Lab 3 Authoring Azure Cosmos DB Stored Procedures using JavaScript

In this lab, you will author and execute multiple stored procedures within your Azure Cosmos DB instance. You will explore features unique to JavaScript stored procedures such as throwing errors for transaction rollback, logging using the JavaScript console and implementing a continuation model within a bounded execution environment.

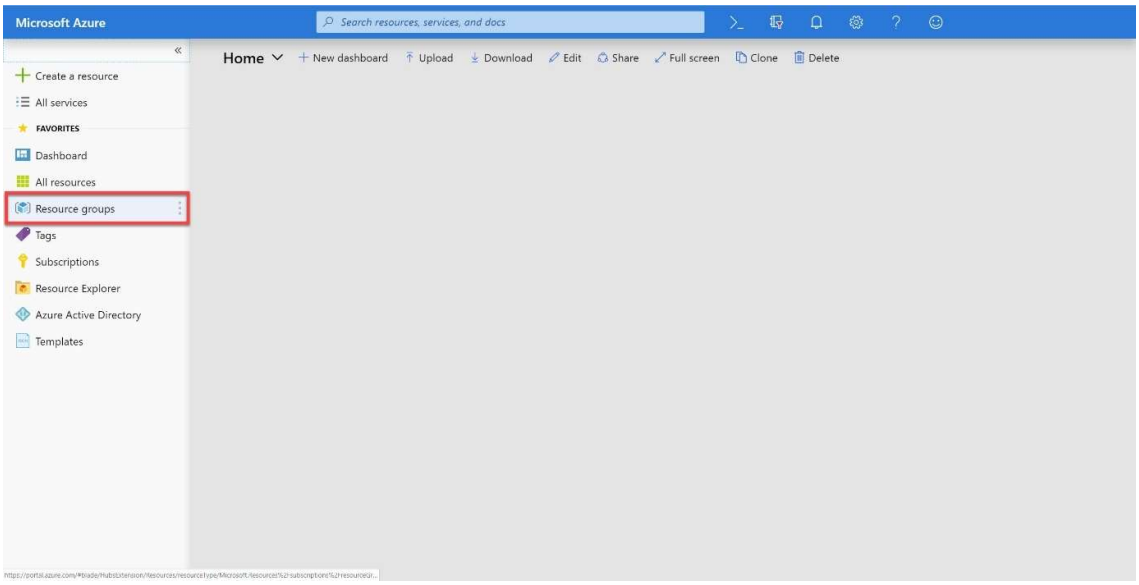
Setup

Before you start this lab, you will need to create an Azure Cosmos DB database and collection that you will use throughout the lab.

Create Azure Cosmos DB Database and Collection

You will now create a database and collection within your Azure Cosmos DB account.

1. On the left side of the portal, click the Resource groups link.



2. In the Resource groups blade, locate and select the cosmosgroup-lab Resource Group.
3. In the cosmosgroup-lab blade, select the Azure Cosmos DB account you recently created.
4. In the Azure Cosmos DB blade, locate and click the Overview link on the left side of the blade.
5. At the top of the Azure Cosmos DB blade, click the Add Collection button.

6. In the Add Collection popup, perform the following actions:
   a. In the Database id field, select the Create new option and enter the value FinancialDatabase.
   b. Ensure the Provision database throughput option is not selected.
   c. In the Collection id field, enter the value InvestorCollection.
   d. In the Storage capacity section, select the Unlimited option.
   e. In the Partition key field, enter the value `/company`.
   f. In the Throughput field, enter the value `1000`.
   g. Click the **OK** button.
7. Wait for the creation of the new database and collection to finish before moving on with this lab.

Author Simple Stored Procedures

You will get started in this lab by authoring simple stored procedures that implement common server-side tasks such as adding one or more documents as part of a database transaction.

Open Data Explorer

1. In the Azure Cosmos DB blade, locate and click the Data Explorer link on the left side of the blade.
2. In the Data Explorer section, expand the FinancialDatabase database node and then expand the InvestorCollection collection node.
3. Within the InvestorCollection node, click the Documents link.

Create Simple Stored Procedure

1. Click the New Stored Procedure button at the top of the Data Explorer section.
2. In the stored procedure tab, locate the Stored Procedure Id field and enter the value: greetCaller.
3. Replace the contents of the stored procedure editor with the following JavaScript code:

```
function greetCaller(name) {
     var context = getContext();
     var response = context.getResponse();
     response.setBody("Hello " + name);
 }
```

This simple stored procedure will echo the input parameter string with the text `Hello` as a prefix.

4. Click the Save button at the top of the tab.
5. Click the Execute button at the top of the tab.
6. In the Input parameters popup that appears, perform the following actions:
    a. In the Partition key value field, enter the value: `example`.
    b. Click the Add New Param button.
    c. In the new field that appears, enter the value: `Person`.
    d. Click the Execute button.
7. In the Result pane at the bottom of the tab, observe the results of the stored procedure's execution.

> The output should be `"Hello Person"`.

Create Stored Procedure with Nested Callbacks

All Azure Cosmos DB operations within a stored procedure are asynchronous and depend on JavaScript function callbacks. A **callback function** is a JavaScript function that is used as a parameter to another JavaScript function. In the context of Azure Cosmos DB, the callback function has two parameters, one for the error object in case the operation fails, and one for the created object.

1. Click the New Stored Procedure button at the top of the Data Explorer section.
2. In the stored procedure tab, locate the Stored Procedure Id field and enter the value: createDocument.
3. Replace the contents of the stored procedure editor with the following JavaScript code:

```
function createDocument(doc) {
     var context = getContext();
     var collection = context.getCollection();
     var accepted = collection.createDocument(
         collection.getSelfLink(),
         doc,
         function (err, newDoc) {
             if (err) throw new Error('Error' + err.message);
             context.getResponse().setBody(newDoc);
         }
     );
     if (!accepted) return;
}
```

> Inside the JavaScript callback, users can either handle the exception or throw an error. In case a callback is not provided and there is an error, the Azure Cosmos DB runtime throws an error. This stored procedure creates a new document and uses a nested callback function to return the document as the body of the response.

4. Click the Save button at the top of the tab.
5. Click the Execute button at the top of the tab.
6. In the Input parameters popup that appears, perform the following actions:

a. In the Partition key value field, enter the value: `contosoairlines`.
b. Click the Add New Param button.
c. In the new field that appears,
   enter the value: `{ "company": "contosoairlines", "industry": "travel" }`.
d. Click the Execute button.

7. In the Result pane at the bottom of the tab, observe the results of the stored procedure's execution.

> You should see a new document in your collection. Azure Cosmos DB has assigned additional fields to the document such as `id` and `_etag`.

8. Click the New SQL Query button at the top of the Data Explorer section.
9. In the query tab, replace the contents of the query editor with the following SQL query:

```
SELECT * FROM investors WHERE investors.company = "contosoairlines" AND
investors.industry = "travel"
```

> This query will retrieve the document you have just created.

10. Click the Execute Query button in the query tab to run the query.
11. In the Results pane, observe the results of your query.
12. Close the Query tab.

Create Stored Procedure with Logging

1. Click the New Stored Procedure button at the top of the Data Explorer section.
2. In the stored procedure tab, locate the Stored Procedure Id field and enter the value: createDocumentWithLogging.
3. Replace the contents of the stored procedure editor with the following JavaScript code:

```
function createDocumentWithLogging(doc) {
     console.log("procedural-start ");
     var context = getContext();
     var collection = context.getCollection();
     console.log("metadata-retrieved ");
     var accepted = collection.createDocument(
         collection.getSelfLink(),
         doc,
         function (err, newDoc) {
             console.log("callback-started ");
             if (err) throw new Error('Error' + err.message);
             context.getResponse().setBody(newDoc.id);
         }
     );
     console.log("async-doc-creation-started ");
     if (!accepted) return;
     console.log("procedural-end");
 }
```

> This stored procedure will use the console.log feature that's normally used in browser-based JavaScript to write output to the console. In the context of Azure Cosmos DB, this feature can be used to capture diagnostics logging information that can be returned after the stored procedure is executed.

4. Click the Save button at the top of the tab.
5. Click the Execute button at the top of the tab.
6. In the Input parameters popup that appears, perform the following actions:
    a. In the Partition key value field, enter the value: `contosoairlines`.
    b. Click the Add New Param button.
    c. In the new field that appears, enter the value: `{ "company": "contosoairlines" }`.
    d. Click the Execute button.
7. In the Result pane at the bottom of the tab, observe the results of the stored procedure's execution.

> You should see the unique id of a new document in your collection.

8. Click the console.log link in the Result pane to view the log data for your stored procedure execution.

> You can see that the procedural components of the stored procedure finished first and then the callback function was executed once the document was created. This can help you understand the asynchronous nature of JavaScript callbacks.

Create Stored Procedure with Callback Functions

1. Click the New Stored Procedure button at the top of the Data Explorer section.
2. In the stored procedure tab, locate the Stored Procedure Id field and enter the value: createDocumentWithFunction.
3. Replace the contents of the stored procedure editor with the following JavaScript code:

```
function createDocumentWithFunction(document) {
     var context = getContext();
     var collection = context.getCollection();
     if (!collection.createDocument(collection.getSelfLink(), document, documentCreated))
         return;
     function documentCreated(error, newDocument) {
         if (error) throw new Error('Error' + error.message);
         context.getResponse().setBody(newDocument);
     }
 }
```

> This is the same stored procedure as you created previously but it is using a named function instead of an implicit callback function inline.

4. Click the Save button at the top of the tab.
5. Click the Execute button at the top of the tab.
6. In the Input parameters popup that appears, perform the following actions:
   a. In the Partition key value field, enter the value: `adventureworks`.
   b. Click the Add New Param button.
   c. In the new field that appears, enter the value: `{ "company": "contosoairlines" }`.
   d. Click the Execute button.
7. In the Result pane at the bottom of the tab, observe that the stored procedure execution has failed.

> Stored procedures are bound to a specific partition key. In this example, tried to execute the stored procedure within the context of the adventureworks partition key. Within the stored procedure, we tried to create a new document using the contosoairlines partition key. The stored procedure was unable to create a new document (or access existing documents) in a partition key other than the one specified when the stored procedure is executed. This caused the stored procedure to fail. You are not able to create or manipulate documents across partition keys within a stored procedure.

8. Click the Execute button at the top of the tab.
9. In the Input parameters popup that appears, perform the following actions:

    a) Click the Add New Param button.
    b) In the Partition key value field, enter the value: `adventureworks`
    c) In the new field that appears, enter the value: `{ "company": "adventureworks" }`.
    d) Click the Execute button.

10. In the Result pane at the bottom of the tab, observe the results of the stored procedure's execution.

    You should see a new document in your collection. Azure Cosmos DB has assigned additional fields to the document such as `id` and `_etag`.

11. Click the New SQL Query button at the top of the Data Explorer section.
12. In the query tab, replace the contents of the query editor with the following SQL query:

```
SELECT * FROM investors WHERE investors.company = "adventureworks"
```

    This query will retrieve the document you have just created.

13. Click the Execute Query button in the query tab to run the query.
14. In the Results pane, observe the results of your query.
15. Close the Query tab.

Create Stored Procedure with Error Handling

1. Click the New Stored Procedure button at the top of the Data Explorer section.
2. In the stored procedure tab, locate the Stored Procedure Id field and enter the value: createTwoDocuments.
3. Replace the contents of the stored procedure editor with the following JavaScript code:

```
function createTwoDocuments(companyName, industry, taxRate) {
     var context = getContext();
     var collection = context.getCollection();
     var firstDocument = {
         company: companyName,
         industry: industry
     };
     var secondDocument = {
         company: companyName,
         tax: {
             exempt: false,
             rate: taxRate
         }
     };
     var firstAccepted = collection.createDocument(collection.getSelfLink(), firstDocument,
         function (firstError, newFirstDocument) {
             if (firstError) throw new Error('Error' + firstError.message);
             var secondAccepted = collection.createDocument(collection.getSelfLink(),
secondDocument,
                 function (secondError, newSecondDocument) {
                     if (secondError) throw new Error('Error' + secondError.message);
                     context.getResponse().setBody({
                         companyRecord: newFirstDocument,
                         taxRecord: newSecondDocument
                     });
                 }
             );
             if (!secondAccepted) return;
         }
     );
     if (!firstAccepted) return;
 }
```

This stored procedure uses nested callbacks to create two seperate documents. You may have scenarios where your data is split across multiple JSON documents and you will need to add or modify multiple documents in a single stored procedure.

4. Click the Save button at the top of the tab.
5. Click the Execute button at the top of the tab.
6. In the Input parameters popup that appears, perform the following actions:
    a. In the Partition key value field, enter the value: `abcairways`.
    b. Click the Add New Param button three times.
    c. In the first field that appears, enter the value: `abcairways`.
    d. In the second field that appears, enter the value: `travel`.
    e. In the third field that appears, enter the value: `1.05`.
    f. Click the Execute button.

7. In the Result pane at the bottom of the tab, observe the results of the stored procedure's execution.

> You should see a new document in your collection. Azure Cosmos DB has assigned additional fields to the document such as `id` and `_etag`.

8. Replace the contents of the stored procedure editor with the following JavaScript code:

```
function createTwoDocuments(companyName, industry, taxRate) {
    var context = getContext();
    var collection = context.getCollection();
    var firstDocument = {
        company: companyName,
        industry: industry
    };
    var secondDocument = {
        company: companyName + "_taxprofile",
        tax: {
            exempt: false,
            rate: taxRate
        }
    };
    var firstAccepted = collection.createDocument(collection.getSelfLink(), firstDocument,
        function (firstError, newFirstDocument) {
            if (firstError) throw new Error('Error' + firstError.message);
            console.log('Created: ' + newFirstDocument.id);
            var secondAccepted = collection.createDocument(collection.getSelfLink(), secondDocument,
                function (secondError, newSecondDocument) {
                    if (secondError) throw new Error('Error' + secondError.message);
                    console.log('Created: ' + newSecondDocument.id);
                    context.getResponse().setBody({
                        companyRecord: newFirstDocument,
                        taxRecord: newSecondDocument
                    });
                }
            );
            if (!secondAccepted) return;
        }
    );
    if (!firstAccepted) return;
}
```

> Transactions are deeply and natively integrated into Cosmos DB's JavaScript programming model. Inside a JavaScript function, all operations are automatically wrapped under a single transaction. If the JavaScript completes without any exception, the operations to the database are committed. We are going to change the stored procedure to put in a different company name for the second document. This should cause the stored procedure to fail since the second document uses a different partition key. If there is any exception that's propagated from the script, Cosmos DB's JavaScript runtime will roll back the whole transaction. This will effectively ensure that the first or second documents are not commited to the database.

9. Click the Update button at the top of the tab.
10. Click the Execute button at the top of the tab.

11. In the Input parameters popup that appears, perform the following actions:

    a) In the Partition key value field, enter the value: jetsonairways.
    b) In the Partition key value field, enter the value: jetsonairways
    c) Click the Add New Param button three times.
    d) In the first field that appears, enter the value: jetsonairways
    e) In the second field that appears, enter the value: travel.
    f) In the third field that appears, enter the value: 1.15.
    g) Click the Execute button.

12. In the Result pane at the bottom of the tab, observe that the stored procedure execution has failed.

    This stored procedure failed to create the second document so the entire transaction was rolled back.

13. Click the New SQL Query button at the top of the Data Explorer section.
14. In the query tab, replace the contents of the query editor with the following SQL query:

```
SELECT * FROM investors WHERE investors.company = "jetsonairways"
```

    This query won't retrieve any documents since the transaction was rolled back.

15. Click the Execute Query button in the query tab to run the query.
16. In the Results pane, observe the results of your query.
17. Close the Query tab.

Author Stored Procedures using the Continuation Model

You will now implement stored procedures that may execute longer than the bounded execution limit time on the server. You will implement the continuation model so that the stored procedures can "pick up where they left off" after they ran out of time in a previous execution.

Create Bulk Upload and Bulk Delete Stored Procedures

1. Click the New Stored Procedure button at the top of the Data Explorer section.
2. In the stored procedure tab, locate the Stored Procedure Id field and enter the value: bulkUpload.
3. Replace the contents of the *stored procedure editor* with the following JavaScript code:

```
function bulkUpload(docs) {
     var collection = getContext().getCollection();
     var collectionLink = collection.getSelfLink();
     var count = 0;
     if (!docs) throw new Error("The array is undefined or null.");
     // additional line needed as parameters come in as strings not arrays!
     if (typeof docs === "string") docs = JSON.parse(docs)
     var docsLength = docs.length;
     if (docsLength == 0) {
         getContext().getResponse().setBody(0);
         return;
     }
     tryCreate(docs[count], callback);
     function tryCreate(doc, callback) {
         var isAccepted = collection.createDocument(collectionLink, doc, callback);
         if (!isAccepted) getContext().getResponse().setBody(count);
     }
     function callback(err, doc, options) {
         if (err) throw err;
         count++;
         if (count >= docsLength) {
             getContext().getResponse().setBody(count);
         } else {
             tryCreate(docs[count], callback);
         }
     }
 }
```

This stored procedure uploads an array of documents in one batch. If the entire batch is not completed, the stored procedure will set the response body to the number of documents that were imported. Your client-side code is expected to call this stored procedure multiple times until all documents are imported.

If you are having trouble copying the stored procedure above, the full source code for this stored procedure is located here:

https://cosmosdb.github.io/labs/solutions/05-authoring_stored_procedures/bulk_upload.js

4. Click the Save button at the top of the tab.

5. Click the New Stored Procedure button at the top of the Data Explorer section.
6. In the stored procedure tab, locate the Stored Procedure Id field and enter the value: bulkDelete.
7. Replace the contents of the stored procedure editor with the following JavaScript code:

```
function bulkDelete(query) {
    var collection = getContext().getCollection();
    var collectionLink = collection.getSelfLink();
    var response = getContext().getResponse();
    var responseBody = {
        deleted: 0,
        continuation: true
    };
    if (!query) throw new Error("The query is undefined or null.");
    tryQueryAndDelete();
    function tryQueryAndDelete(continuation) {
        var requestOptions = {continuation: continuation};
        var isAccepted = collection.queryDocuments(collectionLink, query, requestOptions,
function (err, retrievedDocs, responseOptions) {
            if (err) throw err;
            if (retrievedDocs.length > 0) {
                tryDelete(retrievedDocs);
            } else if (responseOptions.continuation) {
                tryQueryAndDelete(responseOptions.continuation);
            } else {
                responseBody.continuation = false;
                response.setBody(responseBody);
            }
        });
        if (!isAccepted) {
            response.setBody(responseBody);
        }
    }
    function tryDelete(documents) {
        if (documents.length > 0) {
            var isAccepted = collection.deleteDocument(documents[0]._self, {}, function
(err, responseOptions) {
                if (err) throw err;
                responseBody.deleted++;
                documents.shift();
                tryDelete(documents);
            });
            if (!isAccepted) {
                response.setBody(responseBody);
            }
        } else {
            tryQueryAndDelete();
        }
    }
}
```

This stored procedure iterates through all documents that match a specific query and deletes the documents. If the stored procedure is unable to delete all documents, it will return a continuation token. Your client-side code is expected to repeatedly call the stored procedure passing in a continuation token until the stored procedure does not return a continuation token.

If you are having trouble copying the stored procedure above, the full source code for this stored procedure is located here:

https://cosmosdb.github.io/labs/solutions/05-authoring_stored_procedures/bulk_delete.js

8. Click the Save button at the top of the tab.

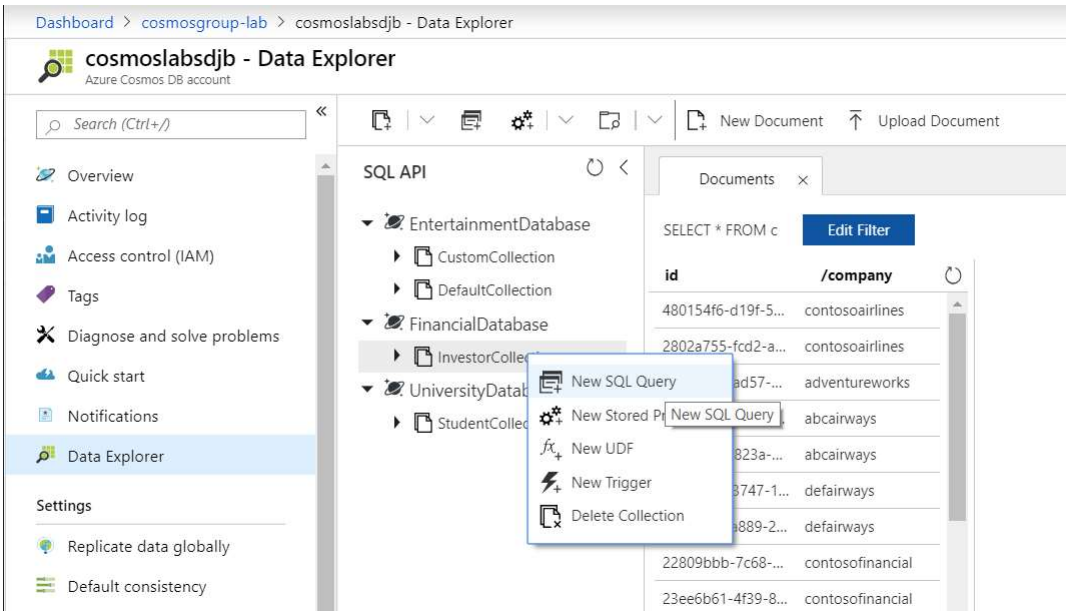
Calling the bulkUpload and bulkDelete Stored Procedures using the Python API

In this section we will create a small program to build up an array of 10,000 documents to insert into the investorCollection. We will then call the bulkUpload stored procedure from Python on order to load as many documents as is possible until the maximum permitted run duration is reached by the bulkUpload stored preocedure. When this happens, the procedure will return the number of documents inserted to the calling python code (in the nDocsCreated variable). This allows the calling program to make an additional call to the stored procedure with the remaining documents. Notice the Python API does not support the passing of arrays to Cosmos DB stored procedures. In the bulkUploadProcCall.py program we convert the array of documents to load into a string representation of a JSON array of the documents and pass this as a string parameter.

```
import azure.cosmos as cosmos
import azure.cosmos.cosmos_client as cosmos_client
import json
from faker import Faker

if __name__ == "__main__":

    # get the access key and endpoint and set up the client
    with open('config.json', 'r') as f:
        config = json.load(f)

    # create a client and set up the database and collection links
    client = cosmos_client.CosmosClient(config['endPoint'],
                                {'masterKey':    config['key']})

    createdDocs = 10000
    databaseID = "FinancialDatabase";
    collectionID = "InvestorCollection"
    storedProcID = "bulkUpload"
    database_link = 'dbs/' + databaseID
    collection_link = database_link + '/colls/' + collectionID
    storedProc_link = collection_link + '/sprocs/' + storedProcID
    options = { "partitionKey": "contosofinancial" }

    # create some random documents with the faker module
    myFactory = Faker()
    newDocArr = []
    for cnt in range(createdDocs):
        newDoc = { "firstName": myFactory.first_name() ,
                   "lastName" : myFactory.last_name() ,
                   "company"  : "contosofinancial"
                  }
        newDocArr.append(newDoc)

    while len(newDocArr) > 0:
        # render the json array of docs as a string we can't pass an array
        # to the stored procedure
        ndaStr = json.dumps(newDocArr)
        nDocsCreated = client.ExecuteStoredProcedure(storedProc_link, ndaStr , options)
        print("Just inserted {} documents".format(nDocsCreated))
        newDocArr = newDocArr[nDocsCreated :]

    print("done!")
```

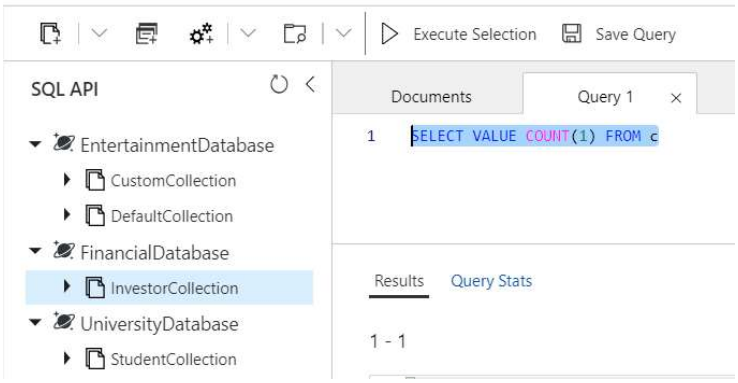
bulkUploadProcCall.py

1. Create a new file in Visual Studio Code with the name bulkUploadProcCall.py and copy the code from above into the file.
2. Notice how we build up the specifier name storedProc_link for the stored procedure call for the collection link. In this case the value for storedProc_link will be **'dbs/FinancialDatabase/colls/InvestorCollection/sprocs/bulkUpload'**
   This illustrates the resource and naming model used internally in Cosmos DB. The stored procedure link is used in the call to the stored procedure call client.ExecuteStoredProcedure() later in the program.
3. Check how many documents are present in the InvestorCollection collection by executing a new query in the Azure Portal Cosmos DB Data Explorer panel. Select New Query in the Data Explorer



4. Type in the following query into the query tab, and execute the query and make a note of the number of documents present

```
SELECT VALUE COUNT(1) FROM c
```



5. Execute the python program by pressing the F5 button. Notice how the documents are inserted in batches of 1500 – 2000 documents from the program command line.

6. Return to the Azure Portal Cosmos DB Data Explorer query panel and re-run the query. Confirm that there are 10,000 more documents in the InvestorCollection.

   Use the following query to count the newly inserted documents

```
SELECT VALUE COUNT(1) FROM c WHERE c.company = "contosofinancial"
```

Delete Documents using the Bulk Delete Stored Procedure using the Python API

In this final section we will create a python program to call the bulkDelete stored procedure repeatedly until all of the newly inserted documents have been deleted. We will use the response document from the stored procedure bulkDelete to check whether there are any more documents to delete and to see how many have been deleted in each call. Once again the stored procedure will execute within the database until its time allocation is exhausted and this will not be enough to delete all of the documents so repeated calls will be required.

1. Create a new file in Visual Studio Code with the name bulkDeleteCall.py and copy the code from below into the file.

```
import azure.cosmos as cosmos
import azure.cosmos.cosmos_client as cosmos_client
import json

if __name__ == "__main__":

    # get the access key and endpoint and set up the client
    with open('config.json', 'r') as f:
        config = json.load(f)

    # create a client and set up the database and collection links
    client = cosmos_client.CosmosClient(config['endPoint'],
                                {'masterKey':    config['key']})

    databaseID = "FinancialDatabase";
    collectionID = "InvestorCollection"
    storedProcID = "bulkDelete"
    database_link = 'dbs/' + databaseID
    collection_link = database_link + '/colls/' + collectionID
    storedProc_link = collection_link + '/sprocs/' + storedProcID

    docsDeleted = {'continuation' : True, 'deleted' : 0 }

    options = {"partitionKey": "contosofinancial"}
    queryStr = "SELECT * FROM investors i WHERE i.company = 'contosofinancial'"
    while docsDeleted['continuation']:
        docsDeleted = client.ExecuteStoredProcedure(storedProc_link, queryStr , options)
        print("Number of documents deleted = {}".format(docsDeleted['deleted']) )

    print("done!")
```

2. Set a breakpoint at the `print("Number of documents deleted…")` line and inspect the contents of the docsDeleted variable – notice it is a dictionary with the number of deleted documents and a continuation flag.
3. Let the program run to completion and verify that the number of deleted documents was 10,000
4. Verify that all of the documents that were inserted by the bulkUpload procedure have been deleted using the following query in the Azure Portal Cosmos DB Data Explorer. (The query should return 0 documents)

```
SELECT VALUE COUNT(1) FROM c WHERE c.company = "contosofinancial"
```

Finally clear up the database and collections by deleting the FinancialDatabase in the Azure Portal by right clicking on it in the Data Explorer and selecting Delete Database to avoid incurring unnecessary costs