Querying an Azure Cosmos DB Database using the SQL API

In this lab, you will query an Azure Cosmos DB database instance using the SQL language. You will use features common in SQL such as projection using SELECT statements and filtering using WHERE clauses. You will also get to use features unique to Azure Cosmos DB's SQL API such as projection into JSON, intra-document JOIN and filtering to a range of partition keys.

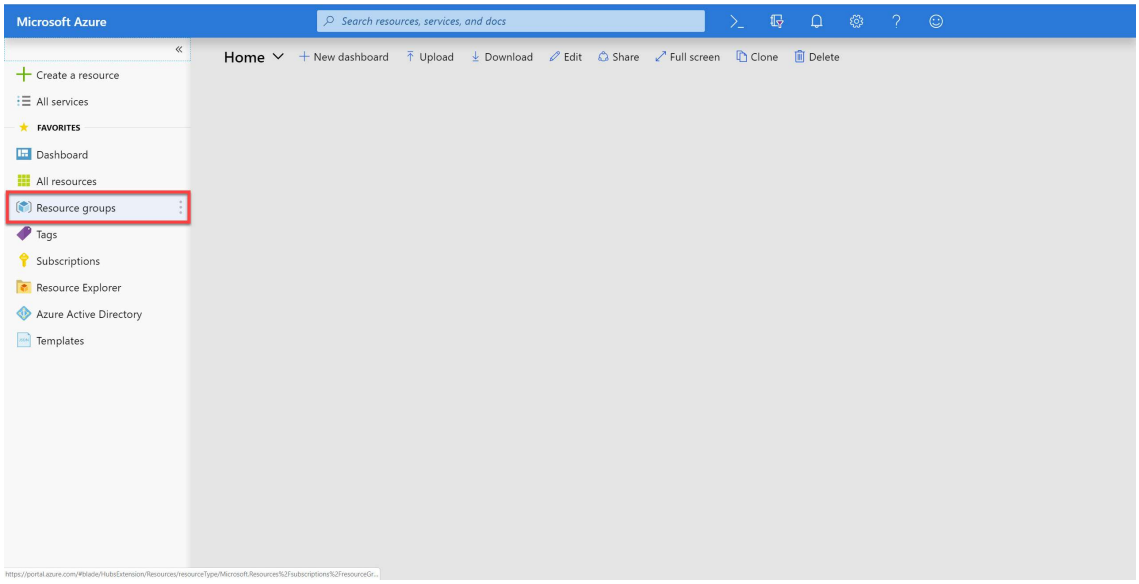
Setup

Before you start this lab, you will need to create an Azure Cosmos DB database and collection that you will use throughout the lab. You will also use the **Azure Data Factory (ADF)** to import existing data into your collection.

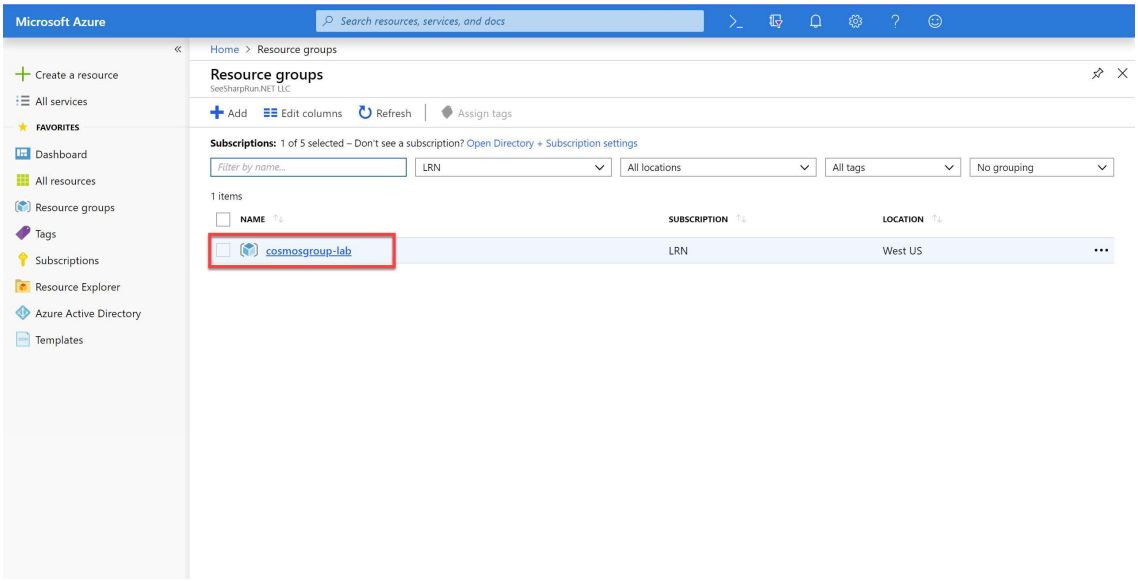
Create Azure Cosmos DB Database and Collection

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

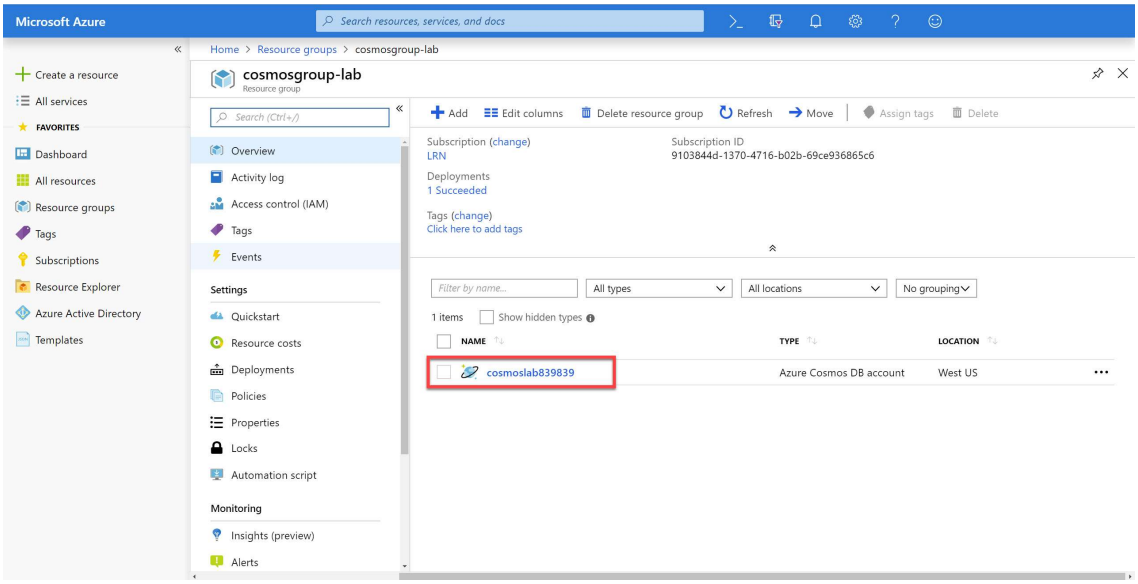
1. On the left side of the portal, click the **Resource groups** link.



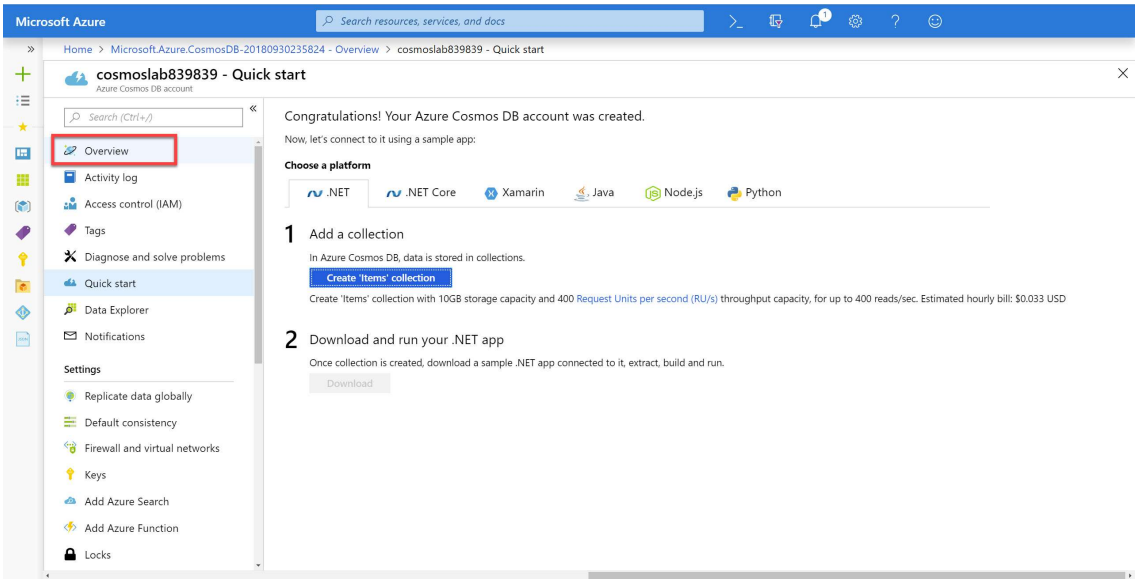
2. In the **Resource groups** blade, locate and select the **cosmosgroup-lab** *Resource Group*.



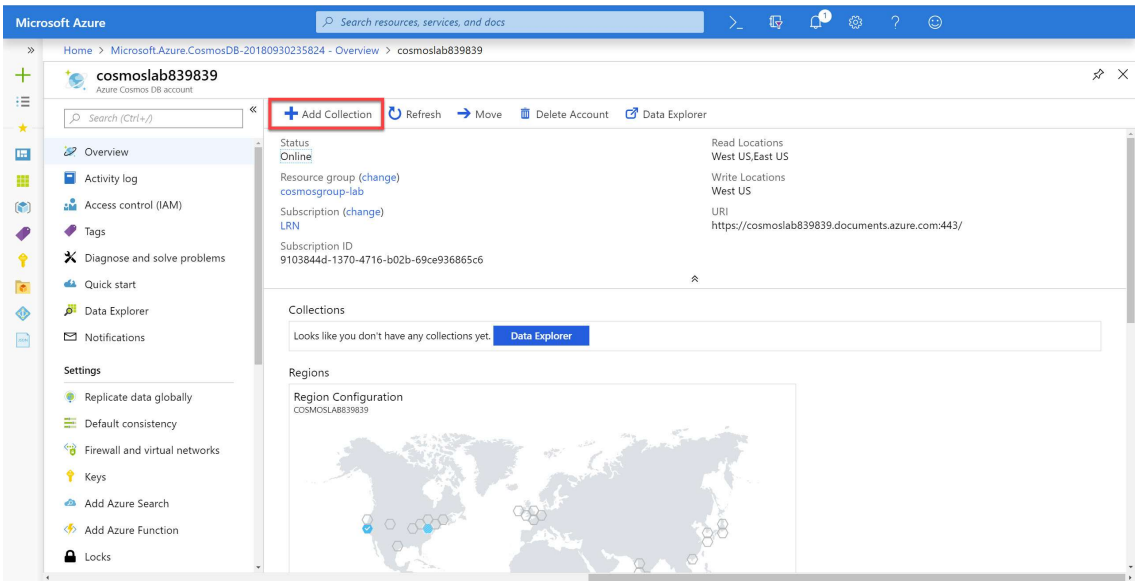
3. In the **cosmosgroup-lab** blade, select the **Azure Cosmos DB** account you recently created.



4. In the **Azure Cosmos DB** blade, locate and click the **Overview** link on the left side of the blade.



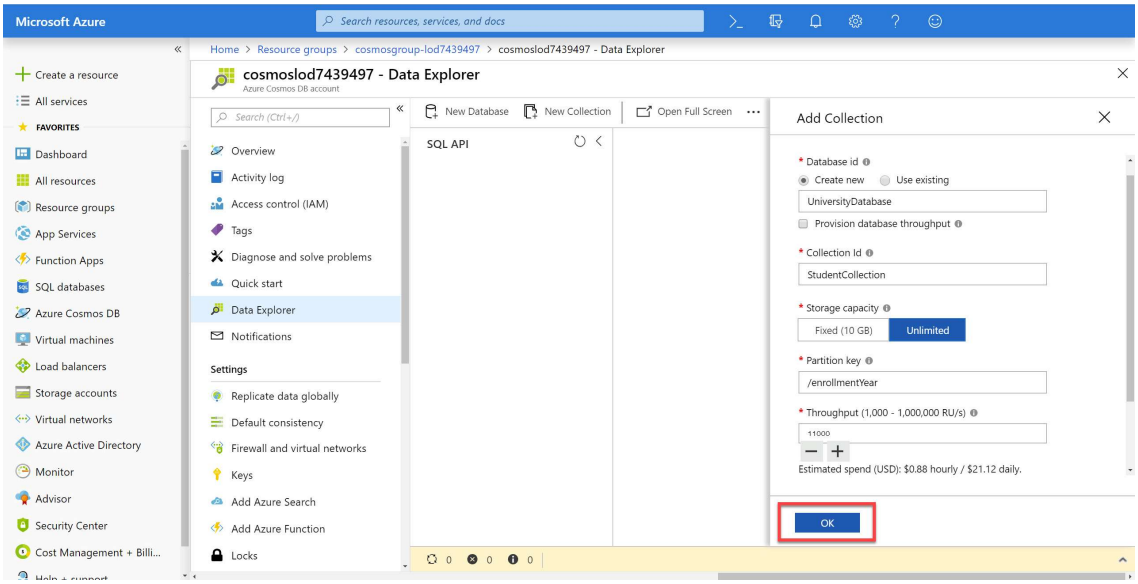
5. At the top of the **Azure Cosmos DB** blade, click the **Add Collection** button.



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 **UniversityDatabase**.
    b. Ensure the **Provision database throughput** option is not selected.

Provisioning throughput for a database allows you to share the throughput among all the containers that belong to that database. Within an Azure Cosmos DB database, you can have a set of containers which shares the throughput as well as containers, which have dedicated throughput.

c. In the **Collection Id** field, enter the value **StudentCollection**.
d. In the **Partition key** field, enter the value `/enrollmentYear`.
e. In the **Throughput** field, enter the value `11000`.
f. Click the **+ Add Unique Key** link.
g. In the new **Unique Keys** field, enter the value `/studentAlias`.
h. Click the **OK** button.



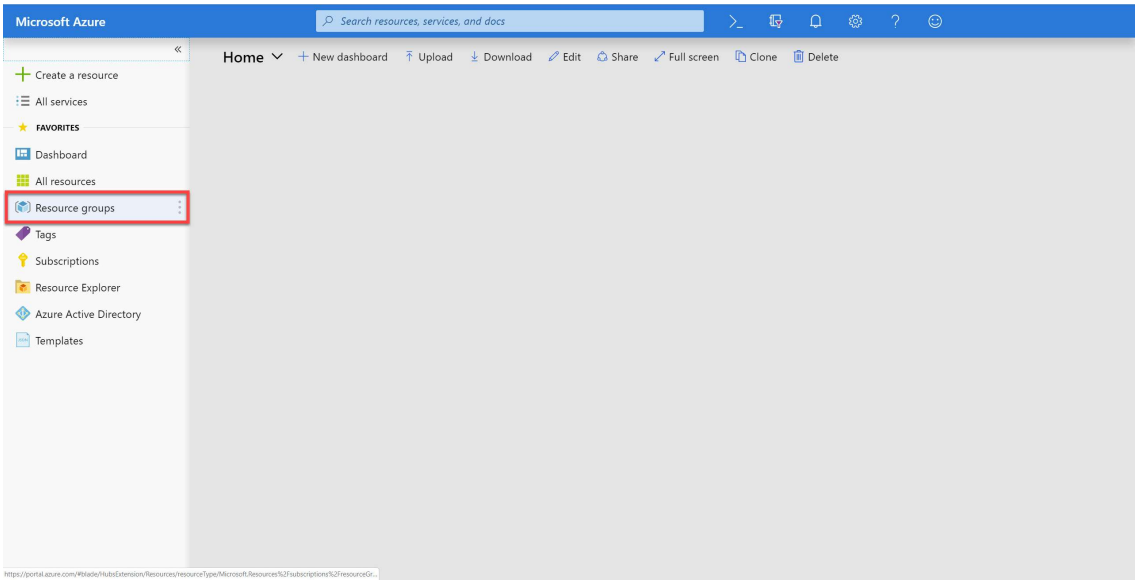
7. Wait until the database and the collection have finished creating before proceeding.

Import Lab Data Into Collection

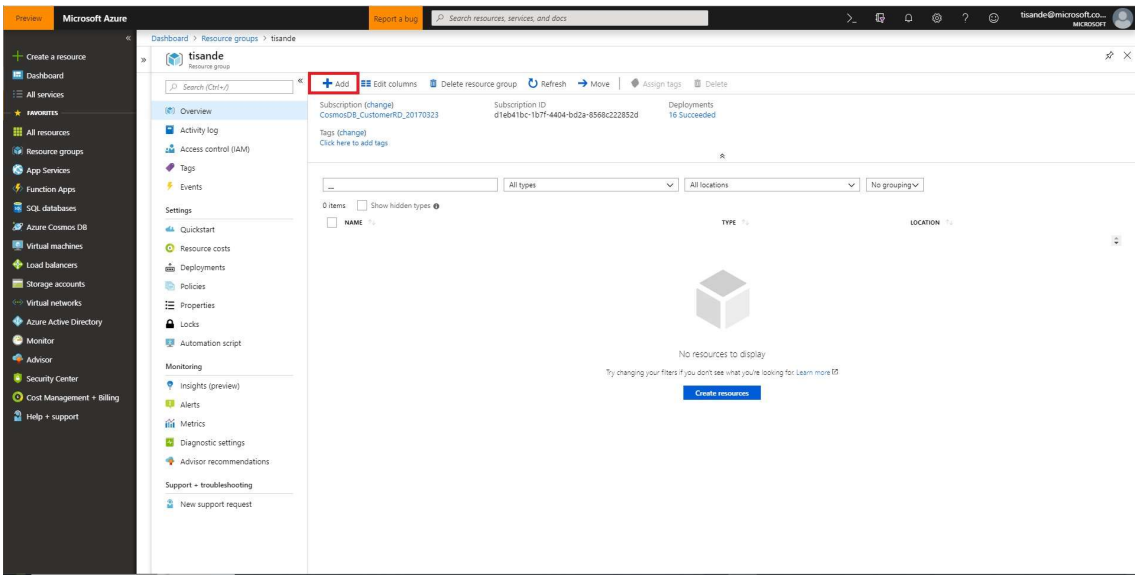
You will use **Azure Data Factory (ADF)** to import the JSON array stored in the **students.json** file from Azure Blob Storage.

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

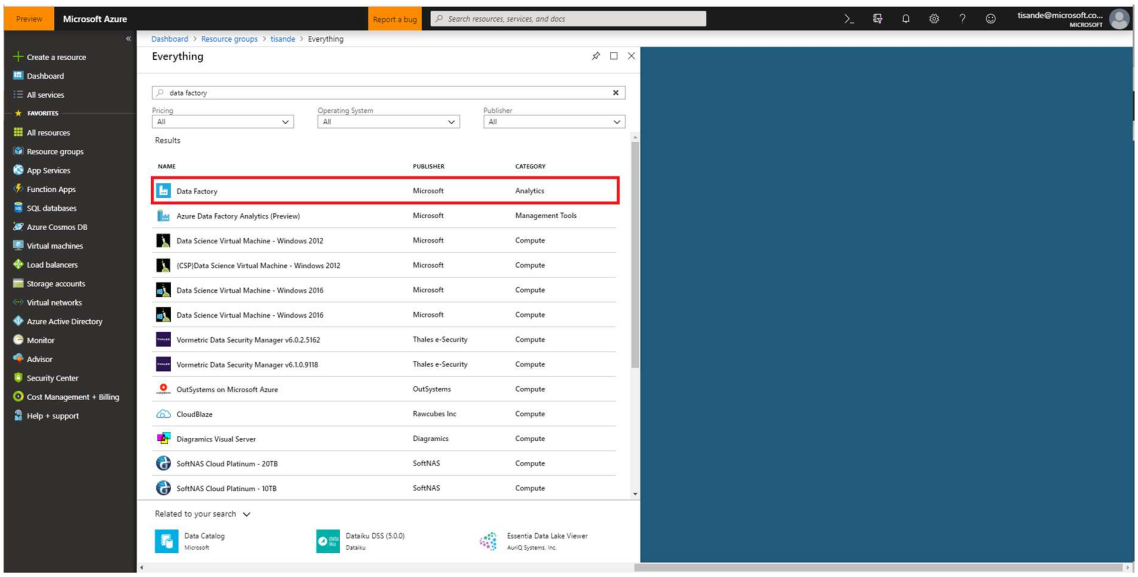
> To learn more about copying data to Cosmos DB with ADF, please read ADF's documentation.



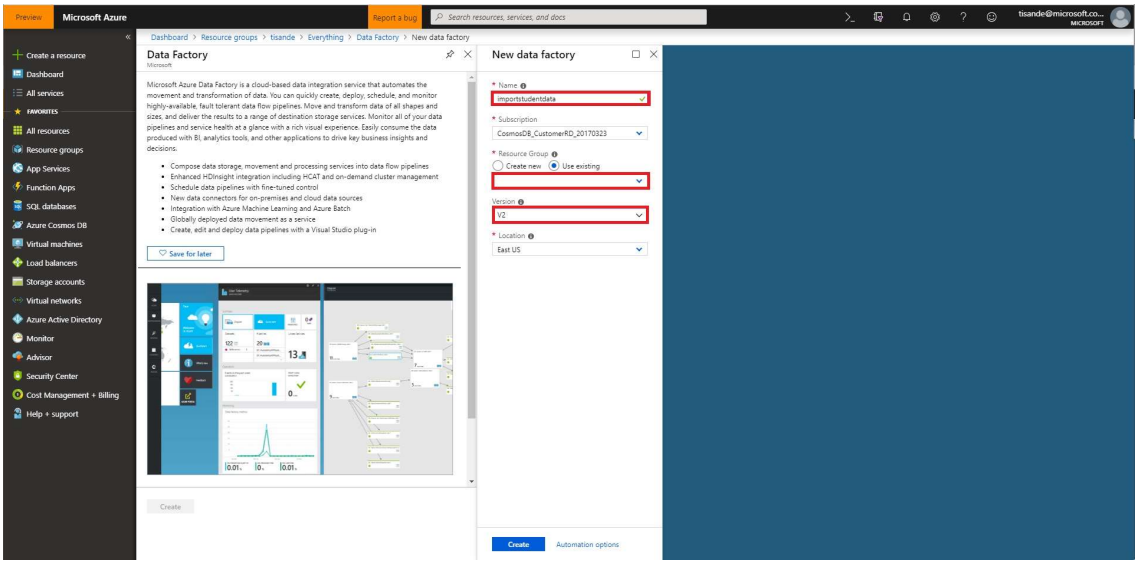
2. In the Resource groups blade, locate and select the cosmosgroup-lab Resource Group.
3. Click add to add a new resource



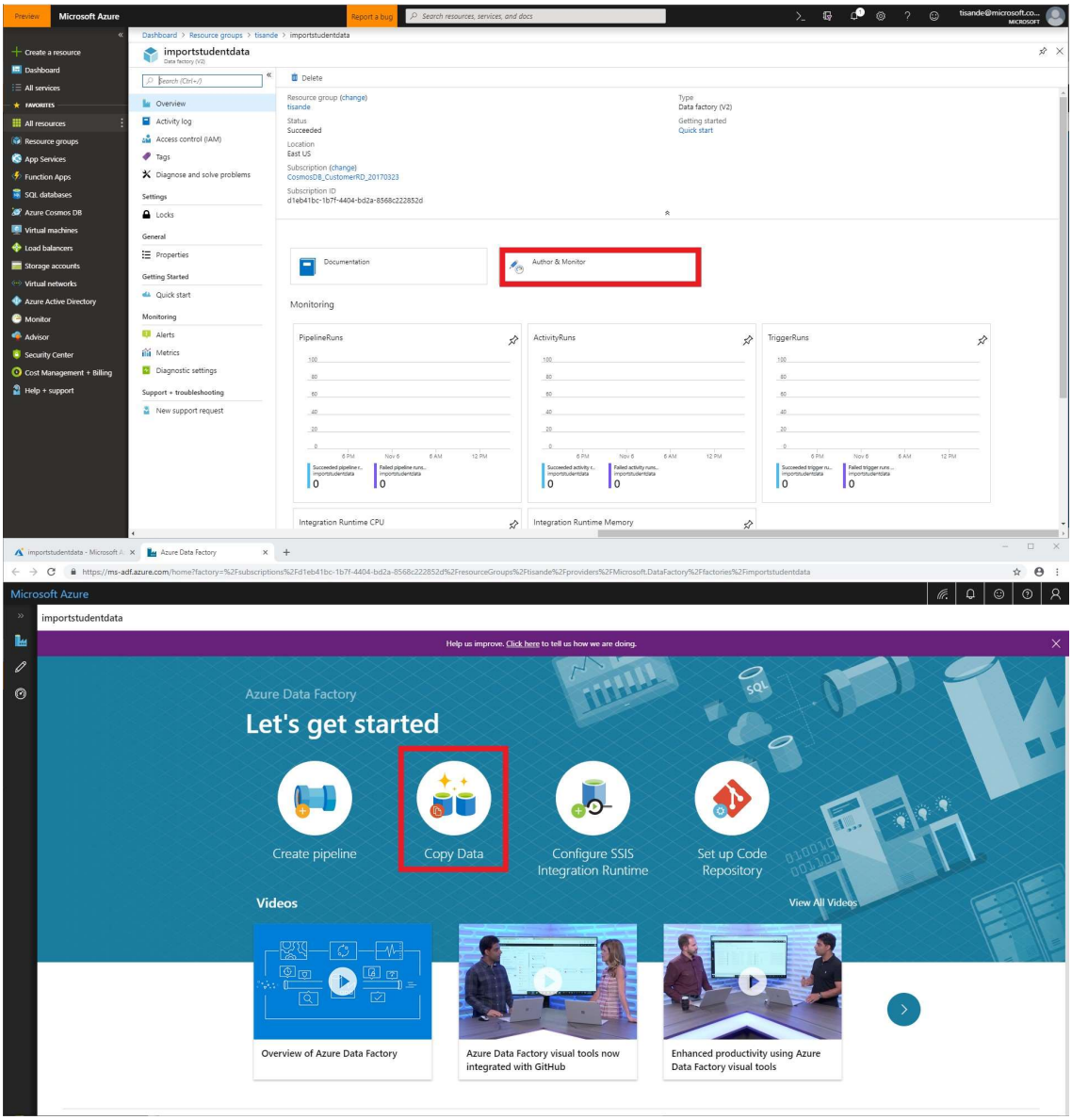
4. Search for Data Factory and select it



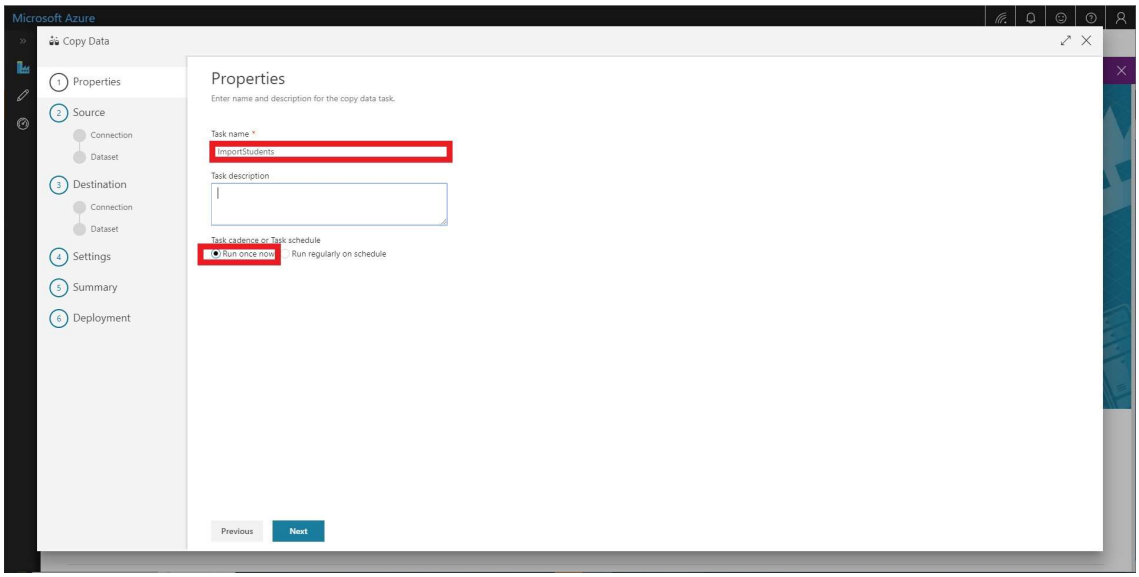
5. Create a new Data Factory. You should name this data factory importstudentdataXX (where XX are your initials) and select the relevant Azure subscription. You should ensure your existing cosmosdblab-group resource group is selected as well as a Version V2. Select North Europe as the region. Don't worry about configuring git. Click create.



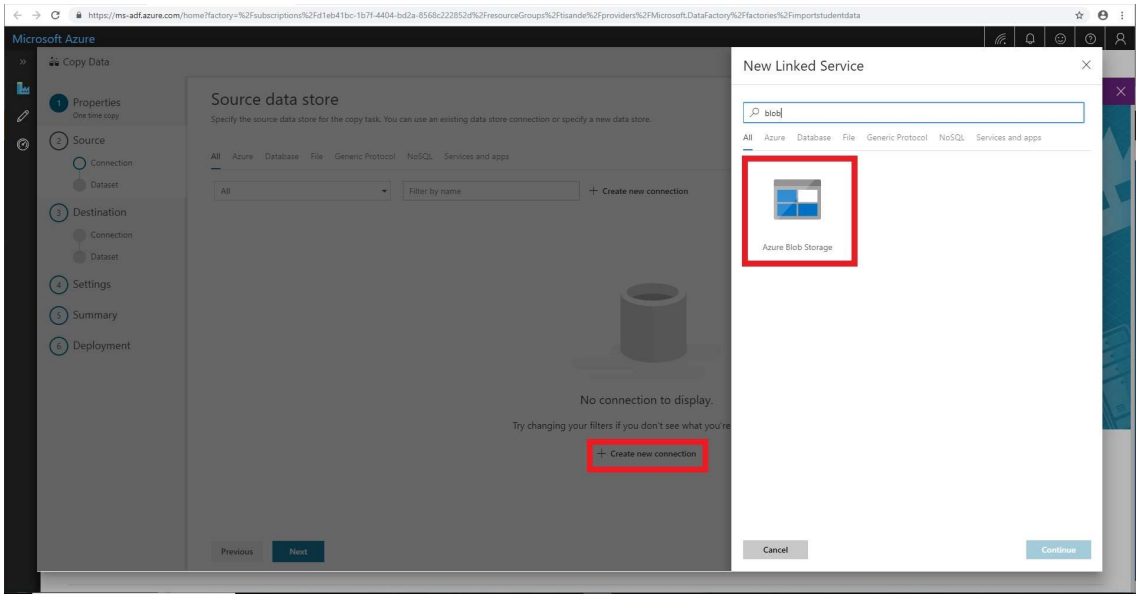
6. After creation, open your newly created Data Factory. Select Author & Monitor and you will launch ADF. You should see a screen similar to the screenshot below. Select Copy Data. We will be using ADF for a one-time copy of data from a source JSON file on Azure Blob Storage to a database in Cosmos DB's SQL API. ADF can also be used for more frequent data transfers from Cosmos DB to other data stores.



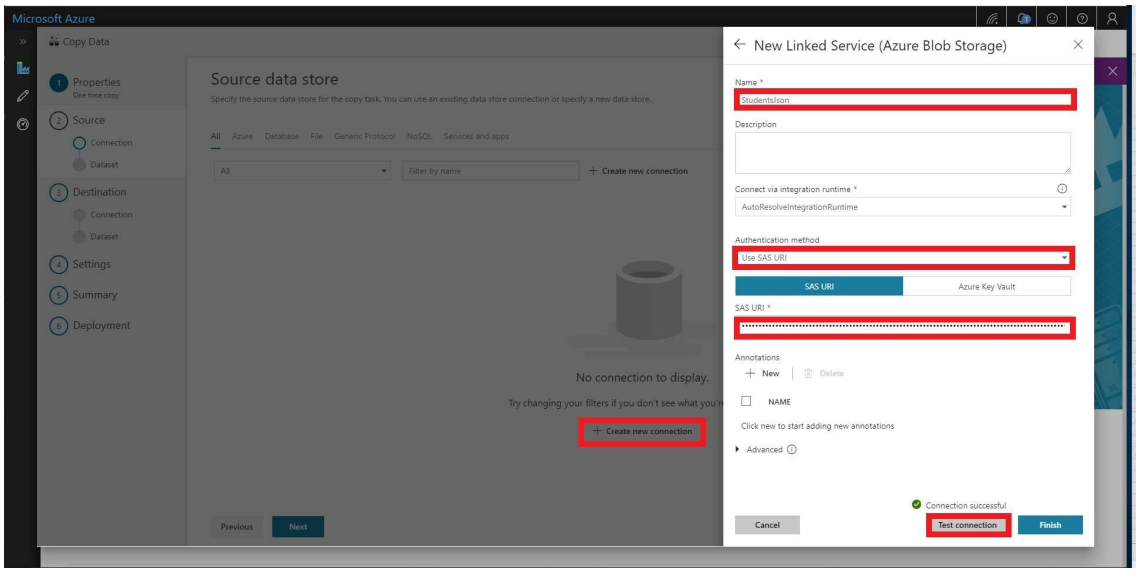
7. Edit basic properties for this data copy. You should name the task ImportStudents and select to Run once now



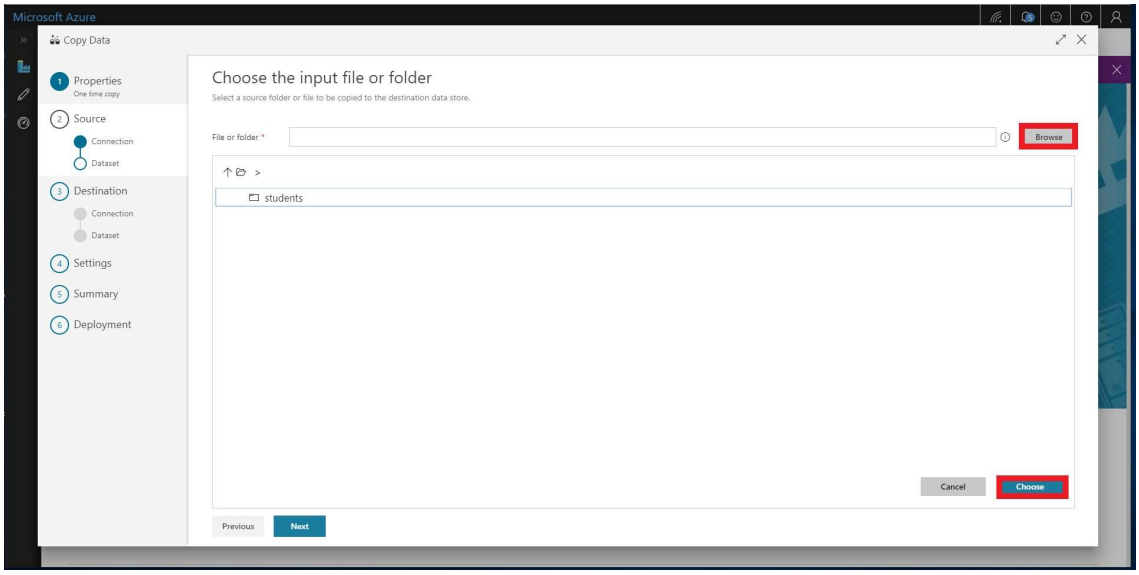
8. Create a new connection and select Azure Blob Storage. We will import data from a json file on Azure Blob Storage. In addition to Blob Storage, you can use ADF to migrate from a wide variety of sources. We will not cover migration from these sources in this tutorial.



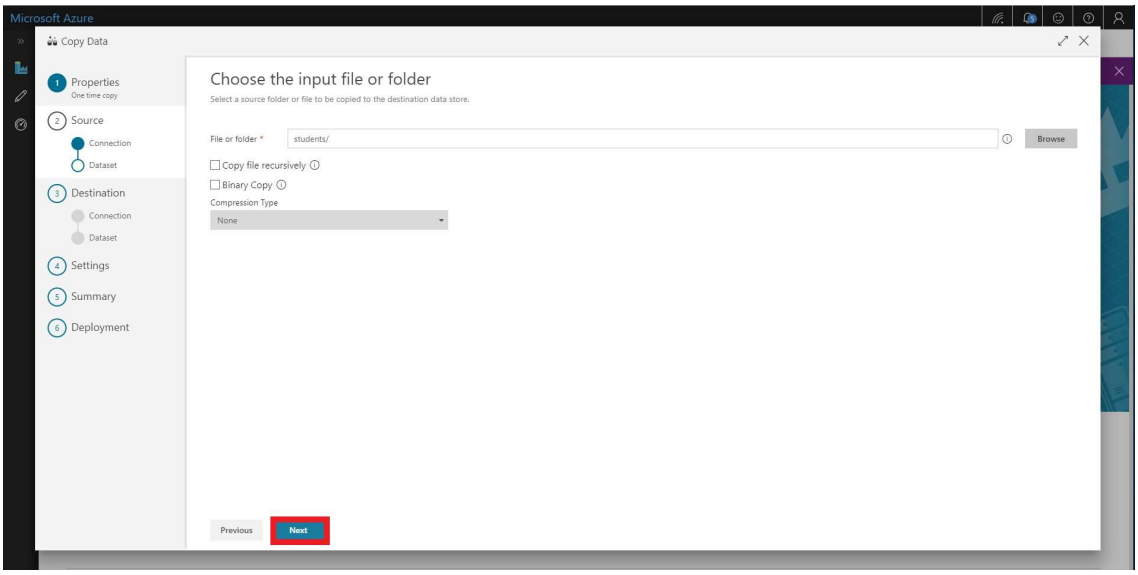
9. Name the source StudentsJson and select Use SAS URI as the Authentication method. Please use the following SAS URI for read-only access to this Blob Storage container:
https://cosmosdblabs.blob.core.windows.net/?sv=2017-11-09&ss=bfqt&srt=sco&sp=rwdlacup&se=2020-03-11T08:08:39Z&st=2018-11-10T02:08:39Z&spr=https&sig=ZSwZhcBdwLVIMRj94pxxGojWwyHkLTAgnL43BkbWKyg%3D



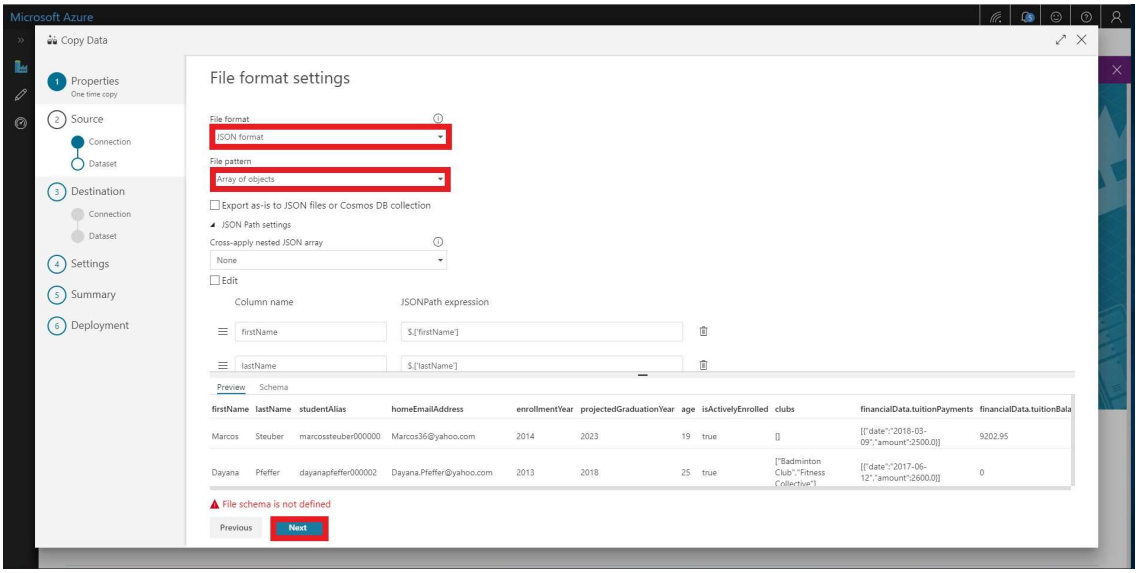
10. Select the students folder



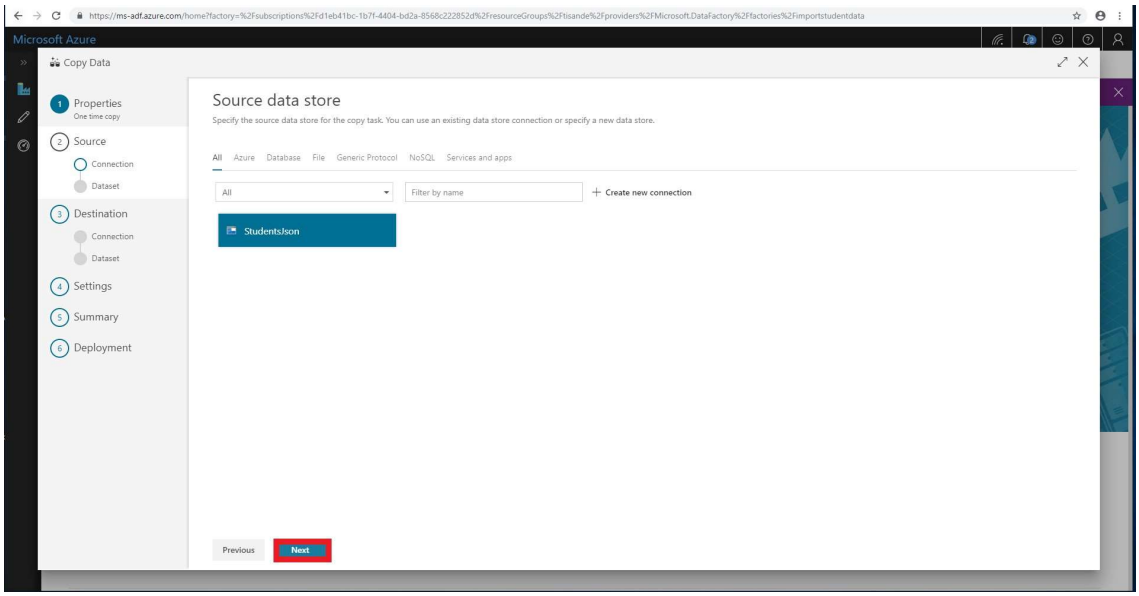
11. Ensure that Copy file recursively and Binary Copy are not checked off. Also ensure that Compression Type is “none”.



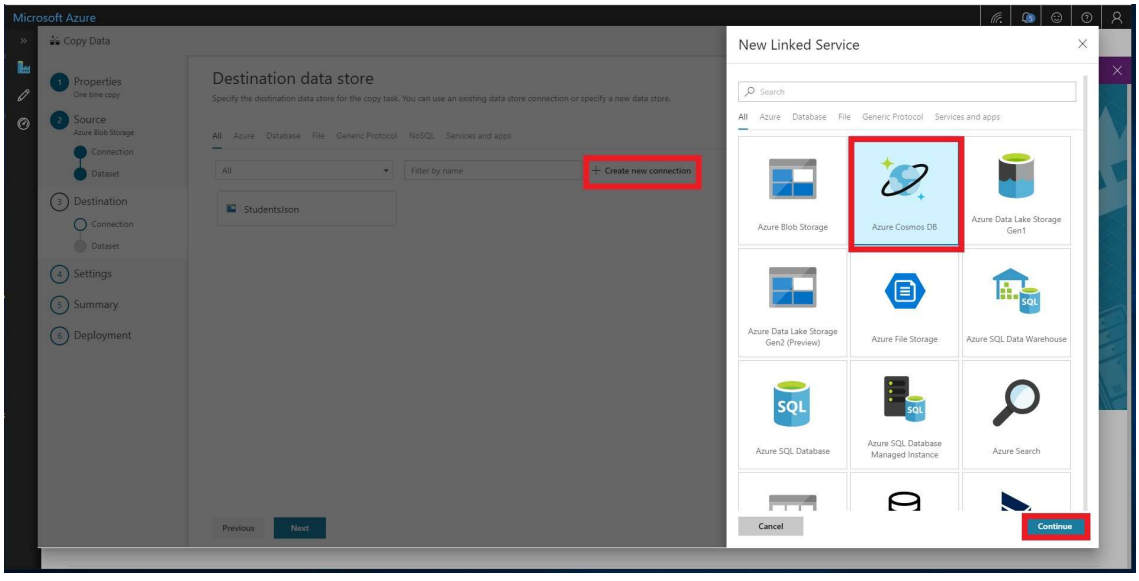
12. ADF should auto-detect the file format to be JSON. You can also select the file format as JSON format. You should also make sure you select Array of Objects as the File pattern.



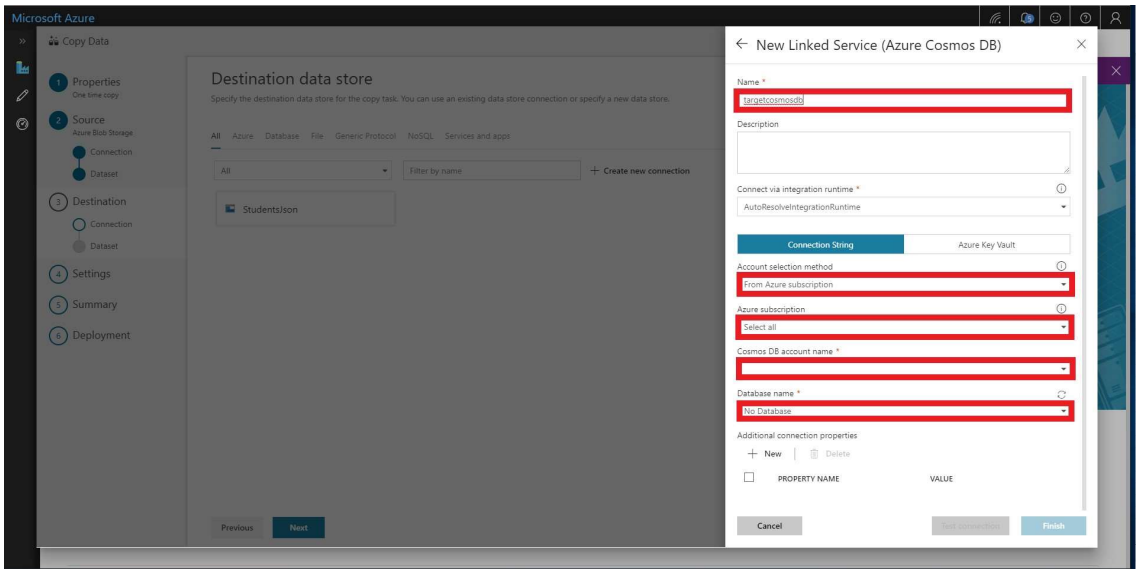
13. You have now successfully connected the Blob Storage container with the students.json file. You should select StudentsJson as the source and click Next.



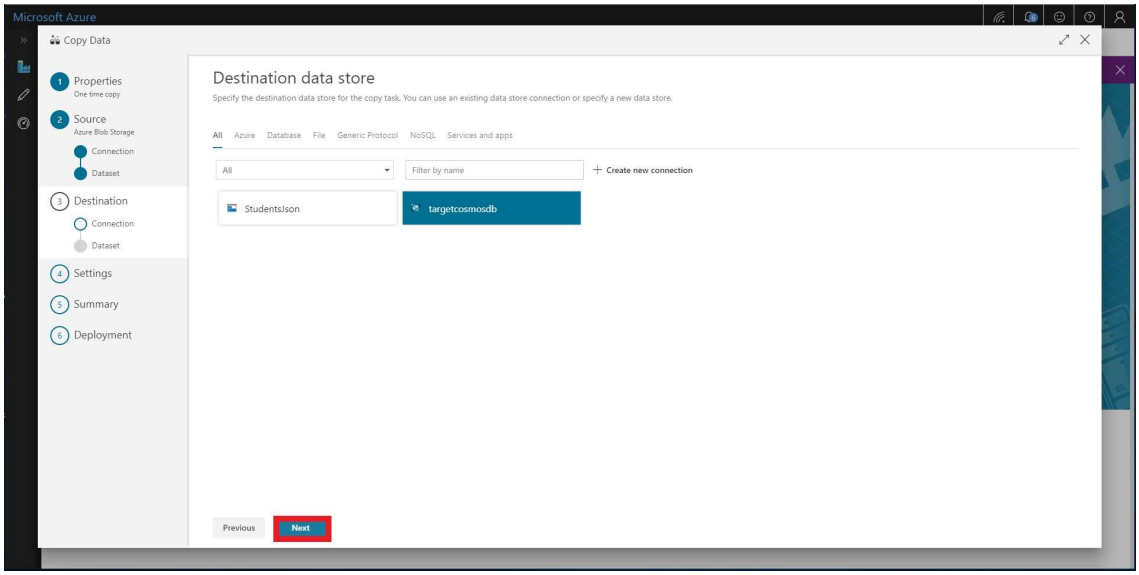
14. Add the Cosmos DB target data store by selecting Create new connection and selecting Azure Cosmos DB (SQL API).



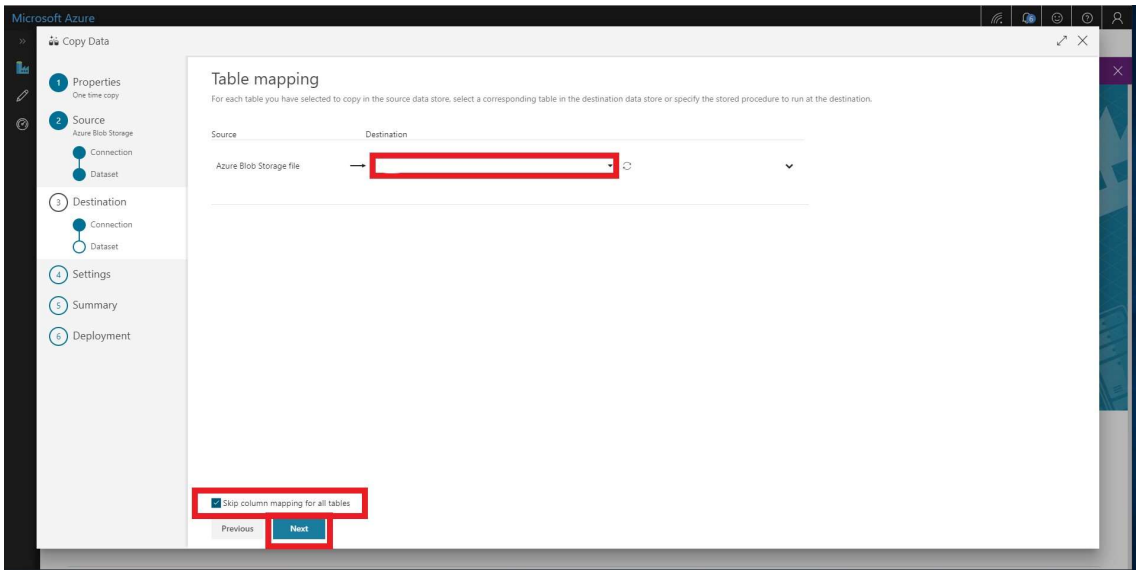
15. Name the linked service targetcosmosdb and select your Azure subscription and Cosmos DB account. You should also select the Cosmos DB database that you created earlier.



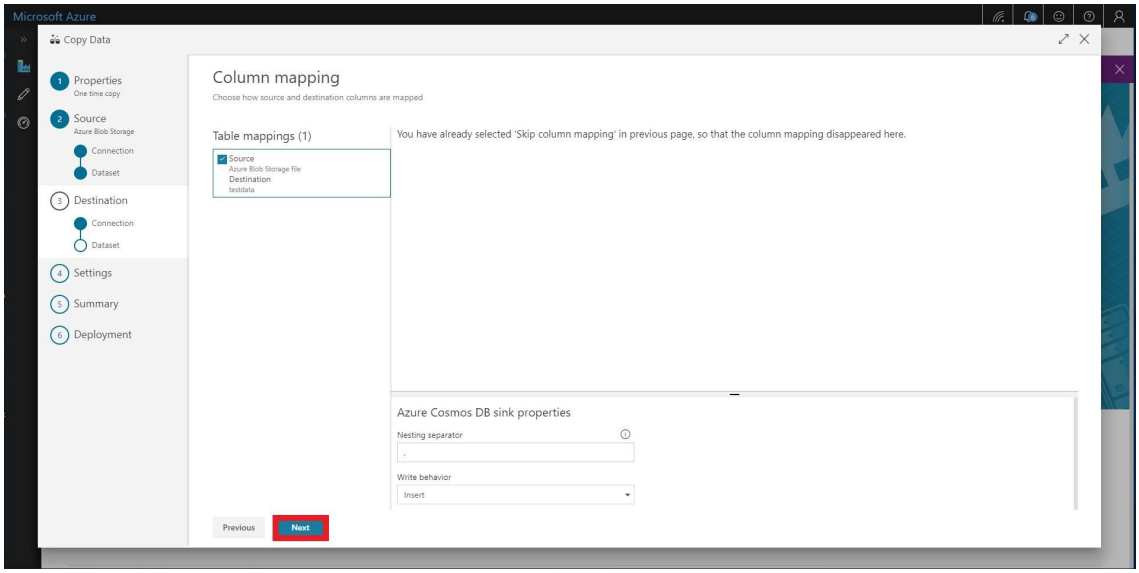
16. Select your newly created targetcosmosdb connection as the Destination date store.



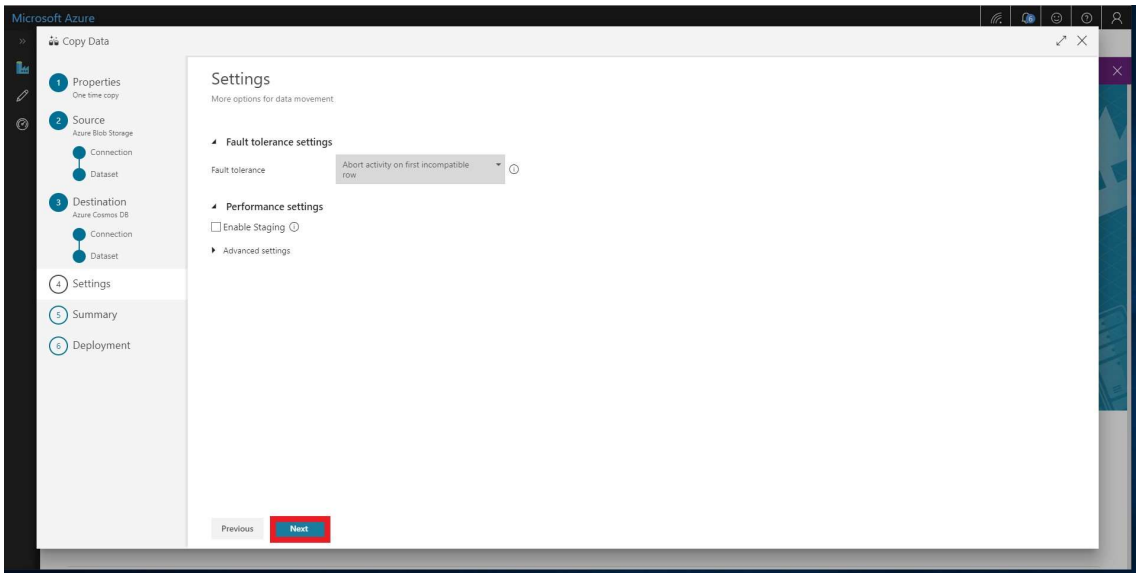
17. Select your collection from the drop-down menu. You will map your Blob storage file to the correct Cosmos DB collection.



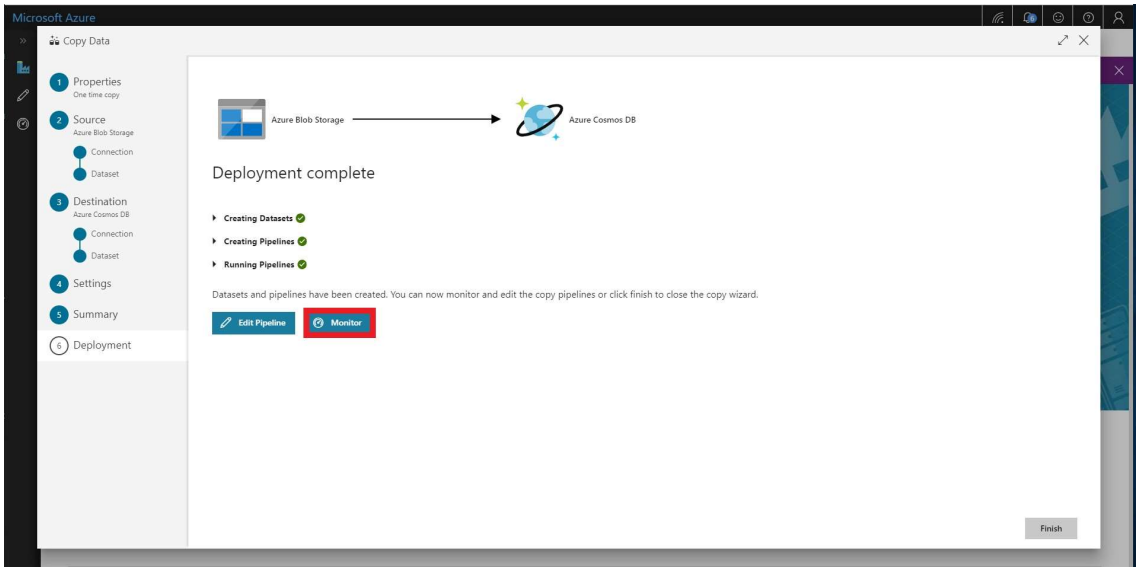
18. You should have selected to skip column mappings in a previous step. Click through this screen.



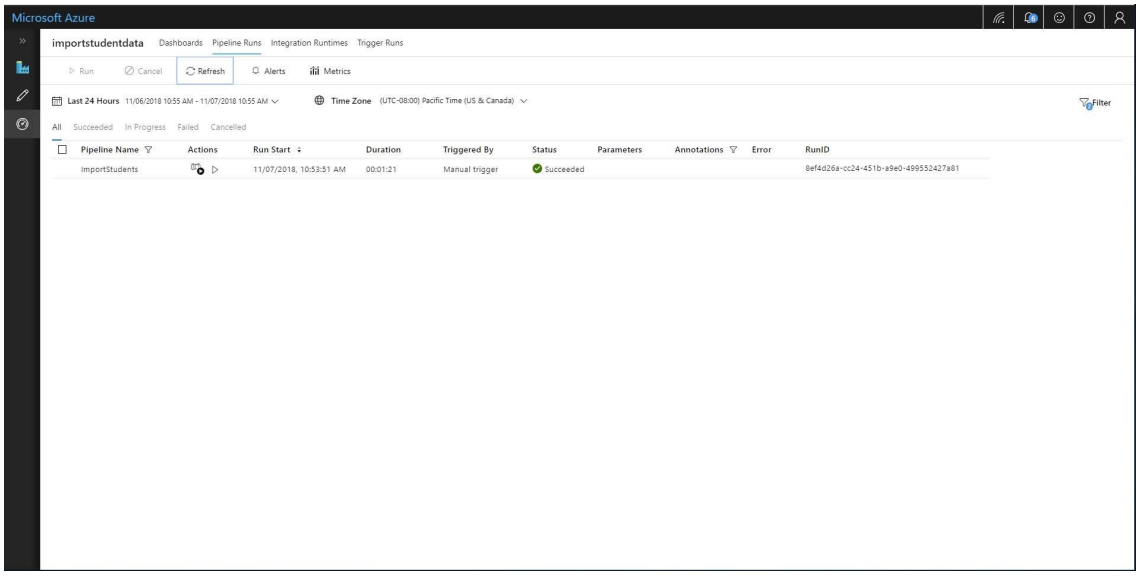
19. There is no need to change any settings. Click next.



20. After deployment is complete, select Monitor.



21. After a few minutes, refresh the page and the status for the ImportStudents pipeline should be listed as **Succeeded**.



22. Once the import process has completed, close the ADF. You will now proceed to execute simple queries on your imported data.

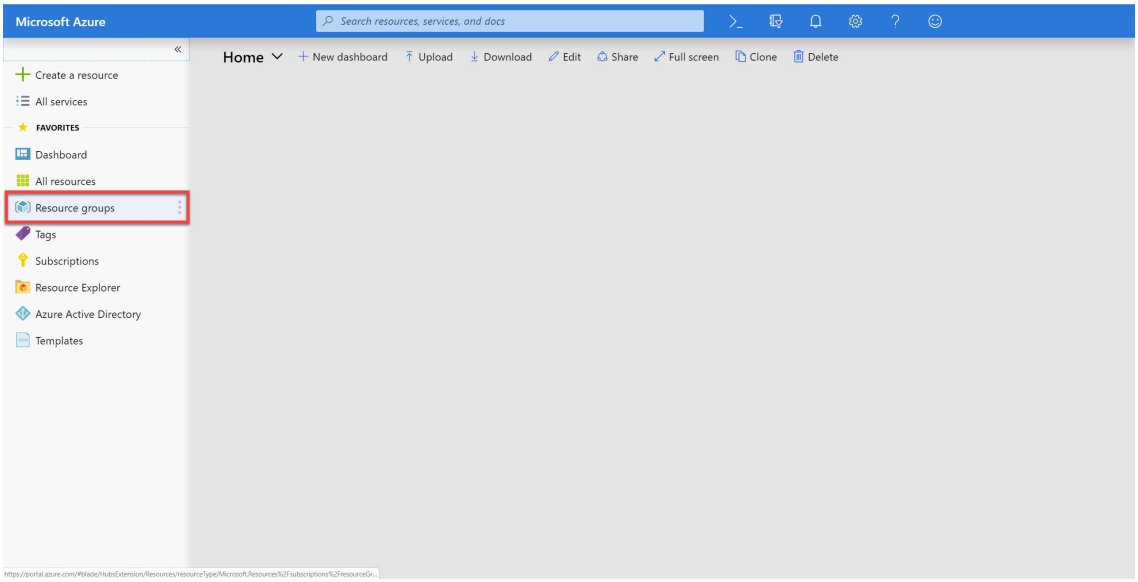
Executing Simple Queries

The Azure Cosmos DB Data Explorer allows you to view documents and run queries directly within the Azure Portal. In this exercise, you will use the Data Explorer to query the data stored in our collection.

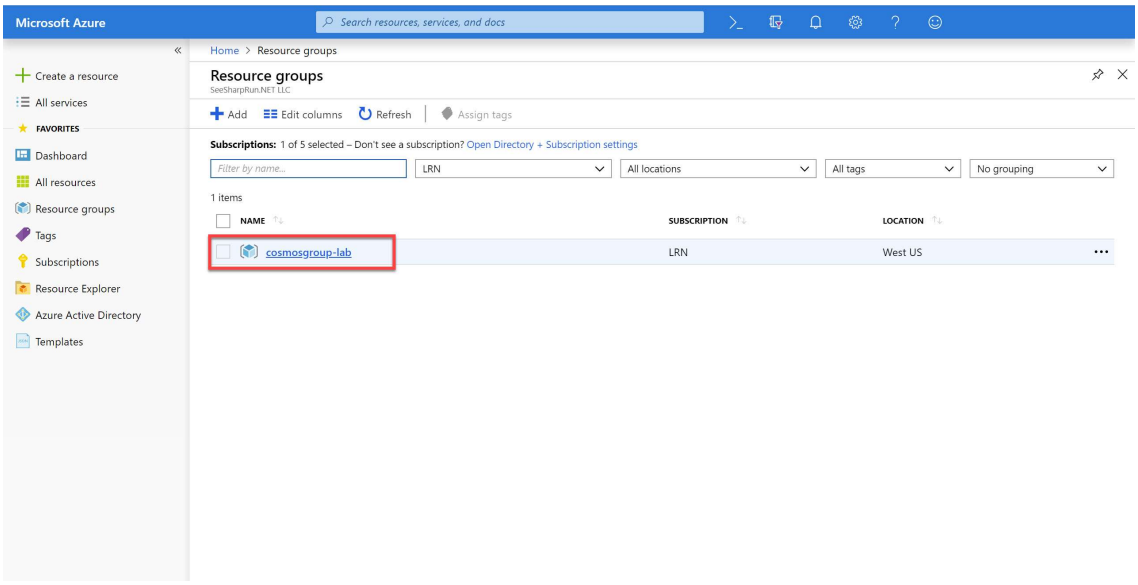
Validate Imported Data

*First, you will validate that the data was successfully imported into your collection using the **Documents** view in the **Data Explorer**.*

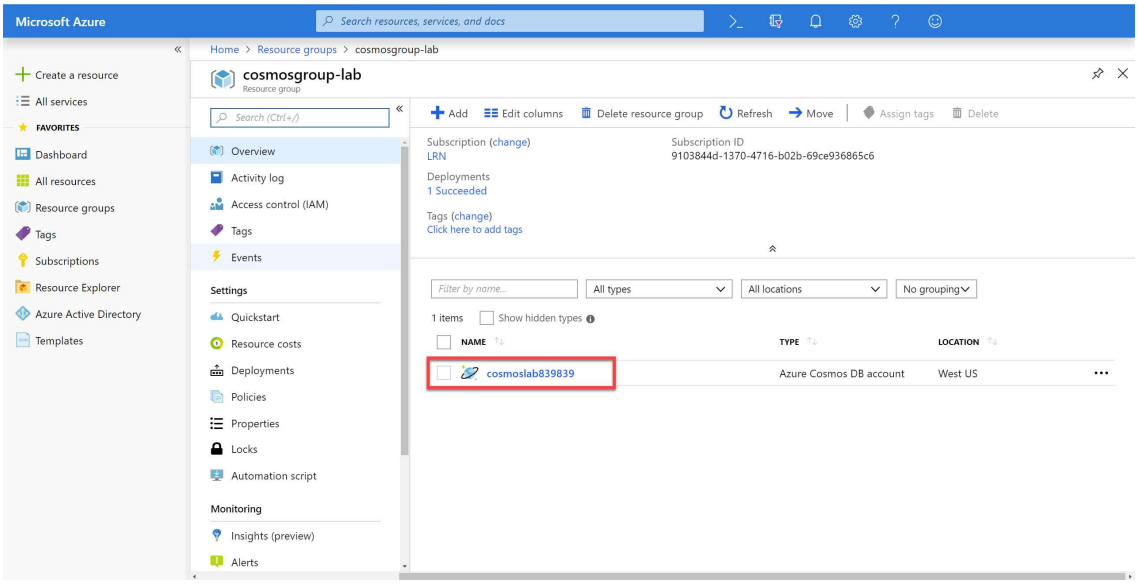
1. Return to the Azure Portal (http://portal.azure.com).
2. On the left side of the portal, click the Resource groups link.



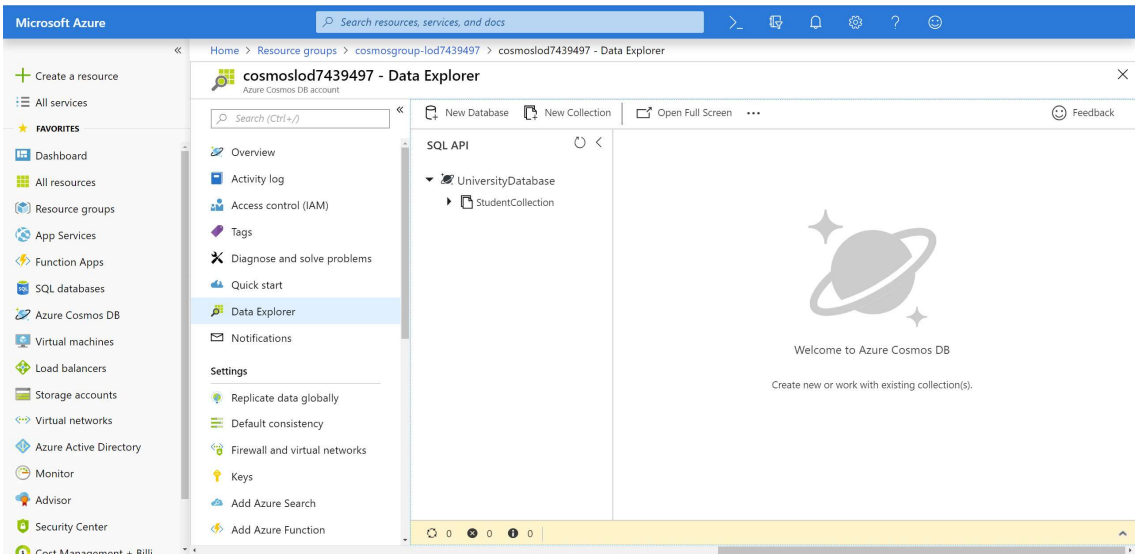
3. In the Resource groups blade, locate and select the cosmosgroup-lab Resource Group.



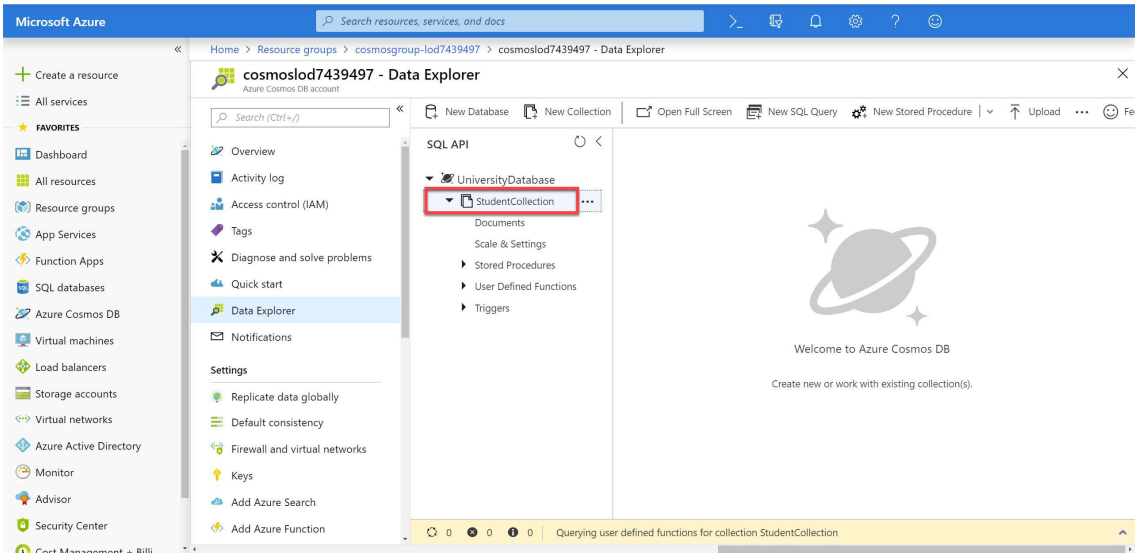
4. In the cosmosgroup-lab blade, select the Azure Cosmos DB account you recently created.



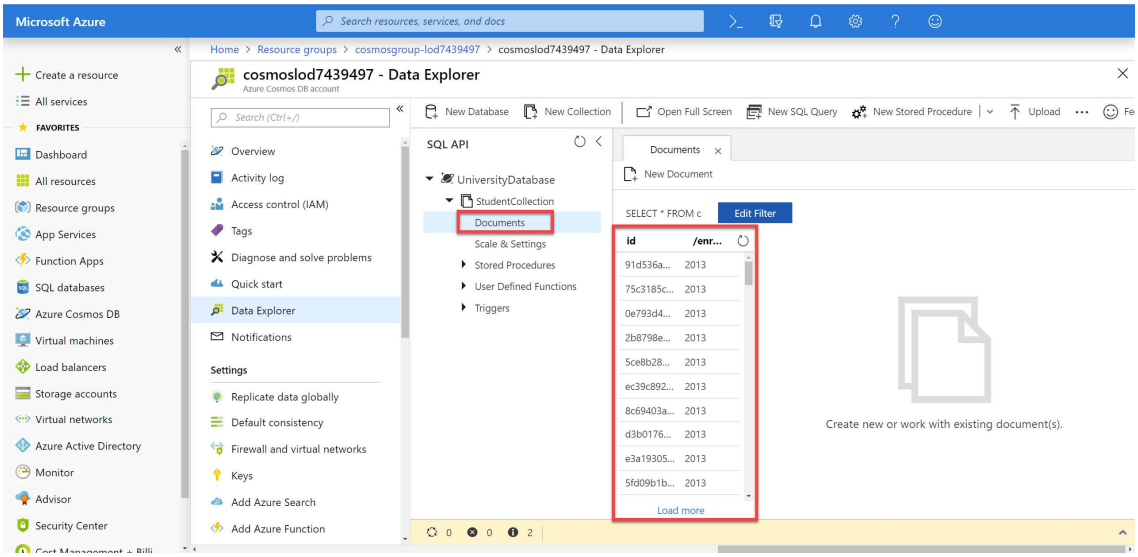
5. In the Azure Cosmos DB blade, locate and click the Data Explorer link on the left side of the blade.

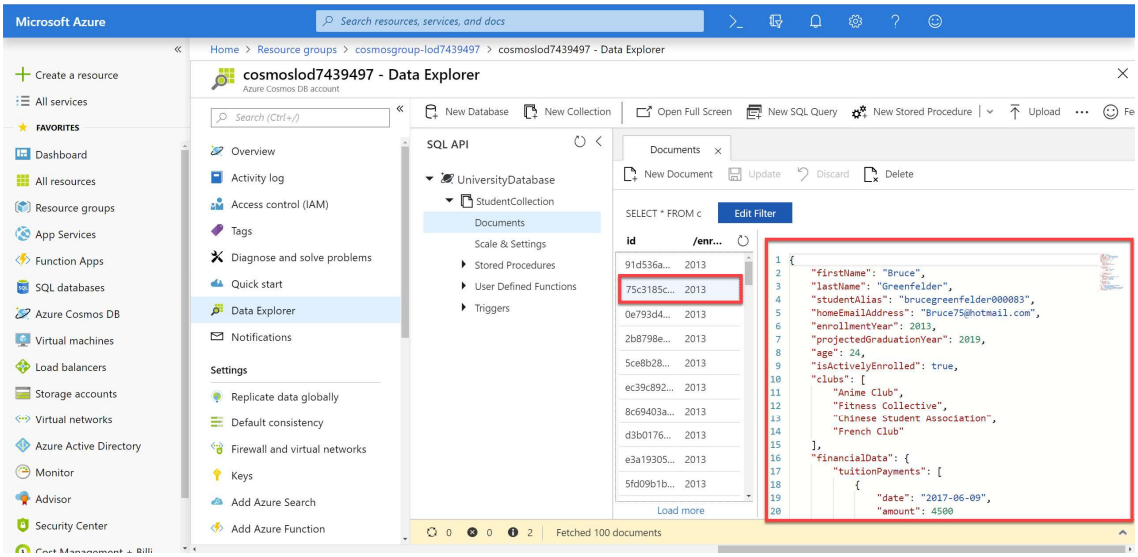


6. In the Data Explorer section, expand the UniversityDatabase database node and then expand the StudentCollectioncollection node.



7. Within the StudentCollection node, click the Documents link to view a subset of the various documents in the collection. Select a few of the documents and observe the properties and structure of the documents.

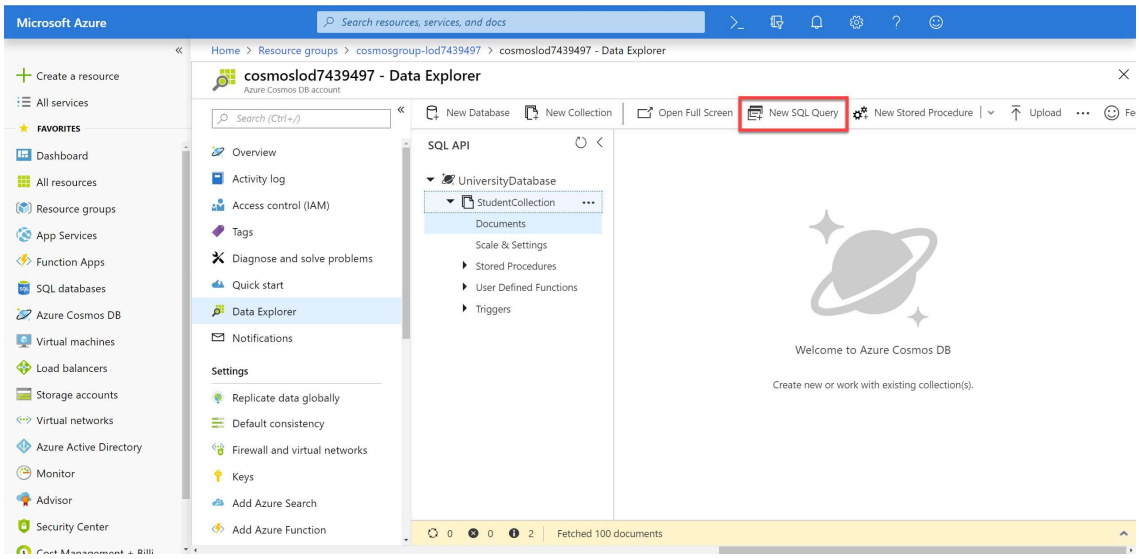




Executing a Simple SELECT Queries

*You will now use the query editor in the **Data Explorer** to execute a few simple SELECT queries using SQL syntax.*

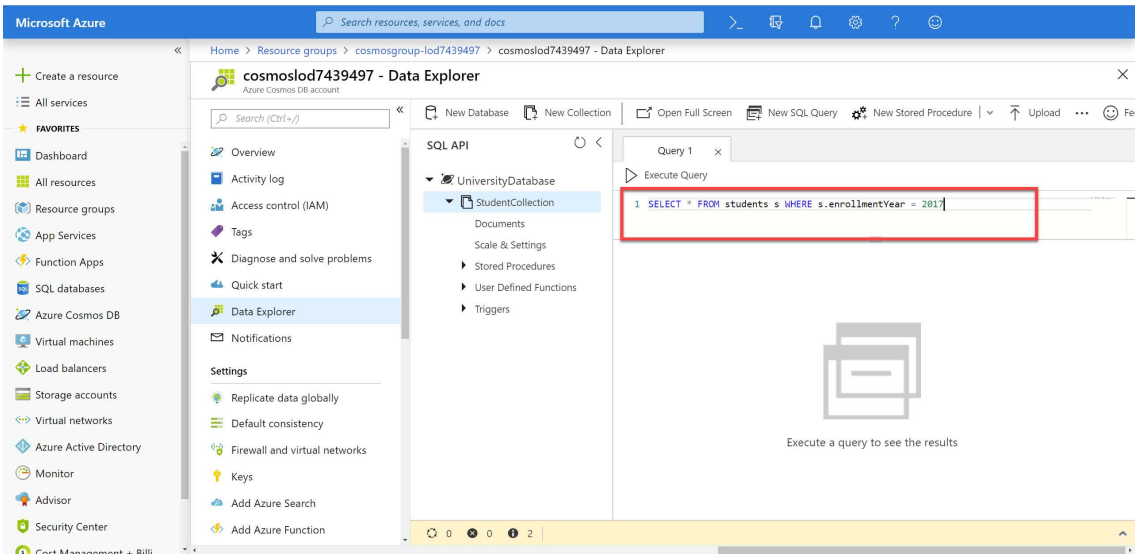
1. Click the New SQL Query button at the top of the Data Explorer section.



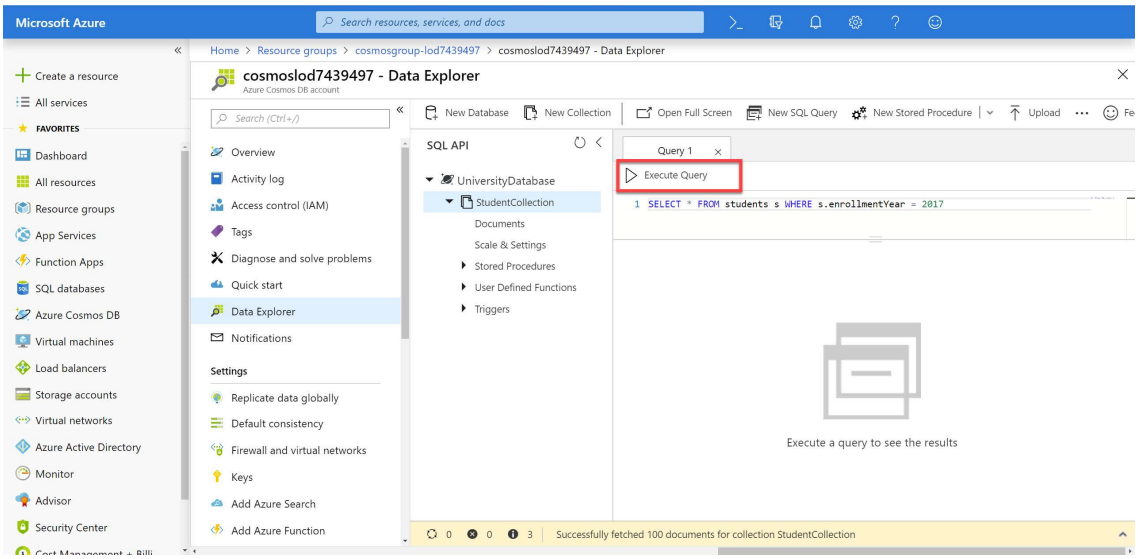
2. In the query tab, replace the contents of the query editor with the following SQL query:

```
SELECT * FROM students s WHERE s.enrollmentYear = 2017
```

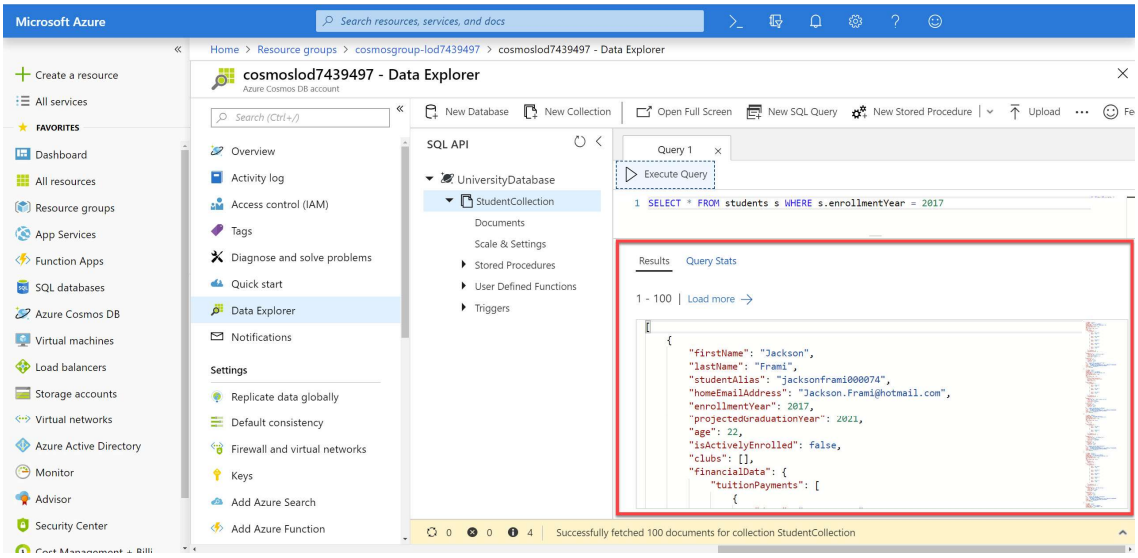
This first query will select all properties from all documents in the collection where the students where enrolled in 2017. You will notice that we are using the alias `s` to refer to the collection.



3. Click the Execute Query button in the query tab to run the query.



4. In the Results pane, observe the results of your query.



5. In the query editor, replace the current query with the following query:

```
SELECT * FROM students WHERE students.enrollmentYear = 2017
```

In this query, we drop the `s` alias and use the `students` source. When we execute this query, we should see the same results as the previous query.

6. Click the Execute Query button in the query tab to run the query. In the Results pane, observe the results of your query.
7. In the query editor, replace the current query with the following query:

```
SELECT * FROM arbitraryname WHERE arbitraryname.enrollmentYear = 2017
```

In this query, we will prove that the name used for the source can be any name you choose. We will use the name `arbitraryname` for the source. When we execute this query, we should see the same results as the previous query.

8. Click the Execute Query button in the query tab to run the query. In the Results pane, observe the results of your query.

9. In the query editor, replace the current query with the following query:

```
SELECT s.studentAlias FROM students s WHERE s.enrollmentYear = 2017
```

> Going back to `s` as an alias, we will now create a query where we only select the `studentAlias` property and return the value of that property in our result set.

10. Click the Execute Query button in the query tab to run the query. In the Results pane, observe the results of your query.
11. In the query editor, replace the current query with the following query:

```
SELECT VALUE s.studentAlias FROM students s WHERE s.enrollmentYear = 2017
```

> In some scenarios, you may need to return a flattened array as the result of your query. This query uses the `VALUE` keyword to flatten the array by taking the single returned (string) property and creating a string array.

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

Implicitly Executing a Cross-Partition Query

The Data Explorer will allow you to create a cross-partition query without the need to manually configure any settings. You will now use the query editor in the Data Explorer to perform single or multi-partition queries

1. Back in the query tab, replace the contents of the query editor with the following SQL query:

```
SELECT * FROM students s WHERE s.enrollmentYear = 2016
```

> Since we know that our partition key is `/enrollmentYear`, we know that any query that targets a single valid value for the `enrollmentYear` property will be a single partition query.

2. Click the Execute Query button in the query tab to run the query. In the Results pane, observe the results of your query.

> Observe the Request Charge (in RU/s) for the executed query.

3. In the query editor, replace the current query with the following query:

```
SELECT * FROM students s
```

If we want to execute a blanket query that will fan-out to all partitions, we simply can drop our `WHERE` clause that filters on a single valid value for our partition key path.

4. Click the Execute Query button in the query tab to run the query. In the Results pane, observe the results of your query.

Observe the Request Charge (in RU/s) for the executed query. You will notice that the charge is relatively greater for this query.

5. Back in the query tab, replace the contents of the query editor with the following SQL query:

```
SELECT * FROM students s WHERE s.enrollmentYear IN (2015, 2016, 2017)
```

Observe the Request Charge (in RU/s) for the executed query. You will notice that the charge is greater than a single partition but far less than a fan-out across all partitions.

6. Click the Execute Query button in the query tab to run the query. In the Results pane, observe the results of your query.

Observe the Request Charge (in RU/s) for the executed query.

Use Built-In Functions

There are a large variety of built-in functions available in the SQL query syntax for the SQL API in Azure Cosmos DB. We will focus on a single function in this task but you can learn more about the others here: https://docs.microsoft.com/azure/cosmos-db/sql-api-sql-query-reference

1. In the query editor, replace the current query with the following query:

```
SELECT s.studentAlias FROM students s WHERE s.enrollmentYear = 2015
```

Our goal is to get the school-issued e-mail address for all students who enrolled in 2015. We can issue a simple query to start that will return the login alias for each student.

2. Click the Execute Query button in the query tab to run the query. In the Results pane, observe the results of your query.
3. In the query editor, replace the current query with the following query:

```
SELECT CONCAT(s.studentAlias, '@contoso.edu') AS email FROM students s WHERE s.enrollmentYear = 2015
```

To get the school-issued e-mail address, we will need to concatenate the `@contoso.edu` string to the end of each alias. We can perform this action using the `CONCAT` built-in function.

4. Click the Execute Query button in the query tab to run the query. In the Results pane, observe the results of your query.
5. In the query editor, replace the current query with the following query:

```
SELECT VALUE CONCAT(s.studentAlias, '@contoso.edu') FROM students s WHERE s.enrollmentYear = 2015
```

In most client-side applications, you likely would only need an array of strings as opposed to an array of objects. We can use the `VALUE` keyword here to flatten our result set.

6. Click the Execute Query button in the query tab to run the query. In the Results pane, observe the results of your query.

Projecting Query Results

In some use cases, we may need to reshape the structure of our result JSON array to a structure that our libraries or third-party APIs can parse. We will focus on a single query and re-shape the results into various formats using the native JSON capabilities in the SQL query syntax.

1. In the query editor, replace the current query with the following query:

```
SELECT
    CONCAT(s.firstName, " ", s.lastName),
    s.academicStatus.warning,
    s.academicStatus.suspension,
    s.academicStatus.expulsion,
    s.enrollmentYear,
    s.projectedGraduationYear
FROM students s WHERE s.enrollmentYear = 2014
```

In this first query, we want to determine the current status of every student who enrolled in 2014. Our goal here is to eventually have a flattened, simple-to-understand view of every student and their current academic status.

2. Click the Execute Query button in the query tab to run the query. In the Results pane, observe the results of your query.

You will quickly notice that the value representing the name of the student, using the `CONCAT` function, has a placeholder property name instead of a simple string.

3. In the query editor, replace the current query with the following query:

```
SELECT
    CONCAT(s.firstName, " ", s.lastName) AS name,
    s.academicStatus.warning,
    s.academicStatus.suspension,
    s.academicStatus.expulsion,
    s.enrollmentYear,
    s.projectedGraduationYear
FROM students s WHERE s.enrollmentYear = 2014
```

We will update our previous query by naming our property that uses a built-in function.

4. Click the Execute Query button in the query tab to run the query. In the Results pane, observe the results of your query.
5. In the query editor, replace the current query with the following query:

```
SELECT {
    "name": CONCAT(s.firstName, " ", s.lastName),
    "isWarned": s.academicStatus.warning,
    "isSuspended": s.academicStatus.suspension,
    "isExpelled": s.academicStatus.expulsion,
    "enrollment": {
        "start": s.enrollmentYear,
        "end": s.projectedGraduationYear
    }
} AS studentStatus
FROM students s WHERE s.enrollmentYear = 2014
```

Another alternative way to specify the structure of our JSON document is to use the curly braces from JSON. At this point, we are defining the structure of the JSON result directly in our query.

6. Click the Execute Query button in the query tab to run the query. In the Results pane, observe the results of your query.

You should notice that our JSON object is still "wrapped" in another JSON object. Essentially, we have an array of the parent type with a property named `studentStatus` that contains the actual data we want.

7. In the query editor, replace the current query with the following query:

```
SELECT VALUE {
    "name": CONCAT(s.firstName, " ", s.lastName),
    "isWarned": s.academicStatus.warning,
    "isSuspended": s.academicStatus.suspension,
    "isExpelled": s.academicStatus.expulsion,
    "enrollment": {
        "start": s.enrollmentYear,
        "end": s.projectedGraduationYear
    }
} FROM students s WHERE s.enrollmentYear = 2014
```

If we want to "unwrap" our JSON data and flatten to a simple array of like-structured objects, we need to use the `VALUE` keyword.

8. Click the Execute Query button in the query tab to run the query. In the Results pane, observe the results of your query.

Use Python SDK to Query Azure Cosmos DB

After using the Azure Portal's Data Explorer to query an Azure Cosmos DB collection, you are now going to use the Python SDK to issue similar queries.

Create a new file to house your simple query

Open Visual Studio Code in the directory you have been working in and create a simpleQuery.py source file. Copy in the following source code

```
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 = "UniversityDatabase";
    collectionID = "StudentCollection"
    database_link = 'dbs/' + databaseID
    collection_link = database_link + '/colls/' + collectionID

    queryStr = """SELECT TOP 5 VALUE s.studentAlias FROM coll s
                  WHERE s.enrollmentYear = 2018 ORDER BY s.studentAlias"""
    resultSet = client.QueryItems(collection_link, queryStr)

    for res in resultSet:
        print(res)
```

The code above loads up the endpoint and key values from the config.json file we configured earlier. As in the earlier examples we create a Cosmos DB client object which has methods that will allow us to query the Cosmos DB service. We then set values for the database_link and collection_like for our StudentCollection and we define a query string to pass the QueryItems method of the client object.
The return value from the client.QueryItems method is an azure.cosmos.query_iterable and implements the iterator protocol meaning you can iterate over it and extract data elements. The final for loop extracts and prints the results from the resultSet azure.cosmos.query_iterable object.

Execute the code by pressing F5 check that the code executes successfully.

Set a breakpoint in VS Code with F9 or selecting the left column of the code at the `print(res)` line and look at the type of the res variable. It is a simple string because of the VALUE function in the query.
Look at the content of the resultSet object in the debug pane – have a look and see if you can find the last_response_headers attribute, explore its contents.

Query an Intra-document Array

1. We will now change the query to bring back a list of clubs for each student. Note that the list of clubs is a list of strings embedded within the document of the student collection. It is a list attribute of the student document and we will return an object when we iterate over the azure.cosmos.query_iterable returned by the client.queryItem method.

   In the simpleQuery.py replace the queryStr with the code below.

```
    queryStr = """SELECT s.clubs FROM students s WHERE
                  s.enrollmentYear = 2018"""
```

   Notice that a dictionary object is returned with a single key clubs, containing a list of clubs.
   You can run the code as is to see the operation of the program even though the client.queryItem method returns an object rather than a string as the Python print function can cope with displaying objects in most cases.

2. We'll now create a simple Student class object to hold each student data document and create a list of Students and their activities. Add the following class definition immediately below the import statements in your simpleQuery.py source file

```
class Student:
    def __init__(self, firstName = None, lastName = None, clubs = None):
        self.firstName = firstName
        self.lastName = lastName
        self.clubs = clubs
```

   The code defines a simple Student class with an initialiser method __init__.

3. Alter the code after the client.QueryItems call with the code below to create the list of Student objects and print out the student name and their activities.

```
    studentList = []
    for res in resultSet:
        newStudent = Student(res['firstName'], res['lastName'], res['clubs'])
        studentList.append(newStudent)
        print(newStudent.firstName, newStudent.lastName + ' activities = ', end = '')
        for act in newStudent.clubs:
            print(act + ", ", end = '')
        print()
```

4. Run the program and confirm that a list of students and their activities are printed out. Insert a breakpoint in the last section and inspect the newStudent object.

Joining to Attributes Embedded within Documents

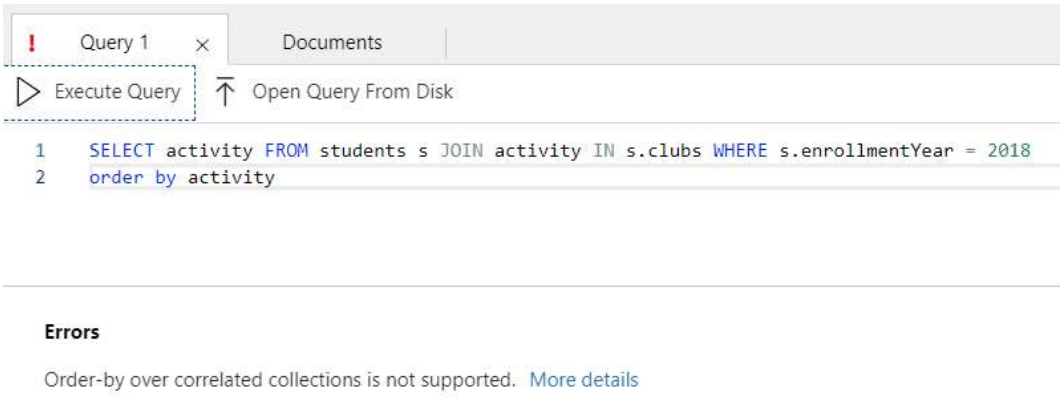
In this next section we will get a list of all of the activities that are available in the clubs run at the university. We will use the join SQL syntax to extract the list of clubs embedded in the student documents.

1. Create a new file in Visual Studio Code called getStudentActivities.py and copy the code below into the new 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 = "UniversityDatabase";
    collectionID = "StudentCollection"
    database_link = 'dbs/' + databaseID
    collection_link = database_link + '/colls/' + collectionID

    queryStr = """SELECT activity FROM students s JOIN activity IN s.clubs
                  WHERE s.enrollmentYear = 2018"""

    resultSet = client.QueryItems(collection_link, queryStr)

    for res in resultSet:
        print(res)
```

2. Observe the syntax of the queryStr using the join syntax to join an empty alias (activity) with the contents of the inner clubs array as if it were a table in its own right.

3. Run the code (press F5) and look at the output, again the forgiving print function in Python is able to render the dictionary returned for each element in the resultSet. The code returns all of activities in the club array for every student as a dictionary object. There are many duplicates in the resultSet and the syntax does not support ordering in the SQL API. If you paste the following query into the query window of the Data Explorer in the Azure Portal, you will see an error saying ordering is not supported over corelated collections.



4. In Python we can use a "set" data structure to ensure there are no duplicates and then use the sorted() function to sort the values in the set. Copy the code below and run into your getStudentActivities.py file replacing everything after the `"resultSet = client.QueryItems(collection_link, queryStr)"` line and execute the file by pressing F5.

```
studentActivities = set()

for res in resultSet:
    studentActivities.add(res['activity'])

for activity in sorted(studentActivities):
    print(activity)
```

5. Your program should now output all of the activities that students are taking part in ordered and without duplicates. Notice how long the program takes to execute. Can you think of a drawback of this approach when working with much larger result sets?

Projecting Query Results

In this example we will use a more complex query to build up and return a dictionary object for each student with their name, home email, college email address and their unique identifier in the collection.

1. Create a new file in Visual Studio Code called projectingQueryResults.py and copy the following code.

```
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 = "UniversityDatabase";
    collectionID = "StudentCollection"
    database_link = 'dbs/' + databaseID
    collection_link = database_link + '/colls/' + collectionID

    queryStr = """ SELECT VALUE {
                                 "id": s.id,
                                 "name": CONCAT(s.firstName, " ", s.lastName),
                                 "email": {
                                 "home": s.homeEmailAddress,
                                 "school": CONCAT(s.studentAlias, '@contoso.edu')
                                           }
                                } FROM students s
                                WHERE s.enrollmentYear = 2018 """

    resultSet = client.QueryItems(collection_link, queryStr)

    for res in resultSet:
        print(res)
```

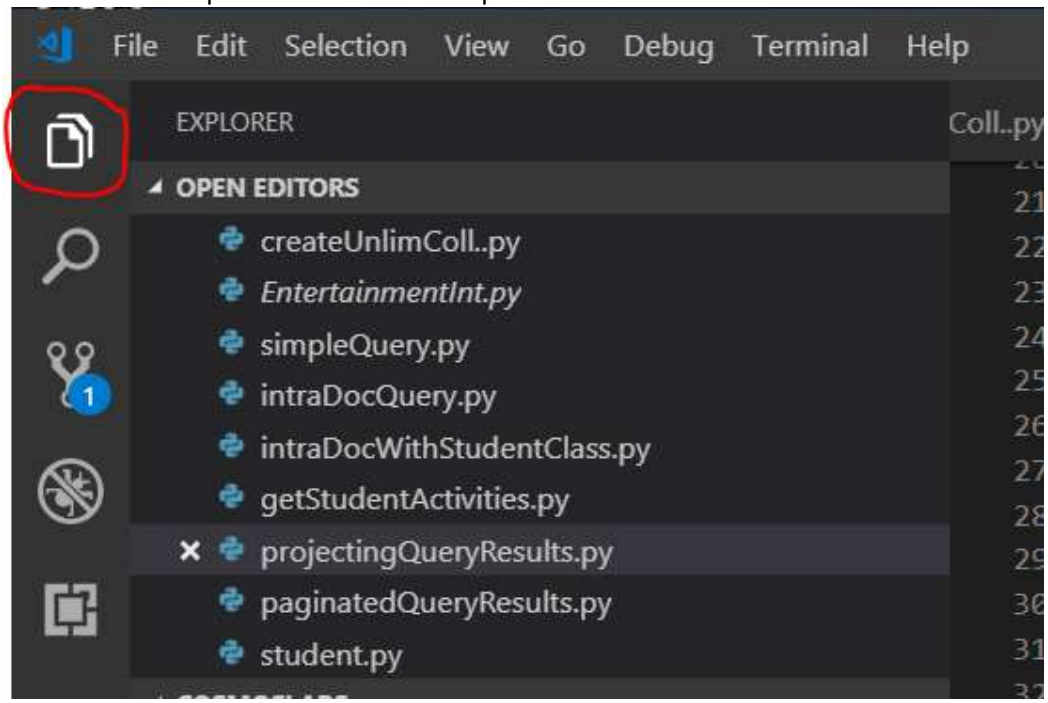
   Once again we rely on the Python print() function to render the dictionary object returned as we iterate through the resultSet.

2. Place a breakpoint on the `print(res)` line and examine the content of the res variable. You can see how it is possible to easily build up complex data structures in the SQL API syntax of Cosmos DB and quickly use them in the Python language.

Implement Pagination using the Python SDK

It is possible to paginate large result sets within the Python SDK returning only an initial page of results and allowing the client to query for additional documents if required. We will investigate this functionality in this section.

1. Select the File Explorer in the left hand pane of visual Studio Code as shown below



2. In the lower files pane right click on the projectingQueryResults.py file and select Copy.

3. Unselect the file and right-click on the mouse in the files pane and select the Paste option

4. Rename the resulting new file from projectingQueryResults1.py to paginatedQueryResults.py

5. Make the following changed to the paginatedQueryResults.py file after the `queryStr =` definition line

```
options = {'maxItemCount': 10, 'continuation': True}
resultSet = client.QueryItems(collection_link, queryStr, options)

while True:
    result_block = resultSet.fetch_next_block()
    if result_block:
        for res in result_block:
            print(res)

        print("=======Page-Break========")
    else:
         break
```

6. Notice the addition of the options dictionary and how it is passed to the client.QueryItems() method. In the iterate code we manually iterate over the resultSet page by page using the fetch_next_block method of the resultSet azure.cosmos.query_iterable object.

7. Execute the paginatedQueryResults.py program in visual studio code and observe how the results are brought back in 10 item pages.

8. Set a breakpoint on the last break line and look at the value of result_block. It should be an empty array (which is equivalent to Boolean False) if there are no more documents to fetch.

In practice the fetch_next_method() is rarely used as the Cosmos DB Python API implements the iterator protocol in the azure.cosmos.query_iterable object and iterating over this with a for loop is a much more natural (and Pythonic) way of handling the documents.

Implementing Cross-Partition Queries

In this final section of this lab we will illustrate how Azure Cosmos DB implements querying across partitions. When this happens we could be setting off fan-out queries that touch multiple physical partitions residing on different servers. This is not necessarily a bad thing, however it is as well to be aware that this operation is occurring as it may affect your application's ability to scale if it is executing too many fan-out queries concurrently and may also cause your application to be quite resource hungry. The default query mode for Azure Cosmos DB does not permit cross-partition queries and they must be enabled with an option in the in the client.QueryItems call.

Execute Single-Partition Query

First we will restrict our query to a single partition key using the `PartitionKey` property of the `options` dictionary. One of our partition key values for the `/enrollmentYear` path is `2016`. We will filter our query to only return documents that uses this partition key. Remember, partition key paths are case sensitive. Since our property is named `enrollmentYear`, it will match on the partition key path of `/enrollmentYear`.

1. In Visual Studio Code copy the paginatedQueryResults.py file and rename the copied file to singlePartitionQuery.py
2. Alter the SQL in queryStr from :

```
    queryStr = """ SELECT VALUE {
                                "id": s.id,
                                "name": CONCAT(s.firstName, " ", s.lastName),
                                "email": {
                                   "home": s.homeEmailAddress,
                                   "school": CONCAT(s.studentAlias, '@contoso.edu')
                                           }
                               } FROM students s
                               WHERE s.enrollmentYear = 2018 """
```

To the following:

```
    queryStr = """ SELECT s.firstName, s.lastName, s.enrollmentYear,
                   s.projectedGraduationYear
                   FROM students s
                   WHERE s.projectedGraduationYear < 2020
                   """
```

3. Run the singlePartitionQuery.py by pressing F5. Notice the Cosmos DB query errors with an HTTP 400 error stating that cross partition queries are not allowed by default. This query should return students from many enrolment years – but this would involve visiting more than one partition.
4. Modify the options dictionary to include a value for the partitionKey
   Alter the options dictionary from

```
    options = {'maxItemCount': 10, 'continuation': True }
```

   to

```
    options = {'maxItemCount': 10, 'continuation': True, 'partitionKey': 2016}
```

5. Re run the singlePartitionQuery.py program by pressing F5 and note that the query now returns values, but only from the partition where enrollmentYear = 2016

Execute Cross-Partition Query

1. In Visual Studio Code copy the singlePartitionQuery.py file and rename the copied file to crossPartitionQuery.py
2. Remove the partitionKey attribute from the options directory variable and replace it with an enableCrossPartitionQuery Boolean attribute set to True as in the code fragment below:

```
    options = {'maxItemCount': 10, 'continuation': True,
                'enableCrossPartitionQuery': True }
```

3. Notice the query now runs and returns documents where the enrollmentYear property has a variety of values.

Cross partition queries are disabled by default in the Azure Cosmos DB API but that does not necessarily mean they are to be avoided. Azure Cosmos DB will fan out the query in parallel so the time to produce your result set will be reduced, however the resource utilisation for such queries can be quite high if they are visiting many partitions. The fact that your application needs to do cross partition queries is something that you need to be aware of and manage accordingly.

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