Azure DocumentDB Lab

Overview

Azure DocumentDB is a NoSQL, JSON document database built for big data solutions that require scaling and high availability.

This hands-on lab will step you through the following features:

1. **Querying** - Connect to a DocumentDB database and execute a simple query
2. **Filtering** - Execute ad-hoc queries on schemaless JSON data.

About the code

This lab uses a simple ASP.NET MVC website as a test application. This application allows you to write arbitrary query commands and execute them against our test databases. Any result set will be rendered automatically into the JSON response panel. There are arrows to navigate left and right through the results.

To begin, open the `Azure DocumentDB Lab.sln` solution in Visual Studio 2015 and press `F5` to compile and launch the web app on the local machine.

Note: The DocumentDB that we will be querying was created via the Azure Portal. For more information on the Azure Portal refer to the **Appendix** at the end of this lab.

Scenario 1

In this scenario. We will change the MVC application to send a query to the DocumentDB server.

Part One

To begin, open the `Azure DocumentDB Lab.sln` solution in Visual Studio 2015 and press `F5` to compile and launch the web app on the local machine.

You should be presented with an application that looks like this:

This page is designed to take the query that the user writes and pass it to a DocumentDB server that we have set up for the purposes of this demo.

Type this query into the query editor:

```
SELECT *
FROM c
```

...and click on **Run It!**

Currently there are no results - we need to finish implementing the DocumentDB call first.

Part Two

In the visual studio solution navigate to the `HomeController` class in the `LabWeb` project.

Find the `Query` action method. There is a line of code that looks like this:

```
IDocumentQuery<dynamic> docQuery = null;
```

We will modify it to create and send a DocumentDB query.

The query text from the page is passed into the action via the `query` variable. Change it to the following:

```
var collectionUri =
UriFactory.CreateDocumentCollectionUri(ConfigurationManager.AppSettings["Docu
mentDBName"], ConfigurationManager.AppSettings["DocumentDBCollectionName"]);
IDocumentQuery<dynamic> docQuery = ReadOnlyClient.CreateDocumentQuery(
    collectionUri,
    query,
    new FeedOptions
    {
        MaxItemCount = 10,
        EnableScanInQuery = true
    }
).AsDocumentQuery();
```

Notice in the FeedOptions, we are setting `MaxItemCount = 10`. This means we will get up to 10 results per execution of the query. The DocumentDB API has support for paging built in (We will see an example of this shortly).

Let's quickly inspect the rest of the Query Action:

```
var results = await docQuery.ExecuteNextAsync();
```

This part is what actually uses the Azure DocumentDB SDK to call DocumentDB and retrieve the results for our query. Notice this will only return up to `MaxItemCount` results as above (In our case 10 items). This can also be set to -1 for dynamic sizing of the resulting set to the maximum response size.

If we wanted to get the next set of results we would have to call `docQuery.ExecuteNextAsync()` again.

In the interests of this demo, we are only retrieving the first ten results. However if this was a real-world application where we need ALL of the results for a query. We would set the MaxItemCount to -1 and do something like the following:

```
while (docQuery.HasMoreResults)
{
    //Can use strongly typed objects by using <T> on
docQuery.ExecuteNextAsync<T>()
    var results = await docQuery.ExecuteNextAsync();

    //dynamic can also be T
    foreach (dynamic result in results)
    {
        //Do something with results
    }
}
```

Note: We are deserializing the JSON string and serializing it back again so that we can format the JSON into human readable string.

Press `F5` to compile and launch the web app on the local machine.

Type this query into the query box:

```
SELECT *
FROM c
```

...and click on **Run It!**

Progress! We have successfully returned results from DocumentDB.

Scenario 2

From now on, we will be working directly in the web browser.

In this scenario we will introduce the SQL-like syntax of DocumentDB and show how we can use it to manipulate our results.

The dataset we are querying was extracted from the performance counters on various computers running windows 10.

The performance counters are:

- LogicalDisk
- PhysicalDisk
- HTTP Service
- ASP.NET v4.0.30319
- Power Meter
- Processor
- Energy Meter

Part One

In the query, the `FROM` name is simply an alias to the entire collection for the user to refer to in the query. It is not actually a table like in traditional SQL.

For example:

```
SELECT *
FROM collection
```

...is exactly the same as:

```
SELECT *
FROM logs
```

Give it a try!

When refering to fields you must use the alias you define in the `FROM` clause.

Execute this query:

```
SELECT id
FROM logs
```

As you can see this resulted in an error.

To fix this error we have to provide the full "path" to the properties of the objects within the database.

Execute this query instead:

```
SELECT logs.id
FROM logs
```

Part Two

Now that we know how to select a certain field, we can filter on them.

Write a query to select a specific record by its ID:

```
SELECT *
FROM logs
WHERE logs.id = "ef811f1d-d271-4e1f-baf7-4d5ac0b6c840"
```

Part Three

Let's use a real-world scenario and filter the results to logs for a certain machine.

Execute this query:

```
SELECT *
FROM logs
WHERE logs.machineName = "WKS-1604"
```

The results are limited to logs from the machine `WKS-1604`.

Scenario 3

In this scenario we are going to see how we can use joins to inspect child objects / arrays.

Part One

We have been using DocumentDB to inspect all the logs of a certain machine. What if we only wanted to see logs of a certain type. We can use the `JOIN` keyword to join to our logs array. We can also give it an alias and inspect its properties.

Let's see the `JOIN` in action. Try this query:

```
SELECT perfLog
FROM machinelogs
JOIN perfLog IN machinelogs.logs
```

Inspect the results and you will see for each log object in the array of each document has been returned as a seperate result set:

Now that we know how to join to our child array we can use it for filtering. Lets find all "Power Meter" logs:

```
SELECT perfLog
FROM machinelogs
JOIN perfLog IN machinelogs.logs
WHERE perfLog.counterType = "Power Meter"
```

Because we are refering to objects / documents, we can filter our result set by seeing if a property exists on the object.

For example:

```
SELECT perfLog
FROM machinelogs
JOIN perfLog IN machinelogs.logs
WHERE perfLog.Power != null
```

Imagine that we want to see both LogicalDisk and PhysicalDisk results. There two ways we can achieve this.

Using an `OR` predicate:

```
SELECT perfLog
FROM machinelogs
```

```
JOIN perfLog IN machinelogs.logs
WHERE (perfLog.counterType = "LogicalDisk" OR perfLog.counterType =
"PhysicalDisk")
```

...or using an `IN` predicate:

```
SELECT perfLog
FROM machinelogs
JOIN perfLog IN machinelogs.logs
WHERE perfLog.counterType IN ("LogicalDisk", "PhysicalDisk")
```

We can filter these results by the "% Disk Time" property.Because this property name contains white space, we must use special index to address it:

```
WHERE perfLog["% Disk Time"] ...
```

This syntax will be familiar to users of JavaScript, or C# dictionary accessor syntax.

We can use the `BETWEEN` keyword to filter by a range of values.

Try this query:

```
SELECT perfLog
FROM machinelogs
JOIN perfLog IN machinelogs.logs
WHERE perfLog["% Disk Time"] BETWEEN 42247790840000 AND 42247790860000
```

We have used the `MaxItemCount` in the code to limit our results to 10 items. We can also restrict the amount of results returned by using a `TOP` clause in our query.

Lets adjust our query to find the top result. Give this a try:

```
SELECT TOP 1 perfLog
FROM machinelogs
JOIN perfLog IN machinelogs.logs
WHERE perfLog["% Disk Time"] BETWEEN 42247790840000 AND 42247790860000
```

Results 1 / 1


```
{
  "perfLog": {
    "counterType": "LogicalDisk",
    "counterFor": "C:",
    "% Free Space": 237642,
    "Free Megabytes": 237642,
    "Current Disk Queue Length": 0,
    "% Disk Time": 42247790850000,
    "Avg. Disk Queue Length": 42247790850000,
    "% Disk Read Time": 27919879072000,
    "Avg. Disk Read Queue Length": 27919879072000,
    "% Disk Write Time": 14327911778000,
    "Avg. Disk Write Queue Length": 14327911778000,
    "Avg. Disk sec/Transfer": 9467341249809,
    "Avg. Disk sec/Read": 6256588037147,
    "Avg. Disk sec/Write": 3210753212661,
```


Part Two

We can use a feature called **Projection** to create an entirely new result set. We could use this to create a common structure or to make it match a structure we already have.

Try this query:

```
SELECT {
    "FreeSpacePercent": perfLog["% Free Space"],
    "FreeMegabytes": perfLog["Free Megabytes"],
    "CurrentDiskQueueLength": perfLog["Current Disk Queue Length"],
    "Power" : perfLog["Power"],
    "PowerBudget" : perfLog["Power Budget"]
} AS SummaryLogs
FROM machinelogs
JOIN perfLog IN machinelogs.logs
```

This query allowed us to combine all logs into one well-known structure which could be useful, for example, when binding to a strongly typed dataset.

Further Reading

Get Started with DocumentDB - http://aka.ms/docdbstart

Documentation and Videos - http://aka.ms/docdbdocs

How does pricing work? - http://aka.ms/docdbpricing

Get help on the forums - http://aka.ms/docdbforum

DocumentDB SQL Query Syntax

DocumentDB Query Playground

Appendix

The Azure Portal was used to create the DocumentDB server. The Azure Portal can be found at https://portal.azure.com/.

Some features that you can use in Azure Portal with DocumentDB include:

Document Explorer

View the JSON documents inside your collections.

Query Explorer

Test your queries and view the results.

Script Explorer

View, add and modify stored procedures, user functions and triggers.