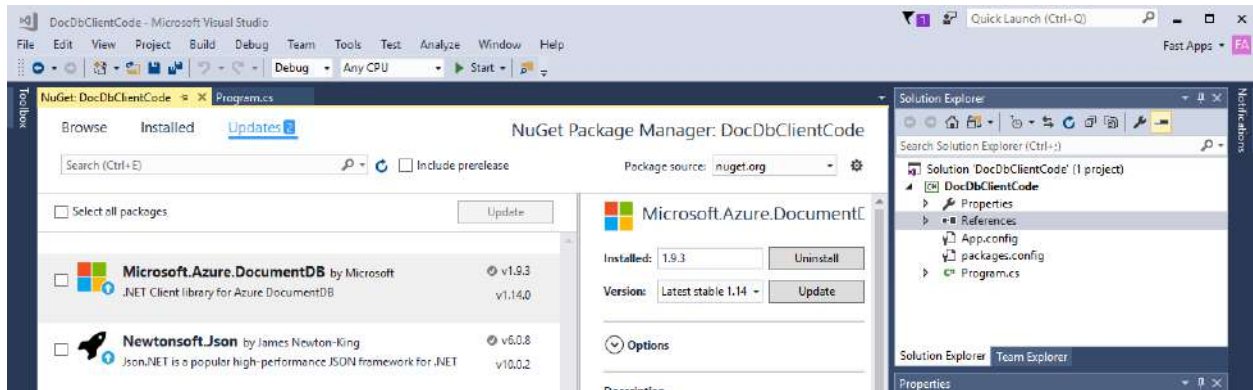

Figure 4-b: Installing the .NET SDK as a NuGet Package with Visual Studio 2017

Once the .NET SDK has been installed, we'll have the **Microsoft.Azure.Documents.Client** and **Newtonsoft.Json** assemblies referenced in our Visual Studio solution.

In the real world, you probably won't be creating a console application, but will likely use a different Visual Studio project such as a Web API acting as a middle-tier or web application. In any case, the DocumentDB code will be just the same. It's also easier to understand the functionality using a console application project.

In the sample application we'll be writing, we'll initially focus on connecting to DocumentDB, then displaying the names of the available databases and the collections for each one.

We'll also have a method to display the documents stored within a collection. Let's look at how we can achieve this. First, let's display the list of databases on our DocumentDB account.

Listing 4-a: Displaying a List of Databases

```csharp
using Microsoft.Azure.Documents.Client;
using System;

namespace DocDbClientCode
{
    class Program
    {
        public const string cStrEndPoint = "<< Your Endpoint >>";
        public const string cStrKey = "<< Your Primary Key >>";

        static void Main(string[] args)
        {
            ListDbs();
```

```
            Console.ReadLine();
        }

        public static void ListDbs()
        {
            using (var client = new DocumentClient(
                    new Uri(cStrEndPoint), cStrKey))
            {
                var dbs = client.CreateDatabaseQuery();
                foreach (var db in dbs)
                {
                    Console.WriteLine(
                    "Database Id: {0}; Rid {1}", db.Id, db.ResourceId);
                }
            }
        }
    }
}
```

Notice how we've had to include the **Microsoft.Azure.Documents.Client** namespace in order to be able to create an instance of **DocumentClient**. We also need the **System** namespace in order to create an instance of **Uri**, used for passing the DocumentDB endpoint.

With the **DocumentClient** instance created, we can then call the **CreateDatabaseQuery** method, which returns a list of objects, each containing information about a DocumentDB database. The list is then written to the console using the **Id** and **ResourceId** properties.

Running this program produces the following output.

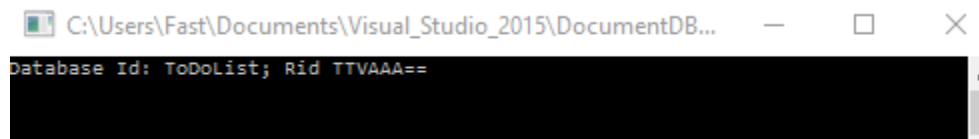*Figure 4-c: Output of the Program Listing DocumentDB Databases*

That was quite straightforward. Let's now explore collections and documents.

Accessing collections

Now that we have some basic code that connects to DocumentDB and is also able to retrieve the name of the databases present on the DocumentDB account, we can look at accessing collections. First, let's modify a bit of the code we already have.

Code Listing 4-b: Displaying a List of Collections

```csharp
using Microsoft.Azure.Documents;
using Microsoft.Azure.Documents.Client;
using System;
using System.Collections.Generic;
using System.Linq;

namespace DocDbClientCode
{
    class Program
    {
        public const string cStrEndPoint = "<< Your Endpoint >>";
        public const string cStrKey = "<< Your Primary Key >>";

        static void Main(string[] args)
        {
            ListDbs();
            Console.ReadLine();
        }

        public static void ListDbs()
        {
            using (var client = new DocumentClient(new Uri(cStrEndPoint),
                    cStrKey))
            {
                var dbs = client.CreateDatabaseQuery();
                foreach (var db in dbs)
                {
                    Console.WriteLine(
                    "Database Id: {0}; Rid {1}", db.Id, db.ResourceId);
                    ListCollections(client, db, db.Id);
                }
            }
        }

        public static void ListCollections(DocumentClient client,
        Database db, string dbname)
        {
            if (client != null && db != null)
            {
                List<DocumentCollection> collections =
                client.CreateDocumentCollectionQuery
                (db.SelfLink).ToList();

                Console.WriteLine(
                "{0} collections for database: {1}",
                collections.Count.ToString(), dbname);
```

```
                foreach (DocumentCollection col in collections)
                {
                    Console.WriteLine("Collection Id: {0}; Rid {1}",
                    col.Id, col.ResourceId);
                }
            }
        }
    }
}
```

The main difference between the previous code example and this one is that we've now included a method called **ListCollections** that is responsible for retrieving the list of collections that exists for each DocumentDB database.
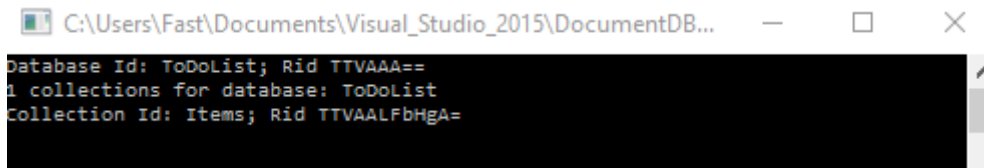
This method is called from the **ListDbs** method inside the **foreach** loop for every DocumentDB database.

Because we already have the **DocumentClient** and **Database** instances available, we pass them along with the name of the current DocumentDB database to the **ListCollections** method.

Now let's look at the **ListCollections** method in detail. This method internally invokes the **CreateDocumentCollectionQuery** method, using the resource URI of the current **Database** (**db.SelfLink**) in order to get a **List<DocumentCollection>** object, which represents all the collections available for the **dbname** database.

The **List<DocumentCollection>** object is then looped with a **foreach** and each **DocumentCollection** instance is then written to the console, using the **Id** and **ResourceId** properties.

If we execute this code, we'll get the following result.

Figure 4-d: Output of the Program Listing DocumentDB Databases and Collections

Accessing documents

We've seen how to access collections within a DocumentDB database, so we'll now look at accessing documents within collections. Let's modify our code to achieve this.

We'll simply create a new method called **ListDocuments** and modify **ListCollections** in order to invoke **ListDocuments**.

Because the rest of the previous code doesn't change, we'll include in the following listing these two methods only and not the full code.

Code Listing 4-c: Displaying a List of Documents

```
public static void ListCollections(DocumentClient client, Database db,
string dbname)
{
    if (client != null && db != null)
    {
        List<DocumentCollection> collections =
        client.CreateDocumentCollectionQuery(db.SelfLink).ToList();

        Console.WriteLine("{0} collections for database: {1}",
        collections.Count.ToString(), dbname);
        foreach (DocumentCollection col in collections)
        {
            Console.WriteLine("Collection Id: {0}; Rid {1}",
            col.Id, col.ResourceId);
            ListDocuments(client, dbname, col.Id);
        }
    }
}

public static void ListDocuments(DocumentClient client, string dbName,
string collName)
{
    if (client != null)
    {
        IEnumerable<Document> docs =
        from c in client.CreateDocumentQuery(
            "dbs" + "/" + dbName + "/colls/" + collName)
        select c;

        if (docs != null)
        {
            Console.WriteLine("Documents for collection {0}", collName);
            foreach (var doc in docs)
            {
                Console.WriteLine(
                "Document Id: {0}; Rid {1} ", doc.Id, doc.ResourceId);
            }
        }
    }
}
```
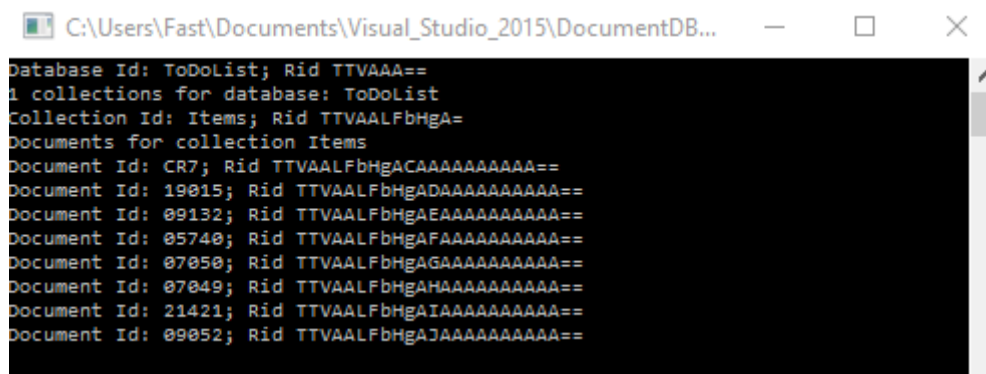
Let's explore the **ListDocuments** method quickly. By using the **DocumentClient** instance, we invoke the **CreateDocumentQuery** method.

The **CreateDocumentQuery** method receives as a parameter a string that represents a relative URI to the location of all documents within a specific collection. This string has two fixed parts: **"dbs"** and **"colls"**.

After **"dbs"**, the name of the DocumentDB database is concatenated, using the variable **dbName**. After **"colls"**, the name of the collection to be queried is also concatenated, using the variable **collName**.

This allows the **CreateDocumentQuery** method to retrieve all documents contained within the collection **collName**. The list of documents is assigned to an **IEnumerable<Document>** object, which is then looped with a **foreach**, writing to the console each document's **Id** and **ResourceId**. Running the updated program with this latest code outputs the following results.

Figure 4-e: Output of the Program Listing DocumentDB Databases, Collections, and Documents

Querying documents

We've seen so far how to connect to DocumentDB and list databases, collections, and documents. However, we haven't seen yet how to retrieve information that is specific to certain document types. This is what we'll be focusing on now.

In order to retrieve information specific to certain document types, we need to be able to specify exactly what type of data we want to retrieve. The way we do that is by defining a C# class that represents the structure of the type of document (with the property names) we want to retrieve.

So far throughout this e-book we've added three document types to our only DocumentDB collection. Two of them are quite similar and are related to food, and the third one is totally unrelated, containing data for the famous soccer player Cristiano Ronaldo.

So if we want to retrieve the properties for the document that contains Cristiano Ronaldo's document type details, we need to define a C# class as follows. We can do this inside the **DocDbClientCode** namespace. I use the **sealed** modifier here on my class to prevent other classes from inheriting from it, but it's not required.

Listing 4-d: Definition Class for Cristiano Ronaldo's Document Type

```
public sealed class CR7DocType
{
    public string id { get; set; }
    public string Name { get; set; }
    public string LastName { get; set; }
    public string Nationality { get; set; }
    public string BirthPlace { get; set; }
}
```

This class definition matches exactly the properties that the Cristiano Ronaldo document type contains. Let's quickly look at this document using **Data Explorer** to refresh our memories.

```
1  {
2      "id": "CR7",
3      "Name": "Cristiano Ronaldo",
4      "LastName": "dos Santos Aveiro",
5      "Nationality": "Portuguese",
6      "Birthplace": "Funchal (Madeira), Portugal",
7      "_rid": "TTVAALFbHgACAAAAAAAAAA==",
8      "_self": "dbs/TTVAAA==/colls/TTVAALFbHgA=/docs/TTVAALFbHgACAAAAAAAAAA==/",
9      "_etag": "\"5b003014-0000-0000-0000-59171f370000\"",
10     "_attachments": "attachments/",
11     "_ts": 1494687543
12  }
```

Figure 4-f: The CR7 Document as Seen with Data Explorer

Now that we've defined the C# class that defines its properties, we can write a method to retrieve it. We won't modify any other part of the code that has been written so far, so we'll only add this new method. Let's have a look.

Code Listing 4-e: Method to List Cristiano Ronaldo's Documents

```
public static void ListCR7DocType(CR7DocType cr7, DocumentClient client,
string dbName, string collName)
{
    if (client != null)
    {
        IEnumerable<CR7DocType> docs =
            from c in client.CreateDocumentQuery<CR7DocType>(
                "dbs" + "/" + dbName + "/colls/" + collName)
            where c.Name.ToUpper().
                Contains(cr7.Name.ToUpper())
            where c.LastName.ToUpper().
                Contains(cr7.LastName.ToUpper())
```

```
            select c;

        if (docs != null)
        {
            foreach (var doc in docs)
            {
                Console.WriteLine("id: {0}", doc.id);
                Console.WriteLine("Name: {0}", doc.Name);
                Console.WriteLine("LastName: {0}", doc.LastName);
                Console.WriteLine("Nationality: {0}", doc.Nationality);
                Console.WriteLine("Birthplace: {0}", doc.Birthplace);
            }
        }
    }
}
```

This method pretty much works the same way as the **ListDocuments** method. The main difference is that when we invoke the **CreateDocumentQuery** method, we are now explicitly telling it that its **T** is a **CR7DocType** class. This allows us to focus the query to look only at **CR7DocType** documents.

The result is an **IEnumerable<CR7DocType>** object that we can loop through and use to output to the console each document's properties. On the **ListCollections** method, let's replace the call to **ListDocuments** with a call to **ListCR7DocType**.

Code Listing 4-f: ListCollections Invoking the ListCR7DocType Method
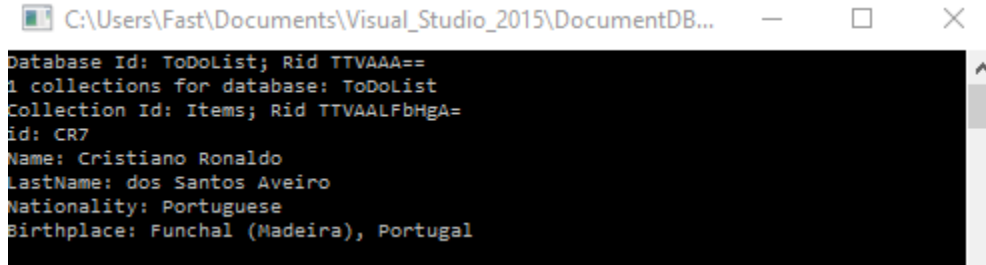
```
public static void ListCollections(DocumentClient client, Database db,
string dbname)
{
    if (client != null && db != null)
    {
        List<DocumentCollection> collections =
            client.CreateDocumentCollectionQuery(db.SelfLink).ToList();

        Console.WriteLine("{0} collections for database: {1}",
            collections.Count.ToString(), dbname);

        foreach (DocumentCollection col in collections)
        {
            Console.WriteLine(
                "Collection Id: {0}; Rid {1}", col.Id, col.ResourceId);
                ListCR7DocType(new CR7DocType { Name = "Ronaldo",
                LastName = "Aveiro"}, client, dbname, col.Id);
        }
    }
}
```

If we now run the updated program with these changes, we get the following output.

Figure 4-g: Output of the Program Invoking the ListCR7DocType Method

Notice how we are passing a **CR7DocType** object with abbreviated string values for the **Name** and **LastName** properties. The LINQ query on the **ListCR7DocType** method is able to return the resulting document because it performs a **Contains** string search rather than a complete string comparison.

Now let's do the same for one of the food document types, just like we did with CR7. Let's go ahead and define some C# definition classes for one of the food document types, inside the **DocDbClientCode** namespace.

Code Listing 4-g: Definition Classes for One of the Food Document Types

```
public sealed class Tags
{
    public string name { get; set; }
}

public sealed class Servings
{
    public int amount { get; set; }
    public string description { get; set; }
    public int weightInGrams { get; set; }
}

public sealed class FoodDocType
{
    public string id { get; set; }
    public string description { get; set; }
    public int version { get; set; }
    public bool isFromSurvey { get; set; }
    public string foodGroup { get; set; }
    public Tags[] tags { get; set; }
    public Servings[] servings { get; set; }
}
```

Now let's create a method to list these food document types.

Listing 4-h: Method to List Food Type Documents

```csharp
public static void ListFoodDocType(FoodDocType fd, DocumentClient client,
string dbName, string collName)
{
    if (client != null)
    {
        IEnumerable<FoodDocType> docs =
            from c in client.CreateDocumentQuery<FoodDocType>(
            "dbs" + "/" + dbName + "/colls/" + collName)
            where c.description.ToUpper().
                Contains(fd.description.ToUpper())
            select c;

        if (docs != null)
        {
            foreach (var doc in docs)
            {
                Console.WriteLine("id: {0}", doc.id);
                Console.WriteLine("description: {0}", doc.description);
                Console.WriteLine("Version: {0}", doc.version);
                Console.WriteLine("isFromSurvey: {0}",
                    doc.isFromSurvey.ToString());
                Console.WriteLine("foodGroup: {0}", doc.foodGroup);
            }
        }
    }
}
```

Let's now modify the **ListCollections** method so it can invoke this newly created **ListFoodDocType** method.

Listing 4-i: ListCollections invoking the ListFoodDocType Method

```csharp
public static void ListCollections(DocumentClient client, Database db,
string dbname)
{
    if (client != null && db != null)
    {
        List<DocumentCollection> collections =
            client.CreateDocumentCollectionQuery(db.SelfLink).ToList();

        Console.WriteLine("{0} collections for database: {1}",
            collections.Count.ToString(), dbname);

        foreach (DocumentCollection col in collections)
        {
            Console.WriteLine("Collection Id: {0}; Rid {1}", col.Id,
                col.ResourceId);
```

```
            ListFoodDocType(new FoodDocType { description = "Snacks" },
                client, dbname, col.Id);
        }
    }
}
```

If we execute the updated program, we get the following output.
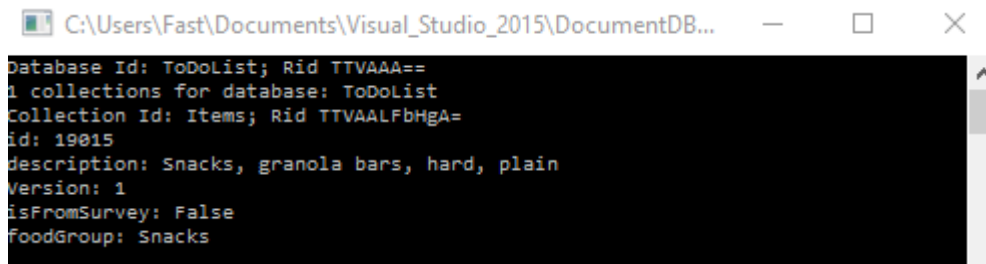
Figure 4-h: Output of the Program Invoking the ListFoodDocType Method

If we quickly analyze the **ListFoodDocType** method, we can see that it is almost identical to the **ListCR7DocType** method, with the exception that the LINQ query is returning an **IEnumerable<FoodDocType>** object instead of an **IEnumerable<CR7DocType>** one. The rest is pretty much the same.

The main difference is the definition class we pass as a **T** parameter to the **CreateDocumentQuery** method in order to be able to retrieve the right document type we expect to get.

Now that we've seen how to query documents, let's briefly explore how to add a document to a collection before wrapping up this chapter.

Adding a document

Adding a document to an existing collection is quite easy. However, it is an asynchronous operation, so it's best to wrap this up nicely in two methods. One method creates the document and the other creates the asynchronous task that invokes the method that creates the document on the collection. Let's see how we can implement this.

Let's start off by adding a **using** statement to our existing code. This is because we'll be using a **Task** object to invoke the asynchronous method that will create the new document.

Code Listing 4-j: Adding the Threading .NET Library

```
using System.Threading.Tasks;
```

The rest of the previous **using** statements, as mentioned in Code Listing 4-b, remain the same. Now, let's create the method that will create the document. We'll name it **CreateDocType**.

```
public static async Task<Document> CreateDocType(FoodDocType fd,
DocumentClient client, string dbName, string collName)
{
    if (client != null)
    {
        string url = "dbs" + "/" + dbName + "/colls/" + collName;
        Document id = await client.CreateDocumentAsync(url, fd);

        return (id != null) ? client.CreateDocumentQuery(url).
        Where(d => d.Id == id.Id).AsEnumerable().FirstOrDefault() : null;
    }
    else
        return null;
}
```

The important part of the **CreateDocType** method, and the one we will be focusing on, is the call to **CreateDocumentAsync**, which is really the one responsible for creating the new document on the collection specified by the **url** string.

Because **CreateDocumentAsync** is an asynchronous method, we have to **await** it and also mark the **CreateDocType** method as **async**. Once the document has been created, a call to **CreateDocumentQuery** is carried out to actually verify that the document has indeed been created within the collection.

With this in place, we can then create a wrapper method we will call **CreateDoc** to execute asynchronous code that will invoke **CreateDocType**.

Code Listing 4-l: The New CreateDoc Wrapper Method

```
public static async void CreateDoc(string dbName, string collName)
{
    await Task.Run(
    async () =>
    {
        using (var client = new DocumentClient(
            new Uri(cStrEndPoint), cStrKey))
        {
            FoodDocType fd = new FoodDocType { id = "TestFoodDoc",
                description = "Organic food",
                isFromSurvey = false,
                foodGroup = "Organic", version = 1 };
            Document issue = await CreateDocType(fd, client,
                dbName, collName);
        }
    });
}
```

The **CreateDoc** method simply invokes **Task.Run** and an anonymous **async** method is passed as a lambda expression. This creates a **FoodDocType** object with some properties, representing the document that will be created using **CreateDocType**.

Finally, we can modify the **Main** method of the program to invoke **CreateDoc**. It would look as follows.

Code Listing 4-m: The Updated Main Program Method

```csharp
static void Main(string[] args)
{
    CreateDoc("ToDoList", "Items");
    Console.ReadLine();
}
```

If we execute this code and then open **Data Explorer** on the Azure Portal, we can see that the document has been created.


```
 1 {
 2     "id": "TestFoodDoc",
 3     "description": "Organic food",
 4     "version": 1,
 5     "isFromSurvey": false,
 6     "foodGroup": "Organic",
 7     "tags": null,
 8     "servings": null,
 9     "_rid": "TTVAALFbHgAKAAAAAAAAAA==",
10     "_self": "dbs/TTVAAA==/colls/TTVAALFbHgA=/docs/TTVAALFbHgAKAAAAAAAAAA==/",
11     "_etag": "\"5b00def7-0000-0000-0000-591756710000\"",
12     "_attachments": "attachments/",
13     "_ts": 1494701680
14 }
```

Figure 4-i: The TestFoodDoc Document Added Through C# Code

Great! So we've now seen how to add a document using C#. The following code listing is the full updated source code of the program we've created so far.

Code Listing 4-n: The Full Program Code

```csharp
using Microsoft.Azure.Documents;
using Microsoft.Azure.Documents.Client;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

namespace DocDbClientCode
```

```csharp
{
    public sealed class CR7DocType
    {
        public string id { get; set; }
        public string Name { get; set; }
        public string LastName { get; set; }
        public string Nationality { get; set; }
        public string Birthplace { get; set; }
    }

    public sealed class Tags
    {
        public string name { get; set; }
    }

    public sealed class Servings
    {
        public int amount { get; set; }
        public string description { get; set; }
        public int weightInGrams { get; set; }
    }

    public sealed class FoodDocType
    {
        public string id { get; set; }
        public string description { get; set; }
        public int version { get; set; }
        public bool isFromSurvey { get; set; }
        public string foodGroup { get; set; }
        public Tags[] tags { get; set; }
        public Servings[] servings { get; set; }
    }

    class Program
    {
        public const string cStrEndPoint = ""<< Your Endpoint >>"";
        public const string cStrKey = "Your Primary Key";

        static void Main(string[] args)
        {
            //Comment out the line below and uncomment ListDbs();
            CreateDoc("ToDoList", "Items");

            //Uncomment and comment out CreateDoc(…);
            //ListDbs();
            Console.ReadLine();
        }

        public static async void CreateDoc(string dbName, string
```

```csharp
            collName)
        {
            await Task.Run(
            async () =>
            {
                using (var client = new DocumentClient(new
                Uri(cStrEndPoint), cStrKey))
                {
                    FoodDocType fd = new FoodDocType { id =
                        "TestFoodDoc", description = "Organic food",
                        isFromSurvey = false, foodGroup = "Organic",
                        version = 1 };
                    Document issue = await CreateDocType(fd, client,
                      dbName, collName);
                }
            });
        }

        public static async Task<Document> CreateDocType(FoodDocType fd,
        DocumentClient client, string dbName, string collName)
        {
            if (client != null)
            {
                string url = "dbs" + "/" + dbName + "/colls/" + collName;
                Document id = await client.CreateDocumentAsync(url, fd);

                return (id != null) ? client.CreateDocumentQuery(url).
                    Where(d => d.Id ==
                    id.Id).AsEnumerable().FirstOrDefault() : null;
            }
            else
                return null;
        }

        public static void ListDbs()
        {
            using (var client = new DocumentClient(
                new Uri(cStrEndPoint), cStrKey))
            {
                var dbs = client.CreateDatabaseQuery();
                foreach (var db in dbs)
                {
                    Console.WriteLine("Database Id: {0}; Rid {1}",
                        db.Id, db.ResourceId);
                    ListCollections(client, db, db.Id);
                }
            }
        }
```

```csharp
        public static void ListCollections(DocumentClient client,
        Database db, string dbname)
        {
            if (client != null && db != null)
            {
                List<DocumentCollection> collections =
                    client.CreateDocumentCollectionQuery
                    (db.SelfLink).ToList();

                Console.WriteLine("{0} collections for database: {1}",
                    collections.Count.ToString(), dbname);
                foreach (DocumentCollection col in collections)
                {
                    Console.WriteLine("Collection Id: {0}; Rid {1}",
                        col.Id, col.ResourceId);

                    // Comment out each List instruction one at a time
                    // to see the different results :)

                    //ListDocuments(client, dbname, col.Id);
                    //ListCR7DocType(new CR7DocType { Name = "Ronaldo",
                    //    LastName = "Aveiro"}, client, dbname, col.Id);

                    //ListFoodDocType(new FoodDocType { description =
                    //    "Snacks" }, client, dbname, col.Id);
                }
            }
        }

        public static void ListDocuments(DocumentClient client, string
        dbName, string collName)
        {
            if (client != null)
            {
                IEnumerable<Document> docs =
                    from c in client.CreateDocumentQuery(
                    "dbs" + "/" + dbName + "/colls/" + collName)
                    select c;

                if (docs != null)
                {
                    Console.WriteLine("Documents for collection {0}",
                        collName);
                    foreach (var doc in docs)
                    {
                        Console.WriteLine("Document Id: {0}; Rid {1} ",
                            doc.Id, doc.ResourceId);
                    }
                }
```

```
        }
    }

    public static void ListFoodDocType(FoodDocType fd, DocumentClient
    client, string dbName, string collName)
    {
        if (client != null)
        {
            IEnumerable<FoodDocType> docs =
                from c in
                client.CreateDocumentQuery<FoodDocType>
                    ("dbs" + "/" + dbName + "/colls/" + collName)
                where c.description.ToUpper().
                    Contains(fd.description.ToUpper())
                select c;

            if (docs != null)
            {
                foreach (var doc in docs)
                {
                    Console.WriteLine("id: {0}", doc.id);
                    Console.WriteLine("description: {0}",
                        doc.description);
                    Console.WriteLine("Version: {0}", doc.version);
                    Console.WriteLine("isFromSurvey: {0}",
                        doc.isFromSurvey.ToString());
                    Console.WriteLine("foodGroup: {0}",
                        doc.foodGroup);
                }
            }
        }
    }

    public static void ListCR7DocType(CR7DocType cr7, DocumentClient
    client, string dbName, string collName)
    {
        if (client != null)
        {
            IEnumerable<CR7DocType> docs =
                from c in client.CreateDocumentQuery<CR7DocType>
                    ("dbs" + "/" + dbName + "/colls/" + collName)
                where c.Name.ToUpper().
                    Contains(cr7.Name.ToUpper())
                where c.LastName.ToUpper().
                    Contains(cr7.LastName.ToUpper())
                select c;

            if (docs != null)
            {
```

```csharp
                    foreach (var doc in docs)
                    {
                        Console.WriteLine("id: {0}", doc.id);
                        Console.WriteLine("Name: {0}", doc.Name);
                        Console.WriteLine("LastName: {0}", doc.LastName);
                        Console.WriteLine("Nationality: {0}",
                            doc.Nationality);
                        Console.WriteLine("Birthplace: {0}",
                            doc.Birthplace);
                    }
                }
            }
        }
    }
}
```

Summary

Throughout this chapter, we've explored how to interact with DocumentDB from C#. We've explained how to list various object types, such as databases, collections, and documents. We've also shown how to create documents.

However, we've just scratched the surface of what is possible with the .NET DocumentDB SDK.

The DocumentDB documentation website contains invaluable information on how to interact with the service and the .NET section is particularly rich and well documented. I highly encourage you to keep exploring and studying this subject with this great tutorial.

Furthermore, I've written another e-book for Syncfusion called *Customer Success for C# Developers Succinctly* that has a chapter devoted to writing a simple CRM app using DocumentDB along with C#. It is worth exploring.

In the next chapter we'll focus on exploring the server-side features of DocumentDB and how to write server code in JavaScript that performs operations on databases, collections, and documents.

Hopefully this chapter has been an eye-opener to what is possible with DocumentDB and C#, and the examples have been fun to follow and implement. Thanks for reading!

Chapter 5 Server-Side Development

Introduction

In the previous chapters, we explored several of DocumentDB's features. We covered how to set it up and also did some development, such as querying documents through the Azure Portal using the SQL flavored syntax and writing code with the .NET SDK in C#.

The journey so far has been quite interesting and we've managed to get a good glimpse of what is possible and why DocumentDB is a great choice when considering a NoSQL back-end.

In this chapter, we'll move our attention to server-side development. We'll focus on how we can write code that runs on the server using the Azure Portal and does more than just query collections and documents. We'll look at code that can perform inserts, updates, and deletions on documents, but run from the server directly in the form of stored procedures, triggers, and user-defined functions.

Knowledge of basic JavaScript is required and the examples should be fun to follow and implement.

By the end of this chapter, and therefore the e-book, you should have a good understanding of how to develop with DocumentDB to have a scalable back-end for your app. Have fun!

Server-side programming model

In DocumentDB it is possible to create server-side code as stored procedures, triggers, or user-defined functions (UDFs). Microsoft calls this server-side code in DocumentDB "JavaScript as T-SQL" because, as you might have figured out by now, it is written in JavaScript.

Server-side code runs inside DocumentDB with full transactional support (ACID guarantee), so all changes are rolled back in the event of an error. And if there are no errors, all commits are done at the same time.

Also, server-side code runs inside a sandboxed environment that is isolated from all other users and under bounded execution, which makes sure that code performs to certain standards and does not take too long to run. If the server code takes too long to run, then DocumentDB will abort the code and roll back any changes done by that code.

For code that needs a long time to execute, we can implement a continuation model for long running processes.

Furthermore, DocumentDB also supports error throttling. This is normally returned in the response header (`x-ms-retry-after-ms`), which indicates how long we need to wait before retrying.

Server-side code acts and runs on a collection level in which it is itself defined. A server-side JavaScript function works with a **context** object. The **context** exposes a **collection**, a **request**, and a **response**.

To understand server-side programming in JavaScript with DocumentDB, it is necessary to understand DocumentDB's resource model, which is shown in Figure 2-h.

Server-side code is written directly on the Azure Portal, so it cannot be debugged with Visual Studio like client-side code can. It's a good idea to check out the DocumentDB server-side scripting documentation throughout the course of this chapter.

So let's get started by creating stored procedures.

Stored procedures

DocumentDB stored procedures are a great way to allow the execution of custom business logic on the server by registering a JavaScript function that acts like a T-SQL stored procedure.

The way to do this is to create a JavaScript function that uses a **context** to access the **collection** and **response**. It receives input values as function arguments and operates on any document in the collection, returning a **response**. It is also possible to adjust the code of a stored procedure to accommodate long running processes by implementing a continuation model, which works under bounded execution, but returns continuation information.

Let's have a look at how we can create a stored procedure. On the Azure Portal, using **Data Explorer**, click the **ellipsis (…)** button next to **Items**. This will display a menu with several options. One of them is labeled **New Stored Procedure**. Click on it to create a stored procedure.
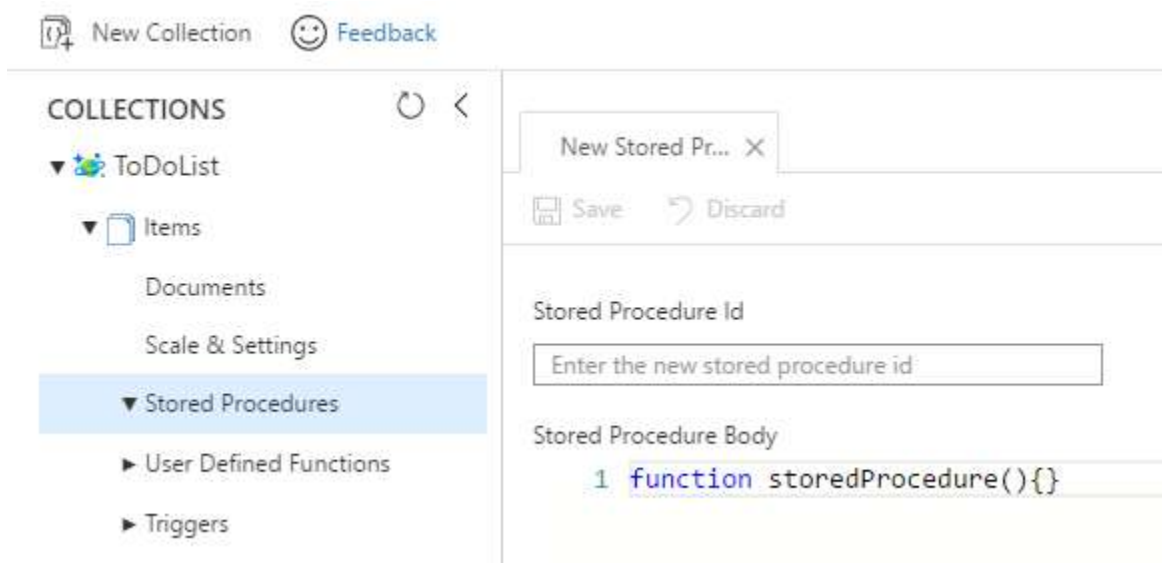

Figure 5-a: Creating a New Stored Procedure

Enter a **Stored Procedure Id** and the **Stored Procedure Body**. Some template code is provided by default.

Let's remove the default code provided and write a simple example as follows.

Code Listing 5-a: A Simple Stored Procedure

```
function spHiDocumentDB()
{
    var context = getContext();
    var response = context.getResponse();
    response.setBody('This is a simple DocumentDB stored procedure');
}
```

Then, on the main Cosmos DB blade, look for **Script Explorer**, open the created stored procedure, and then click the **Save & Execute** button, as shown in the following figure.
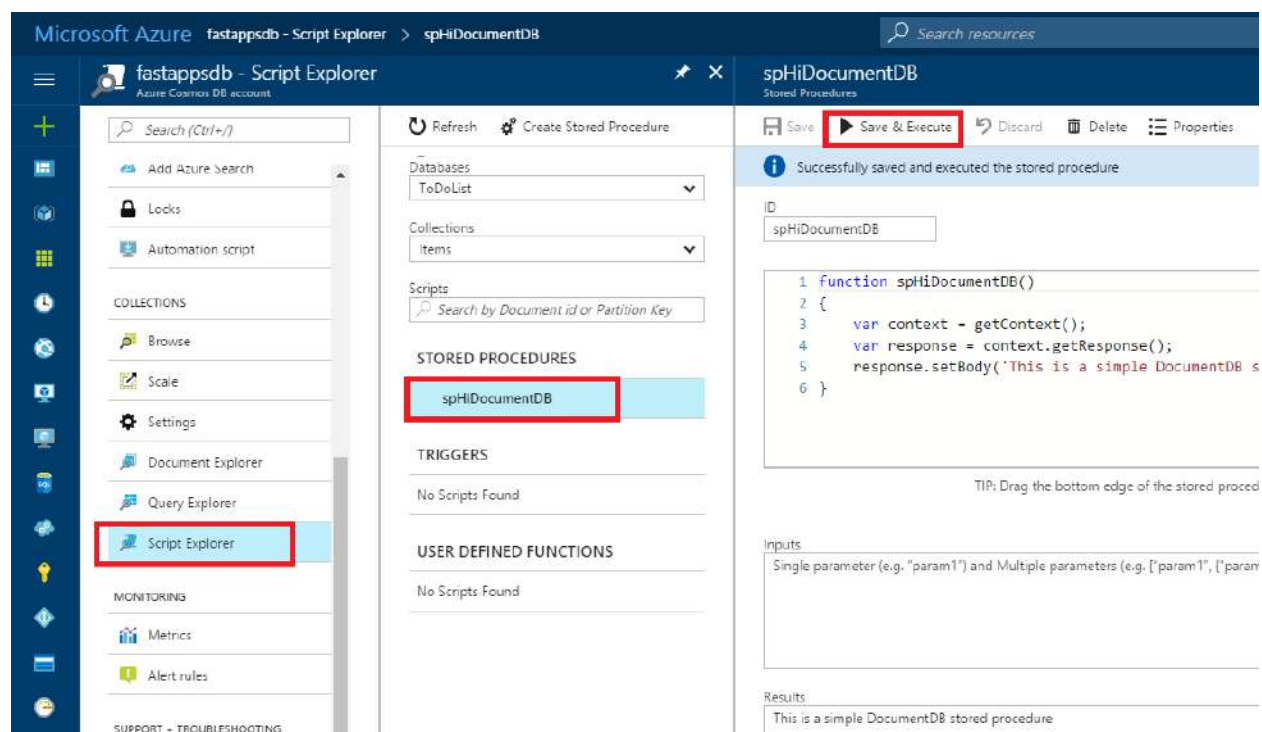

Figure 5-b: Execution of the Stored Procedure

Notice how the result of the execution of the stored procedure is displayed under the **Results** text area.

All we did here was get the **context** and **response** objects and then output a response by calling **setBody**. Very simple. So let's now do something more exciting and create some stored procedures that do something useful.

First, let's write a stored procedure that you can use to pass a document as a parameter and it will insert that document within the DocumentDB collection.

Code Listing 5-b: A Stored Procedure for Creating a Document

```
function spCreateDoc(doc)
{
    var context = getContext();
    var collection = context.getCollection();
    var response = context.getResponse();

    collection.createDocument(collection.getSelfLink(), doc, {},
        function (err, doc) {
            if (err)
                throw new Error('Error creating document: ' + err.Message);
            response.setBody(doc);
        }
    );
}
```

Using **Script Explorer**, click **Create Stored Procedure**, remove the default code, and enter the code provided in Code Listing 5-b. In the **Inputs** area, enter the JSON object shown in the following figure. Then click **Save & Execute**.
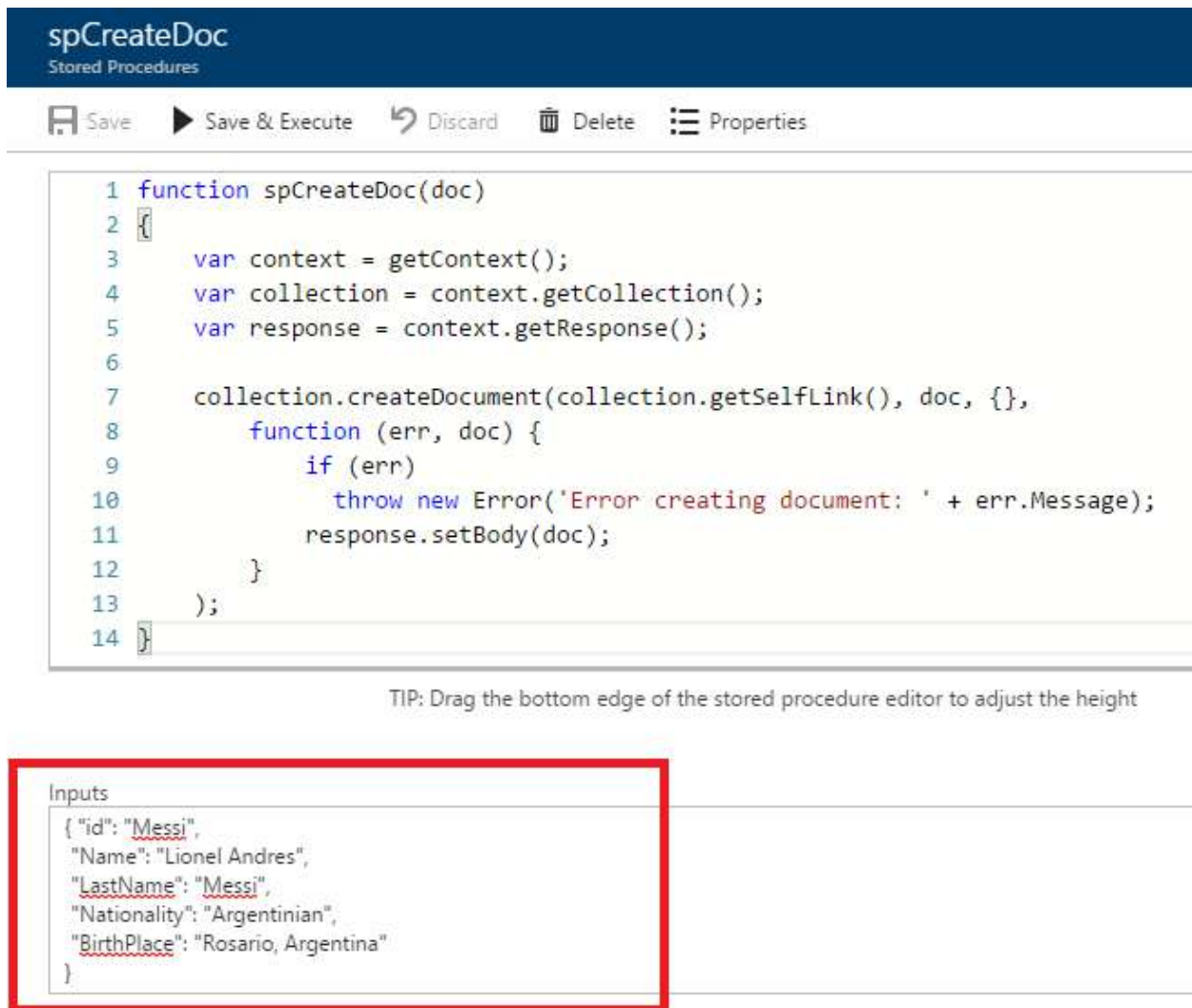
spCreateDoc
Stored Procedures

Save   ▶ Save & Execute   ⤺ Discard   🗑 Delete   ☰ Properties

```
1  function spCreateDoc(doc)
2  {
3      var context = getContext();
4      var collection = context.getCollection();
5      var response = context.getResponse();
6
7      collection.createDocument(collection.getSelfLink(), doc, {},
8          function (err, doc) {
9              if (err)
10                 throw new Error('Error creating document: ' + err.Message);
11             response.setBody(doc);
12         }
13     );
14 }
```

TIP: Drag the bottom edge of the stored procedure editor to adjust the height

Inputs

```
{ "id": "Messi",
  "Name": "Lionel Andres",
  "LastName": "Messi",
  "Nationality": "Argentinian",
  "BirthPlace": "Rosario, Argentina"
}
```

Figure 5-c: The spCreateDoc Stored Procedure

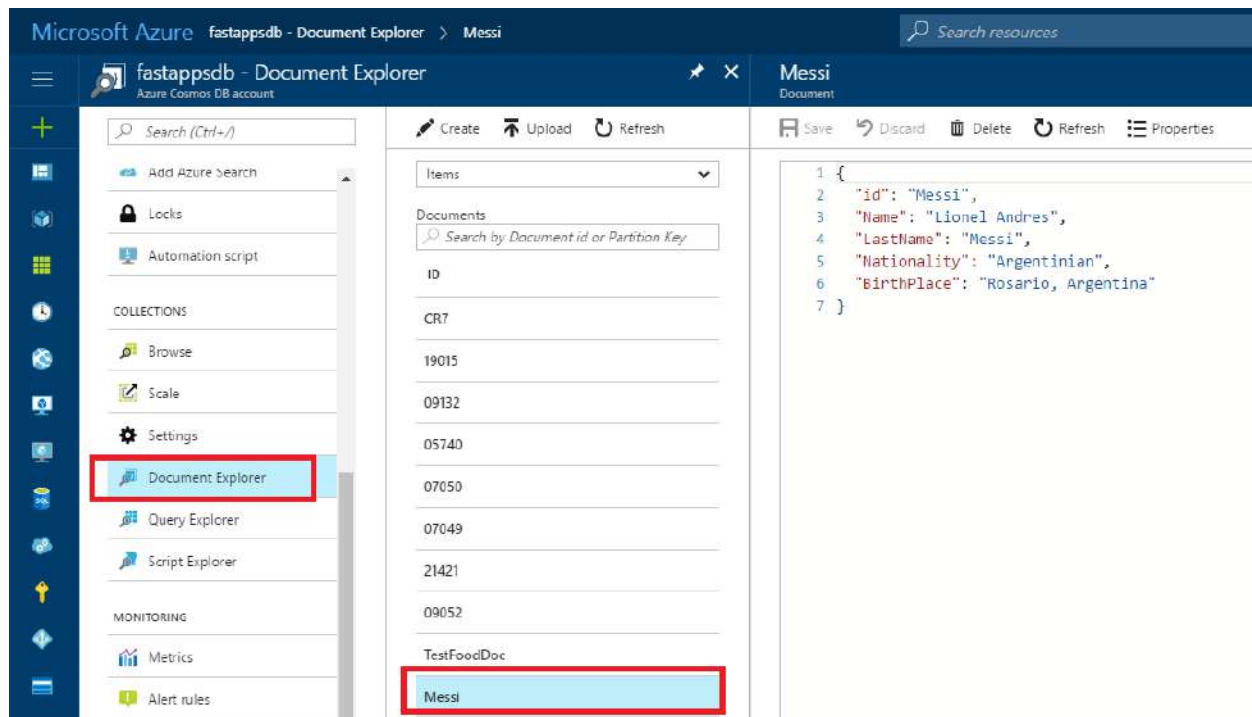If we open **Document Explorer**, we'll be able to see the newly created document.

Figure 5-d: The New Messi Document Created with spCreateDoc

If we closely examine **spCreateDoc**, we can see that we get the **context**, **response** objects, and also the current **collection**. Once we have them, we call the **createDocument** method and pass the collection's **SelfLink**, the **doc** that we want to insert, and a post-insertion function which is a callback (anonymous function) that displays the result, using **response.setBody**.

Also very simple and straightforward. Let's look at another example.

Code Listing 5-c: A Stored Procedure for Checking and Creating a Document

```
function spCreateDocIfIdIsUnique(doc)
{
    var context = getContext();
    var coll = context.getCollection();
    var collLink = coll.getSelfLink();
    var response = context.getResponse();

    CheckIdAndCreateDoc();

    function createDoc()
    {
        coll.createDocument(collLink, doc, {},
        function (err, doc) {
            if (err)
              throw new Error('Error creating document: ' + err.message);

            response.setBody(doc);
```

```
        });
    }

    function CheckIdAndCreateDoc()
    {
        var query = {
            query: 'SELECT VALUE coll.id FROM coll WHERE coll.id = @id',
            parameters: [{name: '@id', value: doc.id}]
        };

        var ok = coll.queryDocuments(collLink, query, {},
            function (err, results) {
                if (err) {
                    throw new Error('Error querying for document' +
                        err.message);
                }
                if (results.length == 0) {
                    createDoc();
                }
                else {
                    response.setBody('Document ' + doc.id +
                        ' already exists.');
                }
            });

        if (!ok) {
            throw new Error ('Timeout for querying document...');
        }
    }
}
```

This stored procedure creates a document, but it first checks whether there's already a document in **coll** with the same **id**.

If there is an existing document with the specified **id**, the document is not inserted and an error message is displayed.

Now, in order to test this stored procedure, let's run it with the same Messi document that already exists (the same one that was used in the previous query).
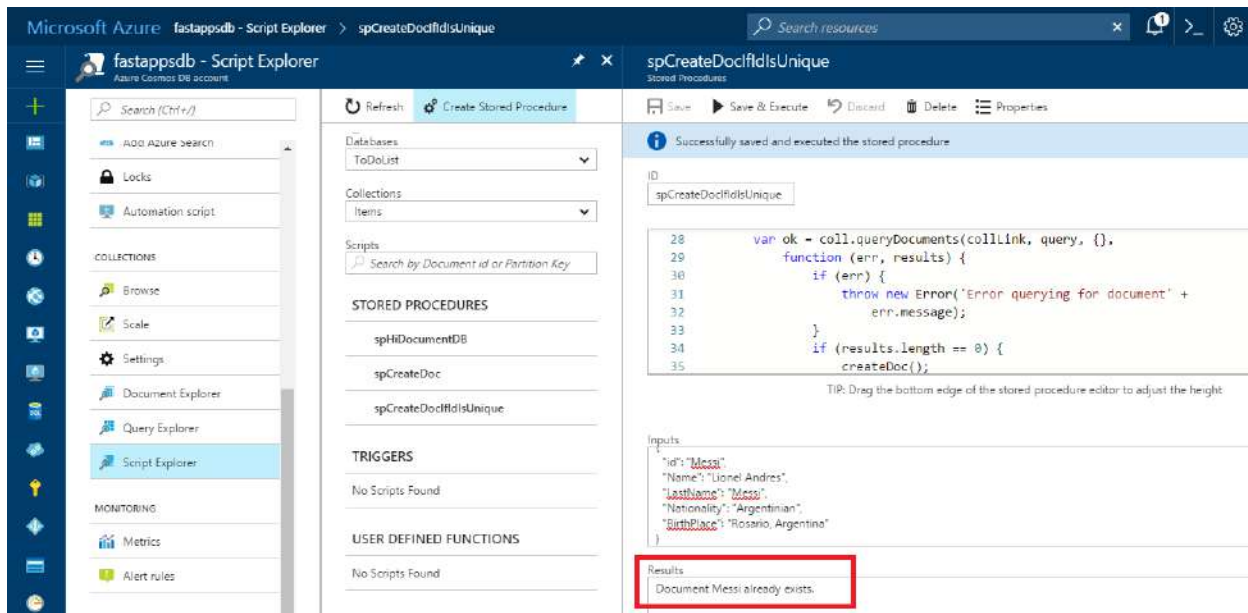
Figure 5-e: The Execution of spCreateDocIfIdIsUnique

As expected, the stored procedure clearly describes that a document with the **id** Messi already exists and therefore this is the result returned.

There are two very interesting parts to this stored procedure. One is that it has a function called **CheckIdAndCreateDoc** that checks whether the document already exists or not by executing a query on the collection using the familiar SQL flavored syntax we already know.

The other interesting part is that if **queryDocuments** returns a **null**, **undefined**, or **false** value, this means that the query execution has not been completed correctly and an exception is thrown in order to time out the stored procedure.

We've seen how we can combine SQL flavored syntax inside a stored procedure and how straightforward it is to write logic in JavaScript to insert documents in a collection. Let's now focus our attention on triggers and then look at user-defined functions.

Triggers

DocumentDB triggers are JavaScript server-side functions associated with create (insert) or replace document operations.

There are two types of triggers: pre-triggers and post-triggers.

A pre-trigger runs before the operation executes and has no access to the **response** object. It basically provides a hook into the document that is about to be inserted or replaced.

A pre-trigger is particularly useful if you want to enforce some type of validation before the document gets inserted or replaced, or if you want to perform certain verifications on properties.

A post-trigger runs after the operation executes and is less common than a pre-trigger. With a post-trigger, you can run logic that is executed after the document has been saved, but before it has actually been committed.

There's one important thing to mention about triggers: they are not automatically triggered like in a traditional relational database. In that sense, the name is a bit misleading. They must be explicitly requested with an operation in order to be executed.

Let's go ahead and create a very simple validation pre-trigger that will be executed just before a document is created. We can do this easily by going into **Script Explorer** within the Azure Portal and then clicking **Create Trigger**. Let's use the following code.

Code Listing 5-d: A Pre-Trigger that Validates New Documents

```
function validateNameExists() {
  var collection = getContext().getCollection();
  var request = getContext().getRequest();
  var doc = request.getBody();

  if (!doc.name) {
    throw new Error('Document must include a "name" property.');
  }
}
```

This is how we'll see it on the Azure Portal.

Figure 5-f: The Pre-Trigger on the Azure Portal (Script Explorer)

The **Trigger Operation** has been set to **Create** so that the trigger is only executed when a document is created. By default, the **Trigger Operation** is set to **All**, which also includes the **Delete** and **Replace** operations. Once done, give it a name (enter a value in the **ID** text box—I'll call it **preTriggerValidateIdExists**) and then click **Save**.

To see it in action, let's go ahead and slightly modify the client-side application we wrote with Visual Studio in the previous chapter. Let's create a **TestTrigger** method inside the **Program** class.

Code Listing 5-e: TestTrigger Method in Our Client Application

```
public static async void TestTrigger(string dbName, string collName)
{
    await Task.Run(
    async () =>
    {
        using (var client = new DocumentClient(
            new Uri(cStrEndPoint), cStrKey))
        {
            FoodDocType fd = new FoodDocType { id = "TestFoodDoc",
                description = "Organic food",
                isFromSurvey = false, foodGroup = "Organic",
                version = 1 };
            string url = "dbs" + "/" + dbName + "/colls/" + collName;

            try
            {
                Document issue = await client.CreateDocumentAsync(url,
                fd, new RequestOptions { PreTriggerInclude = new[] {
                "preTrigValidateIdExists" } });
            }
            catch (Exception ex)
            {
                Console.WriteLine("Exception: "  + ex.ToString());
            }
        }
    });
}
```

In general, using **async void** for methods is not recommended because any exceptions thrown will be raised directly on the **SynchronizationContext** that was active when the **async void** method started. I use **async void** here to keep the main ideas of calling a DocumentDB trigger clear.

The **Main** method of the **Program** class has been modified as follows.

Code Listing 5-f: TestTrigger Method Invoked inside the Main Method

```
static void Main(string[] args)
{
    TestTrigger("ToDoList", "Items");
    Console.ReadLine();
}
```
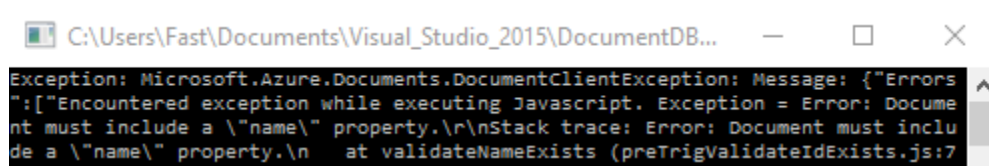
If we now execute this code, we get this result.

Figure 5-g: The Pre-Trigger Execution from the Client Application

We can clearly see that an exception is thrown and the returned message is the same one that we included in the pre-trigger code on the server, indicating that a **name** property must be included when creating the document.

The pre-trigger is actually invoked when **CreateDocumentAsync** is called. This is achieved by creating a **RequestOptions** object and specifying the **ID** that was given to the trigger when we created it through the **Script Explorer** within the Azure Portal.

When a trigger throws an exception, the transaction it is part of aborts, and everything gets rolled back. This includes the work done by the trigger itself and the work done by whatever request caused the trigger to execute.

So that's a quick overview on triggers and how they can be invoked. Let's move on now to user-defined functions.

User-defined functions

DocumentDB's user-defined functions are the third type of JavaScript server-side functions that can be created and will seem familiar. User-defined functions are a great way to write custom business logic that can be called from within queries, such as extending DocumentDB's flavored SQL with functionality that it doesn't provide.

User-defined functions can't make changes to the database—they're read-only. Instead, they provide a way to extend DocumentDB SQL with custom code that otherwise would not be possible.

Because user-defined functions require a full scan (they cannot use index), adding one on a **WHERE** clause can have performance implications. If it is necessary to add one on a **WHERE** clause, I advise you to limit the query as much as possible with other conditions or by just selecting any properties needed (but not all of them).

It is also important to know that user-defined functions have no access to the **context** object, so they are essentially compute-only.

Let's get started and create a user-defined function. We can do this by going into the Azure Portal, clicking **Script Explorer**, and then clicking the **Create User Defined Function** button. After doing this, we get the following default example provided by DocumentDB.

Figure 5-h: Out-of-the-Box User-Defined Function Provided by DocumentDB

Let's now create a user-defined function we can use to check for matching a particular regular expression.
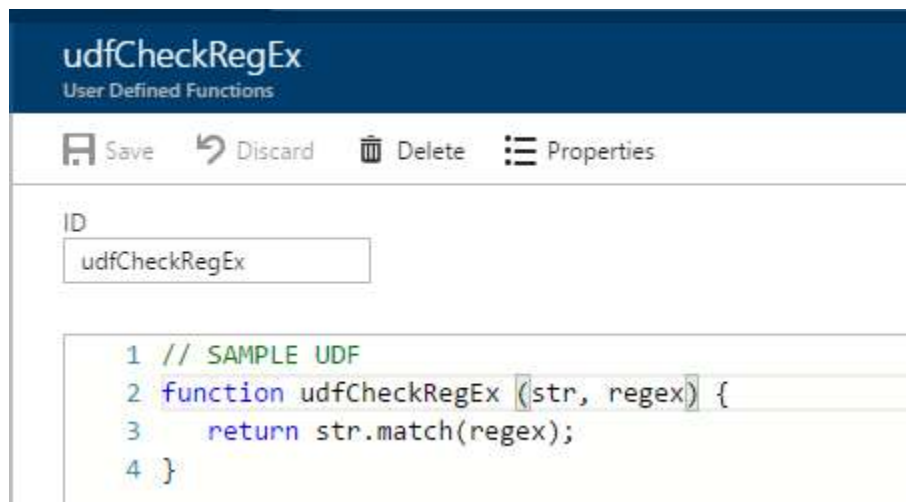

Figure 5-i: A Pattern Matching RegEx User-Defined Function

Here, we are using JavaScript's built-in regular expression pattern matching to check whether a match is found within **str**.

This function will come in handy when looking for string patterns in documents that match a specific variable string value.

Let's look at an example and expand our Visual Studio application by adding a **TestUdf** method to the **Program** class.

Code Listing 5-g: TestUdf Method in Our Client Application

```
public static void TestUdf(string dbName, string collName)
{
    using (var client = new DocumentClient(new Uri(cStrEndPoint),
    cStrKey))
```

```
    {
        string query = "SELECT c.id FROM c WHERE " +
        " udf.udfCheckRegEx(c.id, 'Messi') != null";

        string url = "dbs" + "/" + dbName + "/colls/" + collName;

        Console.WriteLine("Querying for Messi documents");
        var docs = client.CreateDocumentQuery(url, query).ToList();

        Console.WriteLine("{0} docs found", docs.Count);
        foreach (var d in docs)
        {
            Console.WriteLine("{0}", d.id);
        }
    }
}
```

Now, let's modify the **Main** method within the **Program** class to see this working correctly.

Code Listing 5-h: TestUdf Method Invoked inside the Main Method

```
static void Main(string[] args)
{
    TestUdf("ToDoList", "Items");
    Console.ReadLine();
}
```

Before running this code, go to the Azure Portal, open **Document Explorer**, and create a new Messi document. Simply copy the details from the original Messi document previously created and change the **id** of the new one to **Messi1**.
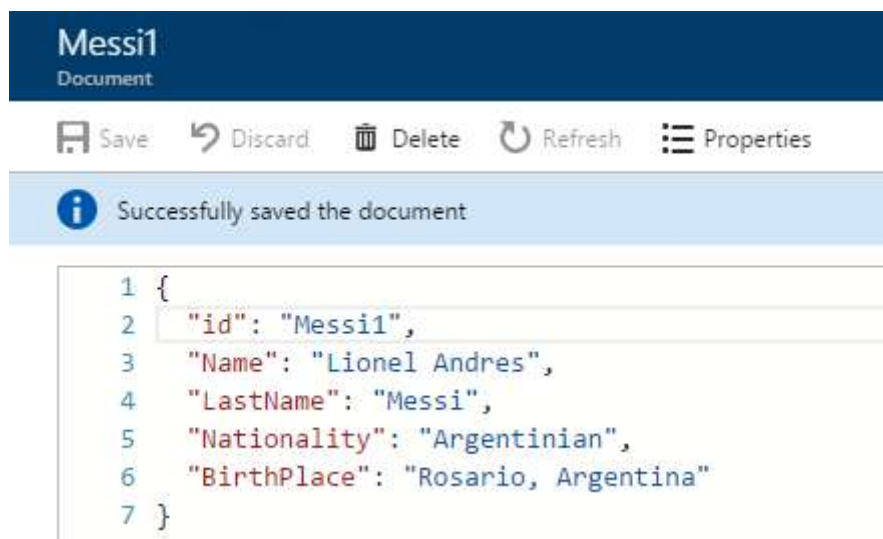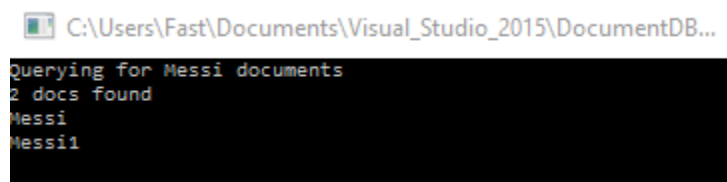

Figure 5-j: The New Messi1 Document

Now if you run the updated code, you will see the following results.

Figure 5-k: The User-Defined Function Execution from the Client Application

Notice that on the **TestUdf** method within the client code, the **udfCheckRegEx** user-defined function created through the Azure Portal is prefixed with the string **udf**. So the actual name of the function from client code is **udf.udfCheckRegEx**.

In order to invoke a user-defined function, it is mandatory to always prefix it with **udf** using dotted notation.

That concludes our overview of user-defined functions. It was simple and easy to understand, but this barely scratches the surface of what is possible and can be achieved.

The DocumentDB documentation is a great resource for learning more about programming with them, so I encourage you to explore this topic further there.

Additional options

Before wrapping up, there are additional items that are interesting to mention and worth exploring if you want to expand your knowledge and understanding of DocumentDB.

The Azure documentation site is filled with valuable details and there's a wealth of information about topics that were not covered in this e-book, but would certainly be beneficial for anyone willing to do serious development with DocumentDB.

For instance, there's valuable information on how to partition data from client-side applications, how automatic indexing works, indexing policies, using multi-region accounts, how to perform global database replication, how to use DocumentDB with PowerBI for business intelligence, and many more topics.

So there's plenty to keep learning and exploring. Also, if you still have any doubts about what kind of use cases DocumentDB is best suited for, here's a great resource that addresses this topic.

Hadoop and DocumentDB

DocumentDB is a great companion for Hadoop, as explained here. It is not uncommon to want to do some kind of analytics on DocumentDB data or push the output of analytics into an operational store like DocumentDB. In the NoSQL world, by far the most common technology for doing analytics is Hadoop. In order to make this technology even easier to use, Microsoft provides a Hadoop Connector for DocumentDB.

This connector works with Azure's Hadoop service, called HDInsight, and it also works with other Hadoop implementations, whether in the cloud or on premise.

HDInsight is an amazing solution from Microsoft that is also available through an Azure subscription, which consists of a managed Apache Hadoop, Spark, R, HBase, and Storm cloud service, fully hosted on Azure.

So whenever you think about Big Data, give DocumentDB a thought. You'll be in for a treat.

Summary

DocumentDB is a relatively simple, scalable, and yet very powerful NoSQL database. It is able to provide advanced data management capabilities such as a SQL flavored query language, stored procedures, triggers, user-defined functions, and atomic transactions.

As seen, it provides out-of-the-box support for JSON and JavaScript, besides supporting many of the most popular programming languages such as C# (.NET), Node.js, Python, and Java.

DocumentDB provides the flexibility of not being locked into a schema and is hosted under a steady and reliable managed cloud service and platform.

As more and more companies continue to move their computing operations to the cloud, DocumentDB is becoming a true contender and great choice, both in terms of pricing and the features it supports.

In an increasingly connected society, DocumentDB is also a great choice when considering building a social network or social application and choosing a platform to support it. Here's a very interesting article from a Microsoft blog that touches this topic.

As Microsoft continues to add more features to the product, the experience of using DocumentDB keeps improving and any barriers to entry continue to fade.

Throughout this e-book, we've explored the most developer-related facets of DocumentDB and how to interact with it in order to get you up-to-speed and able to appreciate its many benefits.

The overall objective was to make the experience of getting off the ground with this technology an enjoyable one and include some simple, yet fun examples on how to do it. Hopefully, that's been achieved.

As mentioned before, I have another Syncfusion e-book called *Customer Success for C# Developers Succinctly* that devotes a chapter to creating a simple CRM application using DocumentDB. It includes examples on how to do more advanced client-side filtering and querying and provides a good use case for the adoption of DocumentDB.

To conclude, it's been an absolute pleasure writing this e-book and the journey has been full of interesting small challenges that have helped shape a better picture of what this amazing NoSQL database can do. The Microsoft Azure team has done an outstanding job creating this product.

Thank you so much for reading along and following this journey of exploration through DocumentDB. Until our next adventure, have fun!