## Cosmos DB Labs for Python Programmers Lab 1 Cosmos DB Databases and Collections

In this lab, you will create multiple Azure Cosmos DB containers. Some of the containers will be unlimited and configured with a partition key, while others will be fixed-sized. You will then use the SQL API and .NET SDK to query specific containers using a single partition key or across multiple partition keys.

## Exercise 1 Log-in to the Azure Portal

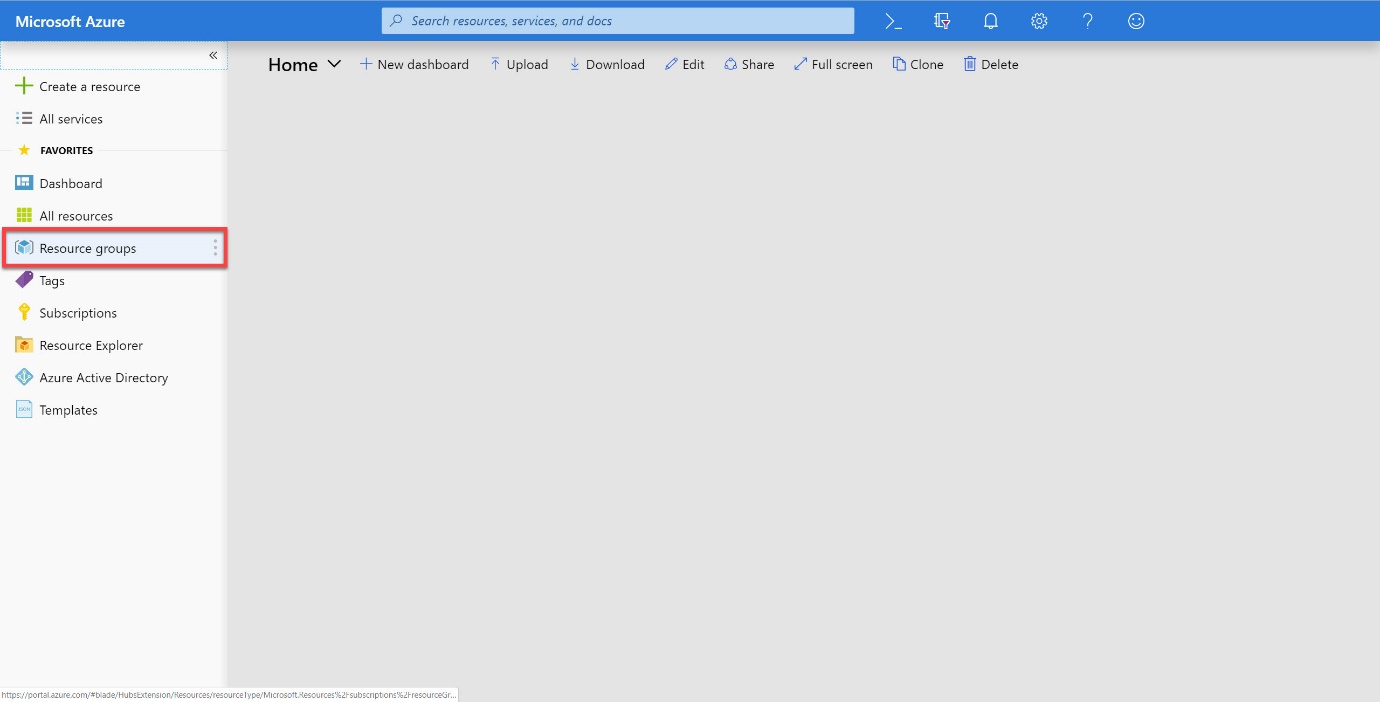
1. In a new window, sign in to the **Azure Portal** ([http://portal.azure.com](http://portal.azure.com/)).
2. Once you have logged in, you may be prompted to start a tour of the Azure portal. You can safely skip this step.

## Exercise 2 Azure Cosmos DB Setup

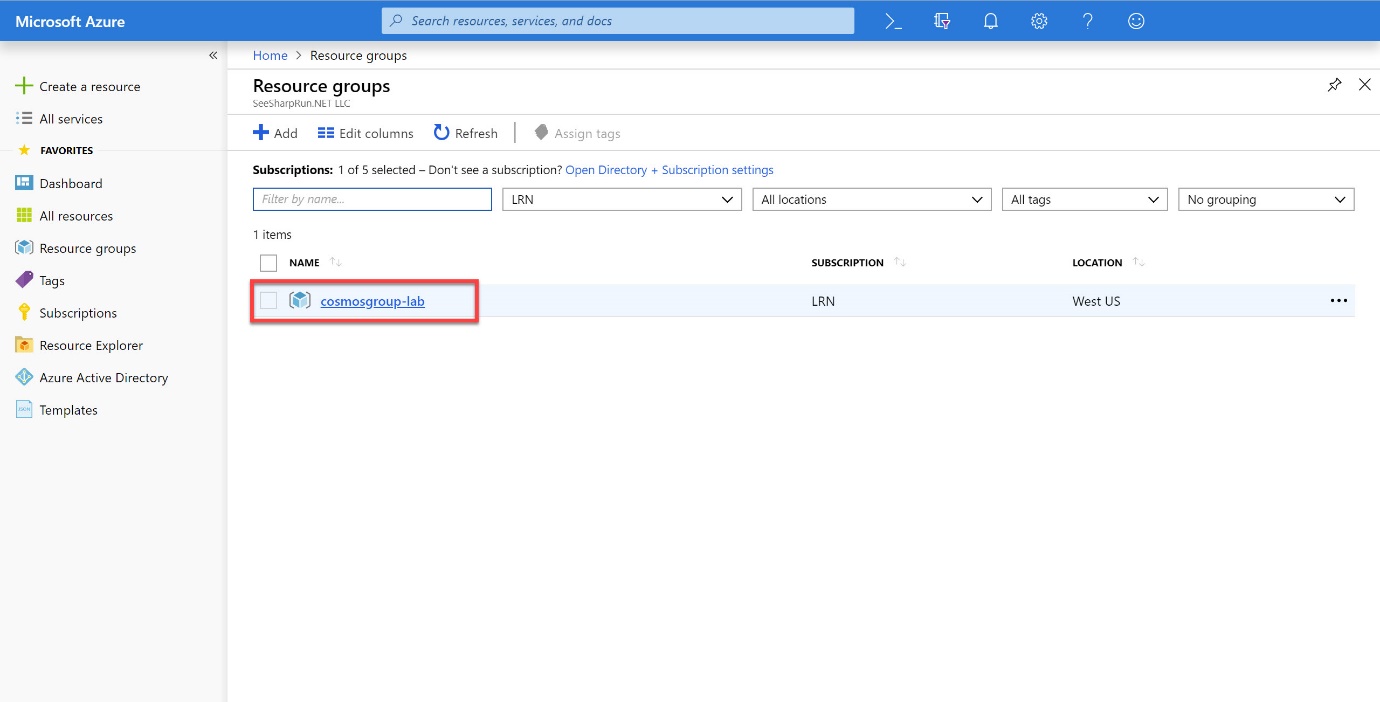
Before you start this lab, you will need to create an Azure Cosmos DB database and collection that you will use throughout the lab. The Python SDK requires credentials to connect to your Azure Cosmos DB account. You will collect and store these credentials for use throughout the lab.

### **Retrieve Account Credentials**

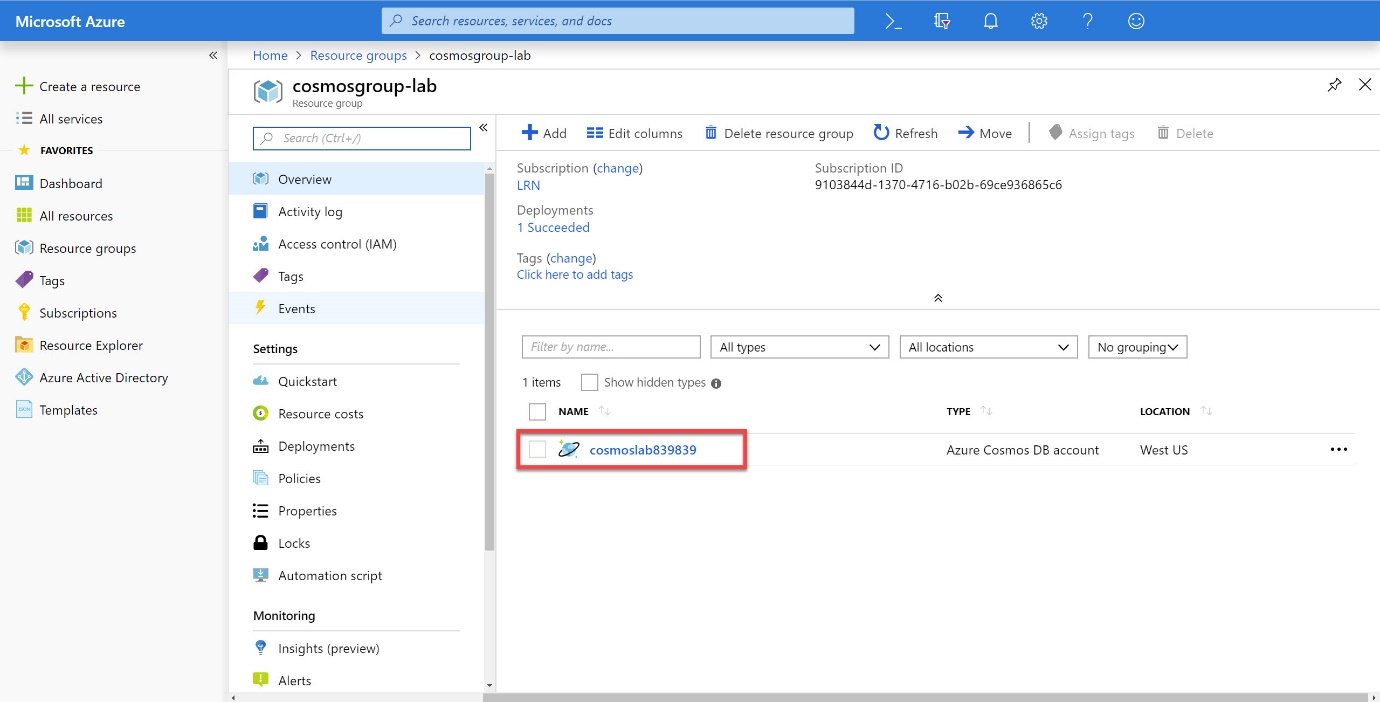
1. On the left side of the portal, click the **Resource groups** link.

[](https://cosmosdb.github.io/labs/media/02-resource_groups.jpg)

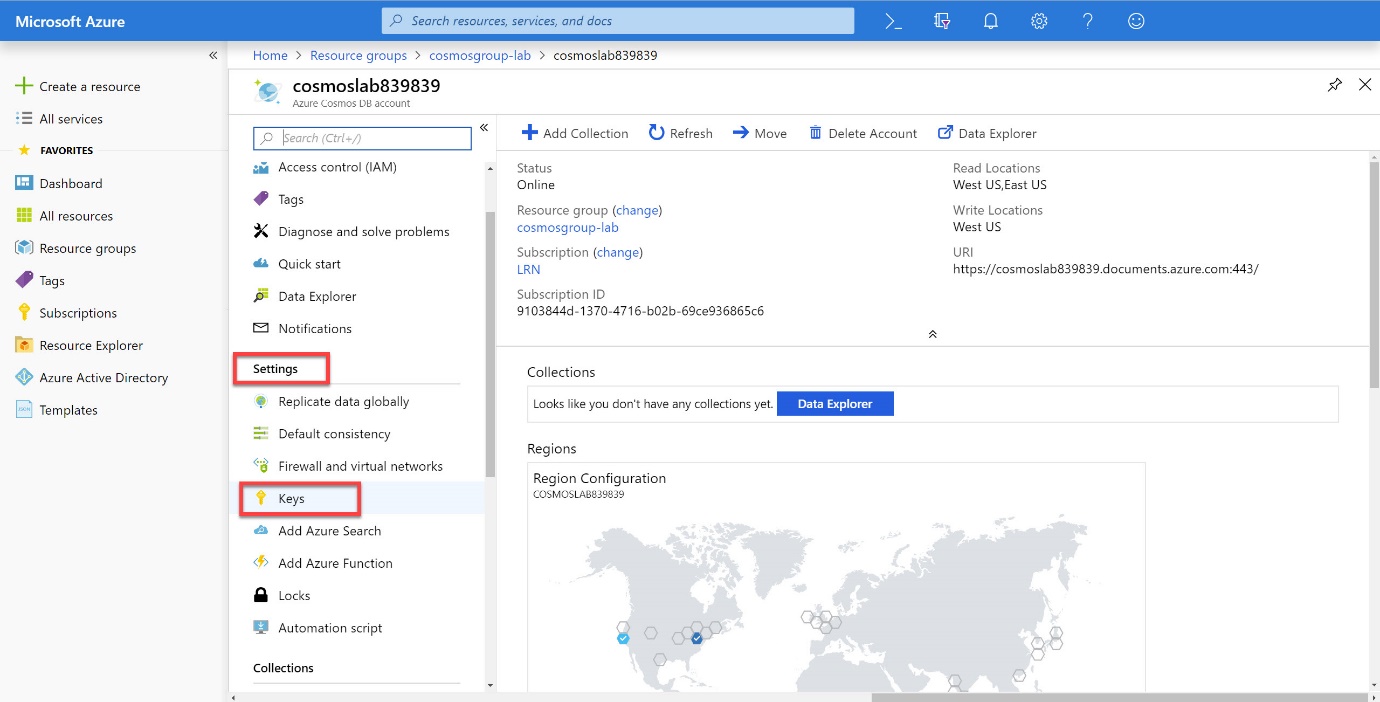
1. In the **Resource groups** blade, locate and select the **cosmosgroup-lab** Resource Group.

[](https://cosmosdb.github.io/labs/media/02-lab_resource_group.jpg)

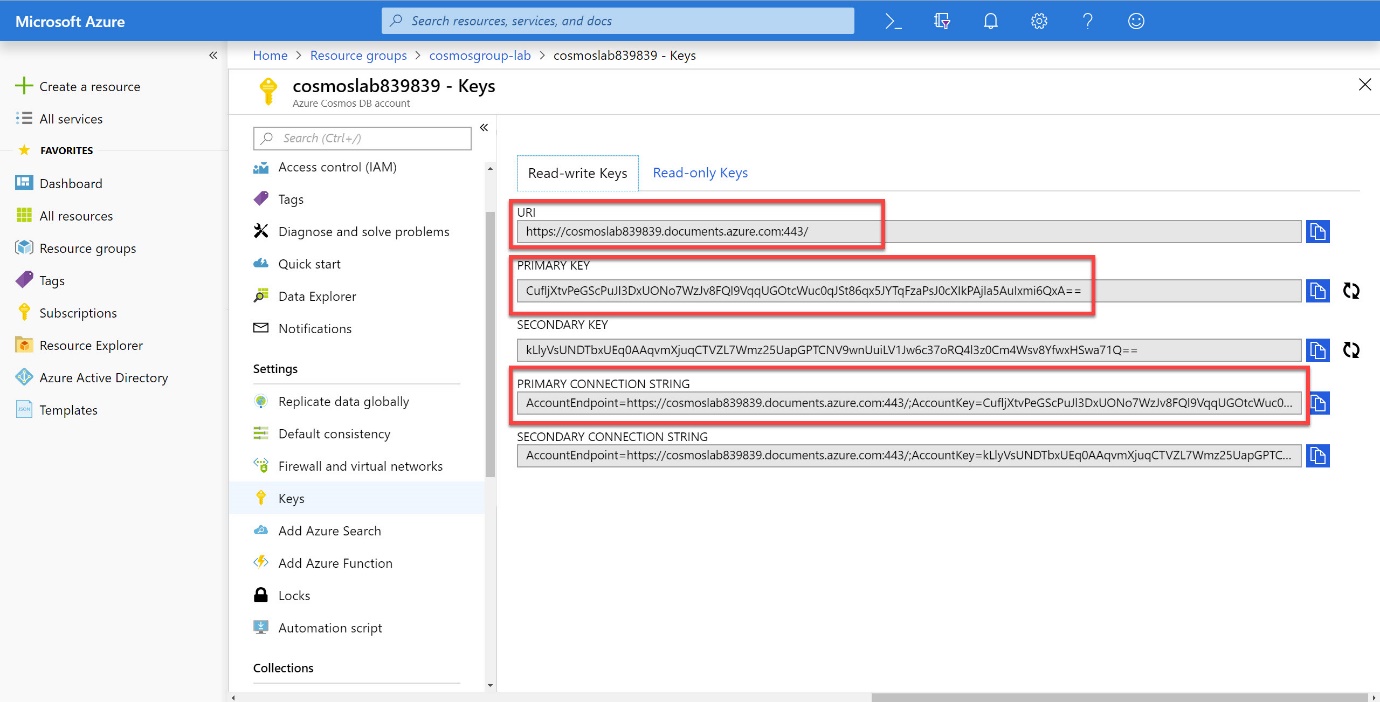
1. In the **cosmosgroup-lab** blade, select the **Azure Cosmos DB** account you recently created.

[](https://cosmosdb.github.io/labs/media/02-cosmos_resource.jpg)

1. In the **Azure Cosmos DB** blade, locate the **Settings** section and click the **Keys** link.

[](https://cosmosdb.github.io/labs/media/02-keys_pane.jpg)

1. In the **Keys** pane, record the values in the **CONNECTION STRING**, **URI** and **PRIMARY KEY** fields. You will use these values later in this lab.

[](https://cosmosdb.github.io/labs/media/02-keys.jpg)

## Exercise 3 Create Containers using the Python SDK

**Task I Setup the Python environment**

Install python 3.6 on your Windows PC   
You must install python on your PC so that it is in your path and you can execute the python command line (REPL) and the pip command from an ordinary command line shell. Test this to avoid problems later.

1. Create a directory to house your project
2. Open a command window in the new directory
3. Type the following to create a virtual python environment in your working directory

D:\src\PythonProjects\CosmosLabs>virtualenv pyenv3

It is possible you may have to install virtualenv with

> pip install virtualenv

1. Activate the environment:

.\pyenv3\Scripts\activate

(pyenv3) D:\src\PythonProjects\CosmosLabs>

You will notice the name of the environment in brackets at the command prompt

1. Install the required Cosmos DB libraries

(pyenv3) D:\src\PythonProjects\CosmosLabs> pip install azure-cosmos

Collecting azure-cosmos

…

Successfully installed azure-cosmos-3.0.2…

(pyenv3) D:\src\PythonProjects\CosmosLabs>pip install azure-mgmt-cosmosdb

Collecting azure-mgmt-cosmosdb

…

…

Install a library to generate test data

(pyenv3) D:\src\PythonProjects\CosmosLabs>pip install faker

Collecting faker

…

…

**Task II Creating a Database Client Connection**

Open the project directory with Visual Studio Code (VSC)

(pyenv3) D:\src\PythonProjects\CosmosLabs>code.

Create a simple config.json file to store the Cosmos DB account credentials in

Remember to put config.json in your .gitignore file if you are using git source control

File Edit Selection 
EXPLORER 
OPEN EDITORS 
X Welcome 
pyenv3 
View 
GO 
Debug 
Terminal 
Help 
Welcome - CosmosLabs - Visual Studio Code 
Welcome X 
Visual Studio Code 
Editing evolved 

Grab the endpoint and the keys for your config.json file

Dashboard > cosmoslabsdjb - Keys 
cosmoslabsdjb - Keys 
Azure Cosmos og account 
p search (Ctrl*" 
Overview 
Activity log 
Access control (IAM) 
Tags 
Diagnose and solve problems 
Quick start 
Notifications 
Data Explorer 
Settings 
• Replicate data globally 
Default consistency 
Firewall and virtual networks 
CORS 
Add Azure Search 
x 
Read-write Keys 
Read-only Keys 
https://cosmoslabsdjb.documents.azure.com443/ 
PRIMARY KEY 
WE-3gzm8ze6HfGKOr8sKgYQE1qc4xF336ySGKSzdtXdQaMZ6WXCwMNwJSNdufZpJxVUSDKmLkGtPbFs67pGCz9wz= 
SECONDARY KEY 
XVUW068kvnxOz9dS3n4MDgyAfdRk8qDkvFjSKFSsxb07K9sZSdptJiDDMEyRDfSBsJeFbCR417Gki3LGTOritw-- 
PRIMARY CONNECTION STRING 
AccountEndpoint=https://cosmoslabsdjbdocumentsazurecom:443/;AccountKey=WE39zm8ze6HfGKOresKgYQElqc4xF336y5GKSzdtXdQaMZ6WXCwMNwJSNdufZpJxVUSDKmLkGtPbFs67 — ; 
SECONDARY CONNECTION STRING 
AccountEndpoint2https://cosmoslabsdjbdocumentsazure.com:443/;AccountKeFXVUW068kvnxOz9dS3n4MDgyAfd Rk8qDkvFjSKFSsxb07K9sZSdptJiDDMEyRDfS3sJeFbCR417Gki3LGTOritw— — ; 

Put your values from the portal in the config.json file in VSC and save them

{

"endPoint": "https://cosmoslabsdjb.documents.azure.com:443/",

"key": "AZ3fh8Znkkgdf8shf7fHHJJ7r49BGSOEUIEJHE354mjb54WAfd48f7sd8A847BSGS8ujMiydh98nWomd9FGx0r=="

}

Create a new file called client.py and we will create a database client using the configuration and verify connection.

Copy and paste the following code

import azure.cosmos.cosmos\_client as cosmos\_client

import json

with open('config.json', 'r') as f:

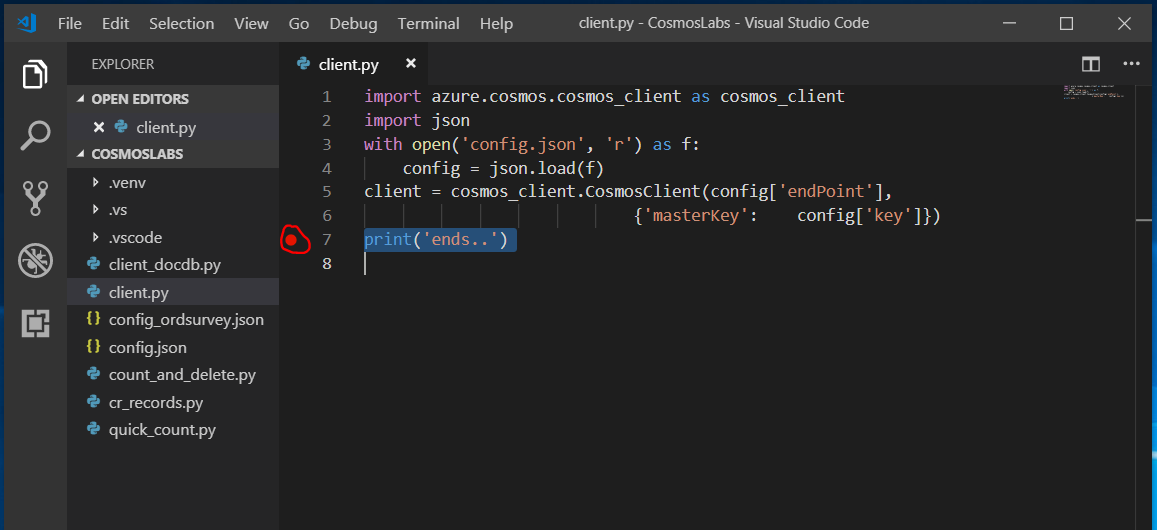
config = json.load(f)

client = cosmos\_client.CosmosClient(config['endPoint'],

{'masterKey': config['key']})

print('ends..')

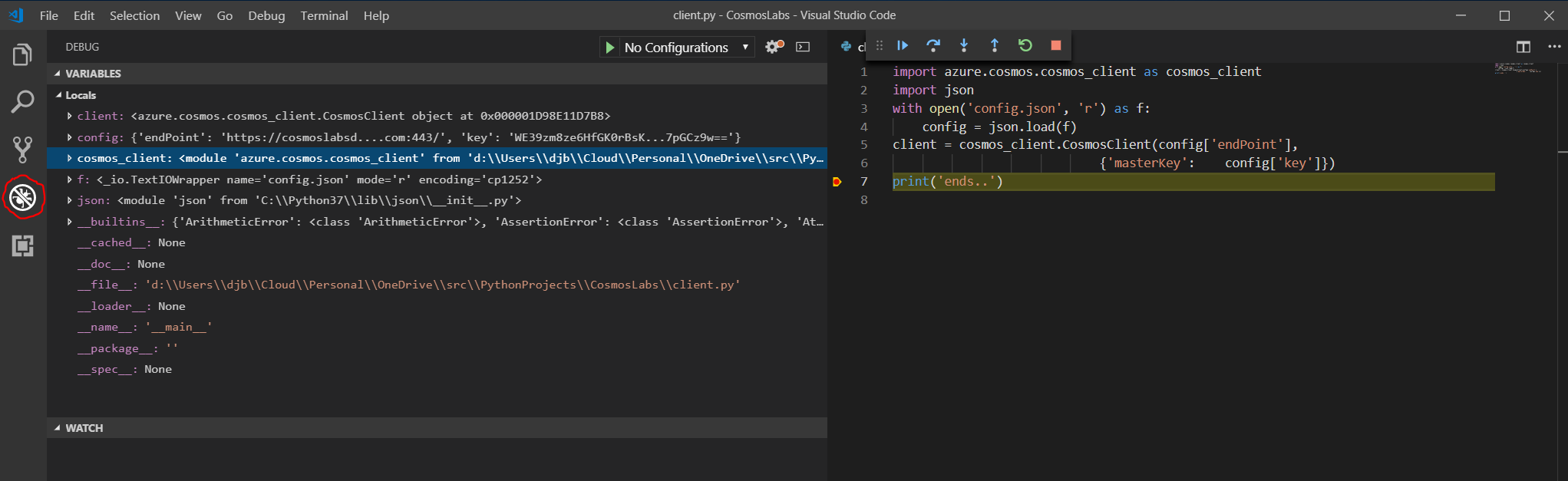
In the code window for the client.py script click in the left hand margin and a red dot will appear. This will set a breakpoint for the python debugger integrated with Visual Studio Code.



Select Debug > Start Debugging from the menu or press F5 to begin debugging. The python interpreter will run your code and stop at the break point

File 
Edit Selection View 
EXPLORER 
OPEN EDITORS 
• client-py 
.vscode 
pyenv3 
client-py 
configjson 
Go 
Debug Terminal Help 
Start Debugging 
Start Without Debugging 
Stop Debugging 
Restart Debugging 
Open Configurations 
Add Configuration.„ 
Steo Over 
• client.py - CosmosLa 
Ctrl+F5 
Iment 
Shift+F5 
Ctrl FS 
t. Docu 

The interpreter will stop at the breakpoint and the debug context should be selected for the left hand panel of Visual Studio Code (VSC).



Press the Continue icon above the code at the top of the right panel to finish the script and exit

**Task III Create a Database using the Python SDK**

In the client.py script add the following line below the client definition in the script.

db = client.CreateDatabase({ 'id': 'EntertainmentDatabase'} )

Add a line following to print out the database resource link for the newly created database

print('Database self link is: {}'.format(db['\_self']))

Your code should look like the following :

import azure.cosmos.cosmos\_client as cosmos\_client

import json

with open('config.json', 'r') as f:

config = json.load(f)

client = cosmos\_client.CosmosClient(config['endPoint'],

{'masterKey': config['key']})

db = client.CreateDatabase({ 'id': 'EntertainmentDatabase'} )

print('Database self link is: {}'.format(db['\_self']))

print('ends..')

Run the code by pressing F5 in Visual Studio Code and verify that a database link is emitted at the Terminal window, below the source pane on the right.

Look in the Azure portal to verify that the database has been created. Select the "Data Explorer" tab from the main page of the Cosmos DB Account you created earlier.

Dashboard cosmoslabsdjb - Data Explorer 
cosmoslabsdjb - Data Explorer 
Cosmos account 
p Search (Ctrl 
Tags 
New Database 
SQL API 
e 
New Collection 
Open Full Screen 
Welcome to Azure Cosmos DB 
Feedback 
Overview 
Activity log 
Access control (IAM) 
Diagnose and solve problems 
Quick start 
Notifications 
Data Explorer 
EntertainmentDatabase 
Create new or work with existing collection(s). 

Run the script again and verify it fails on the client.CreateDatabase as there is already a database with the id EntertainmentDatabase created.

Have a look at the contents of the db variable in the debugger window.

Tidy up the database creation code so that we catch the exception if a database with the chosen name already exists and read the link for the existing database with the following code:

import azure.cosmos as cosmos

import azure.cosmos.cosmos\_client as cosmos\_client

import json

with open('config.json', 'r') as f:

config = json.load(f)

client = cosmos\_client.CosmosClient(config['endPoint'],

{'masterKey': config['key']})

databaseID = 'EntertainmentDatabase'

try:

db = client.CreateDatabase({ 'id': databaseID} )

except Exception as ex:

if type(ex) == cosmos.errors.HTTPFailure and ex.status\_code == 409:

# database already exists

db = client.ReadDatabase('dbs/' + databaseID )

else:

print("we had a problem")

raise ex

print('Database self link is: {}'.format(db['\_self']))

print('ends..')

Rename the python source file to createDatabase.py and save it.

**Task IV Create a Fixed size collection using the Python SDK**

Azure Cosmos DB containers can be created as fixed or unlimited in Python SDK. You can create unlimited sized collections in the Azure portal, but you can no longer created fixed size collections. Fixed-size containers have a maximum limit of 10 GB and 10,000 RU/s throughput. Throughput is the rate at which the database takes in and processes data. A request unit is a normalized quantity that represents the amount of computation required to serve a request. In Cosmos DB, you reserve a guaranteed amount of throughput on a collection or across the database, measured in RU/s. To learn more, refer to [/docs.microsoft.com/azure/cosmos-db/request-units](https://docs.microsoft.com/en-us/azure/cosmos-db/request-units). You will create a fixed-size container in this task.

Add the following code to a new python source file named createFixedColl.py to create an options dictionary that is used to then create the fixed size collection in the existing database. In the definition for the collection we simply pass a directory object in with a single id : value attribute.

We can then create the collection in the try: block

Set breakpoints as shown before (hint: click to the right of the code until a red dot appears) and examine the values for collection and db objects.

import azure.cosmos as cosmos

import azure.cosmos.cosmos\_client as cosmos\_client

import json

with open('config.json', 'r') as f:

config = json.load(f)

client = cosmos\_client.CosmosClient(config['endPoint'],{'masterKey': config['key']})

databaseID = 'EntertainmentDatabase'

collectionID = 'DefaultCollection'

try:

db = client.CreateDatabase({ 'id': databaseID} )

except Exception as ex:

if type(ex) == cosmos.errors.HTTPFailure and ex.status\_code == 409:

# database already exists

db = client.ReadDatabase('dbs/' + databaseID )

else:

print("we had a problem creating database")

raise ex

print('Database self link is: {}'.format(db['\_self']))

options = {

'offerEnableRUPerMinuteThroughput': True,

'offerVersion': "V2",

'offerThroughput': 400

}

try:

collection = client.CreateContainer(db['\_self'], { 'id': collectionID }, options)

except Exception as ex:

if type(ex) == cosmos.errors.HTTPFailure and ex.status\_code == 409:

# collection already exists

collection = client.ReadContainer('dbs/' + databaseID + '/colls/' + collectionID )

else:

print("we had a problem creating collection")

raise ex

print('Collection self link for {} is: {}'.format(collectionID, collection['\_self']))

print('\nends..')

### createFixedColl.py

Examine the DefaultCollection you have created in the Azure portal - data explorer. Look at the values in EntertainmentDatabase > DefaultCollection > Scale & Settings. Notice the storage Capacity is Fixed - limited to 10GB maximum.

**Task V Create an Unlimited size collection using the Python SDK**

In order to create an unlimited collection we pass a dictionary object with specifiers for the id and the partition key for the collection to the Client.CreateContainer method. Create a new file called CreateUnlimColl.py and add the following code to it.

import azure.cosmos as cosmos

import azure.cosmos.cosmos\_client as cosmos\_client

import json

with open('config.json', 'r') as f:

config = json.load(f)

client = cosmos\_client.CosmosClient(config['endPoint'],{'masterKey': config['key']})

databaseID = 'EntertainmentDatabase'

collectionID = 'CustomCollection'

try:

db = client.CreateDatabase({ 'id': databaseID} )

except Exception as ex:

if type(ex) == cosmos.errors.HTTPFailure and ex.status\_code == 409:

# database already exists

db = client.ReadDatabase('dbs/' + databaseID )

else:

print("we had a problem creating database")

raise ex

print('Database self link is: {}'.format(db['\_self']))

coll = {

"id": collectionID,

"partitionKey": {

"paths": ["/AccountNumber"],

"kind": "Hash"

}

}

options = {

'offerEnableRUPerMinuteThroughput': True,

'offerVersion': "V2",

'offerThroughput': 400

}

try:

collection = client.CreateContainer(db['\_self'], coll, options)

except Exception as ex:

if type(ex) == cosmos.errors.HTTPFailure and ex.status\_code == 409:

# collection already exists

collection = client.ReadContainer('dbs/' + databaseID + '/colls/' + collectionID )

else:

print("we had a problem creating collection")

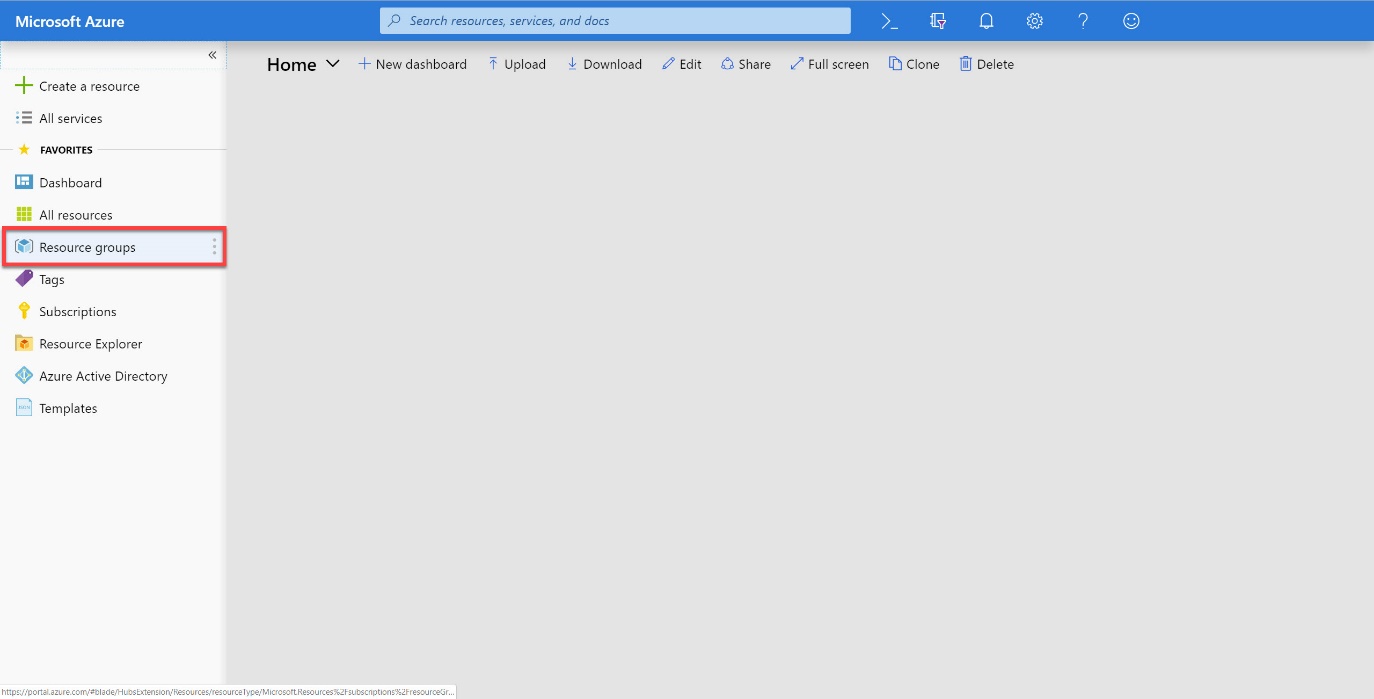
raise ex

print('Collection self link for {} is: {}'.format(collectionID, collection['\_self']))

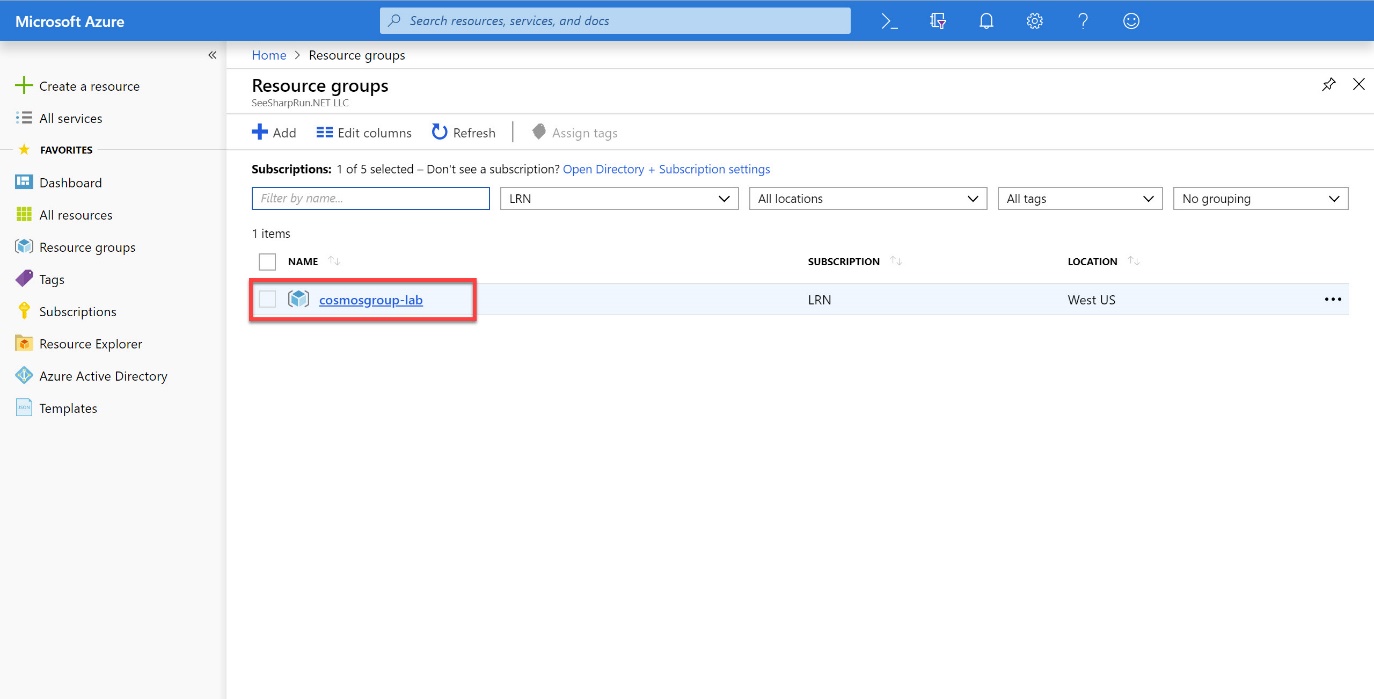
print('\nends..')

**Task VI Observe Newly Created Database and Collections in the Portal**

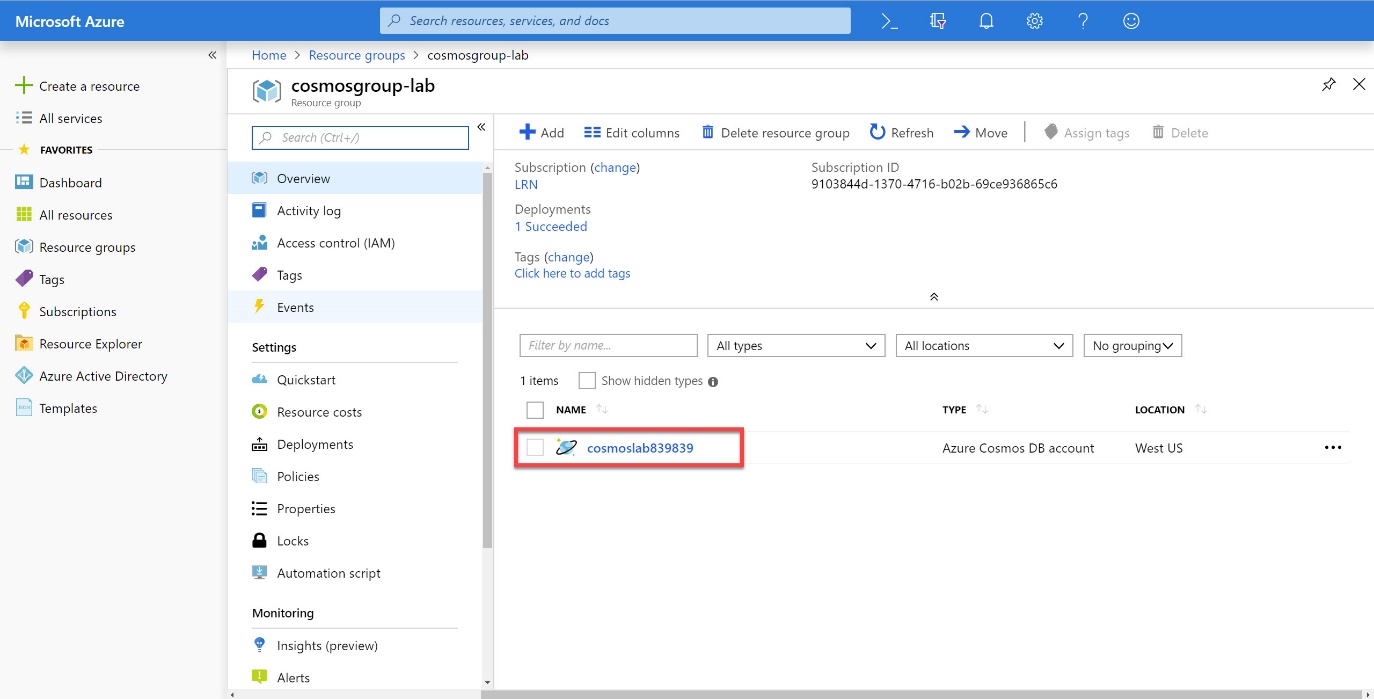
* 1. In a new window, sign in to the **Azure Portal** ([http://portal.azure.com](http://portal.azure.com/)).
  2. On the left side of the portal, click the **Resource groups** link.

[](https://cosmosdb.github.io/labs/media/02-resource_groups.jpg)

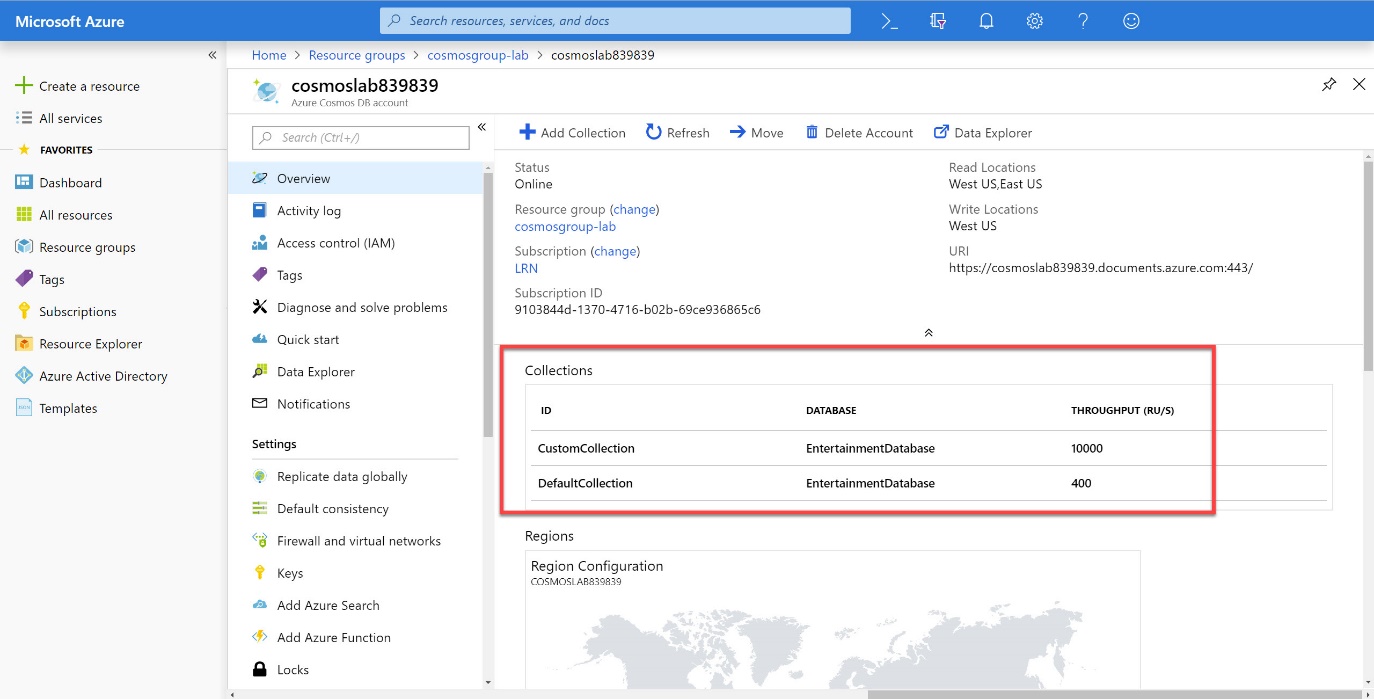
* 1. In the **Resource groups** blade, locate and select the **cosmosgroup-lab** *Resource Group*.

[](https://cosmosdb.github.io/labs/media/02-lab_resource_group.jpg)

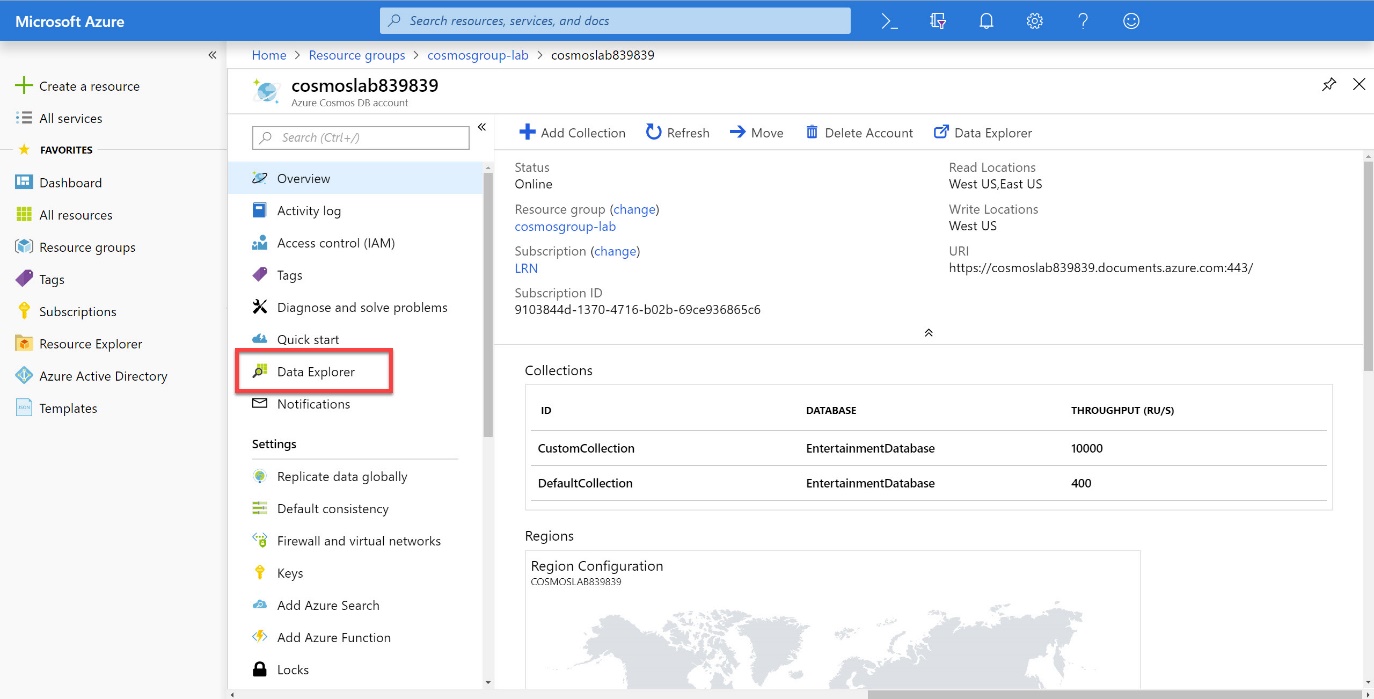
* 1. In the **cosmosgroup-lab** blade, select the **Azure Cosmos DB** account you recently created.

[](https://cosmosdb.github.io/labs/media/02-cosmos_resource.jpg)

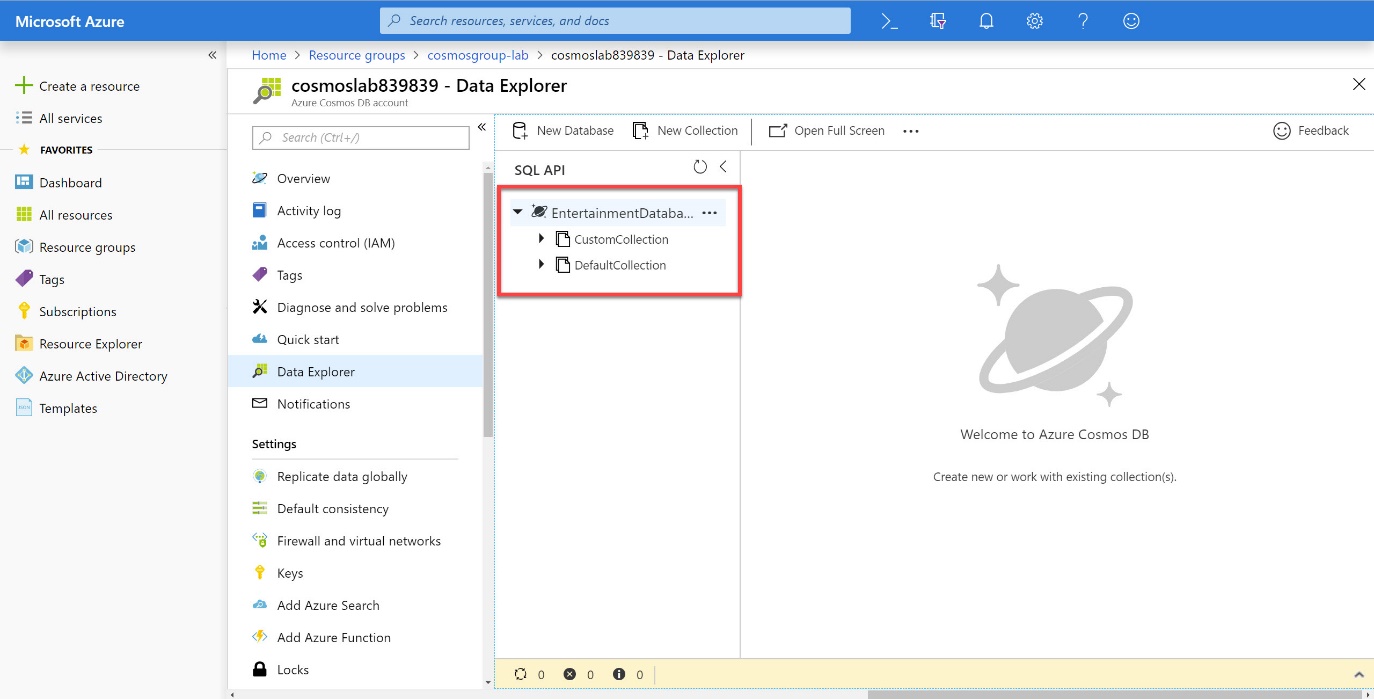
* 1. In the **Azure Cosmos DB** blade, observe the new collections and database displayed in the middle of the blade.

[](https://cosmosdb.github.io/labs/media/02-created_collections.jpg)

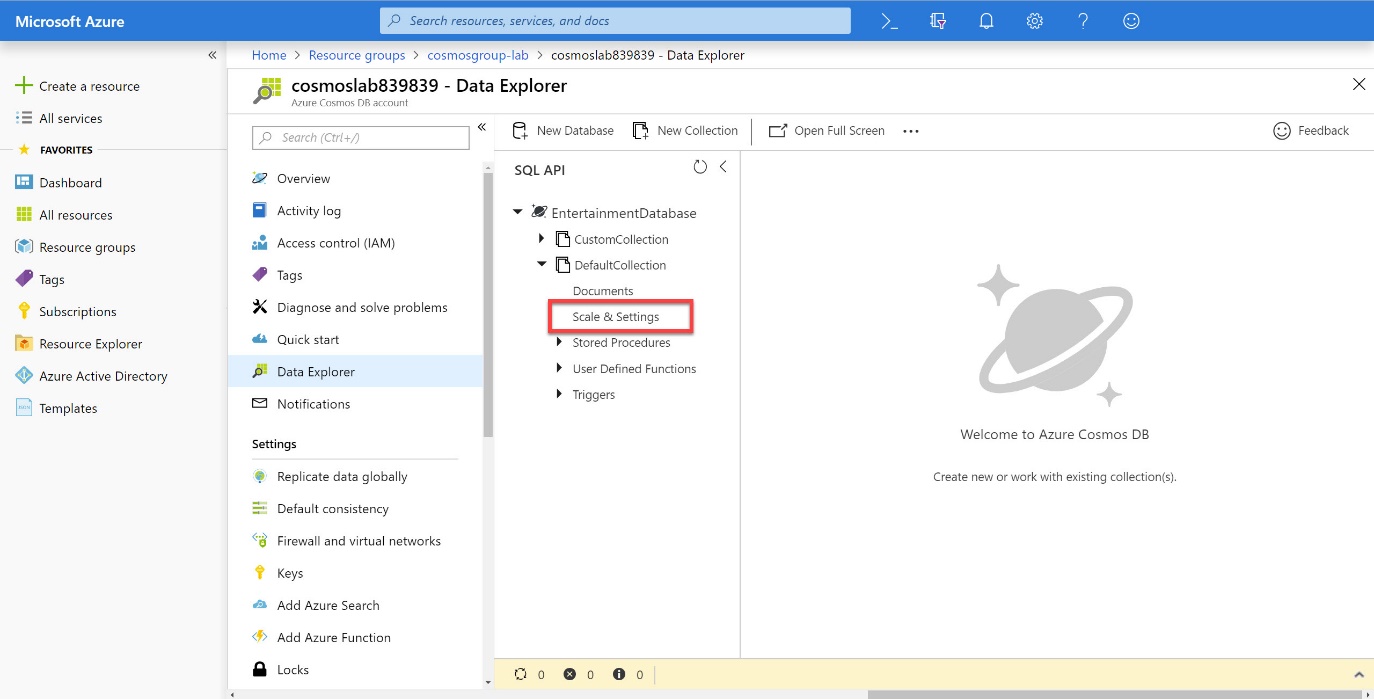
* 1. Locate and click the **Data Explorer** link on the left side of the blade.

[](https://cosmosdb.github.io/labs/media/02-data_explorer_pane.jpg)

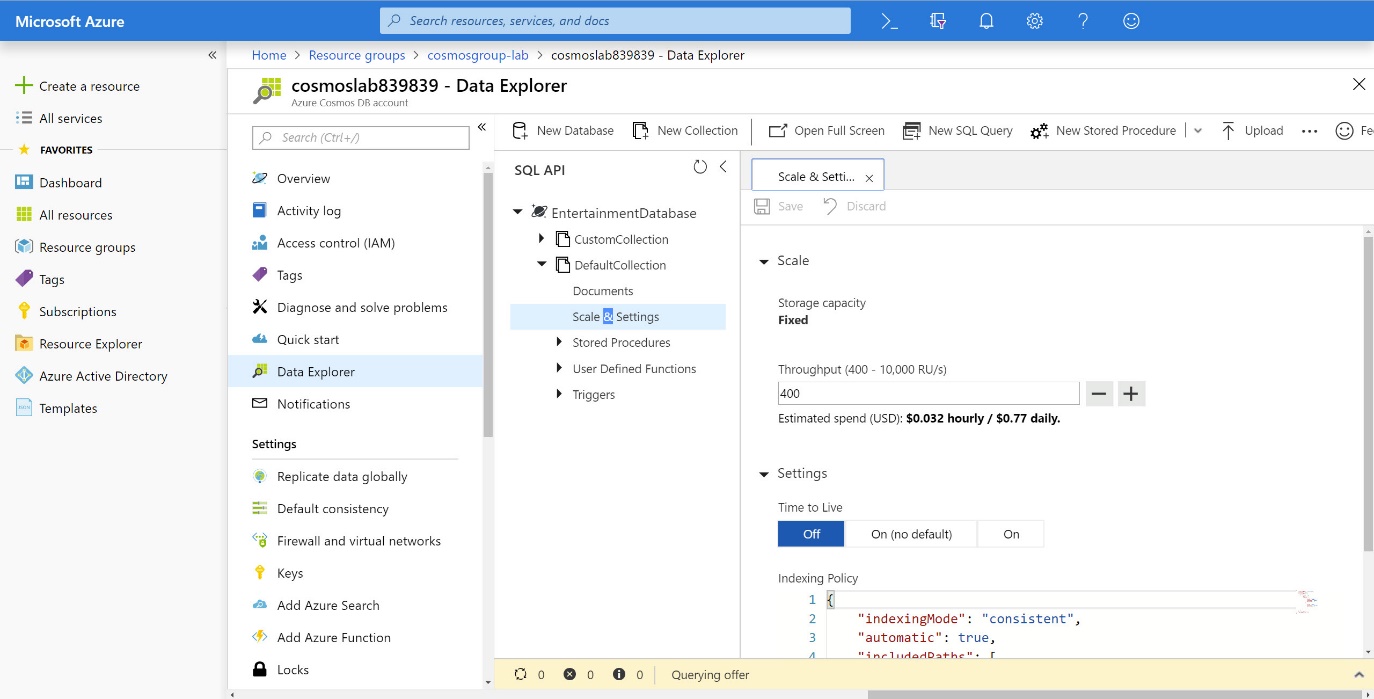
* 1. In the **Data Explorer** section, expand the **EntertainmentDatabase** database node and then observe the collection nodes.

[](https://cosmosdb.github.io/labs/media/02-database_node.jpg)

* 1. Expand the **DefaultCollection** node. Within the node, click the **Scale & Settings** link.

[](https://cosmosdb.github.io/labs/media/02-scale_and_settings.jpg)

* 1. Observe the following properties of the collection:
     + Storage Capacity
     + Assigned Throughput
     + Indexing Policy

[](https://cosmosdb.github.io/labs/media/02-fixed_configuration.jpg)

You will quickly notice that this is a fixed-size container that has a limited amount of RU/s. The indexing policy is also interesting as it implements a Hash index on string types and Range index on numeric types.

Index Policy for the collection

{  
 "indexingMode": "consistent",  
 "automatic": true,  
 "includedPaths": [  
 {  
 "path": "/\*",  
 "indexes": [  
 {  
 "kind": "Range",  
 "dataType": "Number",  
 "precision": -1  
 },  
 {  
 "kind": "Hash",  
 "dataType": "String",  
 "precision": 3  
 }  
 ]  
 }  
 ],  
 "excludedPaths": []  
 }

* 1. Back in the **Data Explorer** section, expand the **CustomCollection** node. Within the node, click the **Scale & Settings** link.
  2. Observe the following properties of the collection and compare them to the last collection:
     + Storage Capacity
     + Assigned Throughput
     + Partition Key
     + Indexing Policy  
       You configured all of these values when you created this collection using the SDK. You should take time to look at the custom indexing policy you created using the SDK.

{  
 "indexingMode": "consistent",  
 "automatic": true,  
 "includedPaths": [  
 {  
 "path": "/\*",  
 "indexes": [  
 {  
 "kind": "Range",  
 "dataType": "Number",  
 "precision": -1  
 },  
 {  
 "kind": "Range",  
 "dataType": "String",  
 "precision": -1  
 }  
 ]  
 }  
 ],  
 "excludedPaths": []  
 }

* 1. Close your browser window displaying the Azure Portal.

## Exercise 4 Populate Collections with Documents using the Python SDK

OK so now we go and populate the collection with 1000 documents containing one of 3 types of interactions

WatchLiveTelevisionChannel

PurchaseFoodOrBeverage

ViewMap

Populate the partitioned – custom collection using some python code to demonstrate that one can store differing document types within the collections.

Copy the code (3 blocks) below into a file called EntertainmentInt.py

import azure.cosmos as cosmos

import azure.cosmos.cosmos\_client as cosmos\_client

import json

import random

def randListElem(inList):

""" Return a random element from a list supplied as a parameter """

retVal = inList[random.randint(0, len(inList)-1 )]

return(retVal)

def watchLiveTelevisionChannel():

""" Return a random watchLiveTV document as a dict """

channelNames = ['BBC1', 'BBC2', 'ITV1', 'Channel 4', 'BBC News', 'Sky Sports 1',

'Sky Sports 2']

minutesViewed = random.randint(1, 121)

channelName = randListElem(channelNames)

intType = 'Watch live TV'

wltvc = { "channelName" : channelName,

"minutesViewed" : minutesViewed,

"type" : intType}

return wltvc

def purchaseFoodBeverage():

""" Return a random food or beverage purchase as a dictionary """

foodItems = ['hotdog', 'veggie meat pie', 'crisps', 'meat and potato pie', 'peanuts'

]

drinkItems = ['tea', 'Bovril', 'Hot Chocolate', 'Fanta', 'Diet Coke', 'Pepsi',

'Coffee', 'bottled water']

fbTypes = ['food', 'beverage']

fbPrices = {"hotdog" : 0.45, "veggie meat pie" : 0.55, "crisps" : 0.25,

"meat and potato pie" : 0.5, "peanuts" : 0.21 , "tea" : 0.10,

"Bovril" : 0.15, "Hot Chocolate" : 0.15, "Fanta" : 0.2,

"Diet Coke" : 0.2, "Pepsi" : 0.2, "Coffee" : 0.12,

"bottled water" : 0.18 }

fbType = randListElem(fbTypes)

if fbType == 'food':

fbItem = randListElem(foodItems)

fbUnitPrice = fbPrices[fbItem]

fbQuantity = random.randint(1, 12)

elif fbType == 'beverage':

fbItem = randListElem(drinkItems)

fbUnitPrice = fbPrices[fbItem]

fbQuantity = random.randint(1, 12)

fbTotalPrice = fbQuantity \* fbUnitPrice

return(

{"type" : fbType,

"item" : fbItem,

"unitPrice" : fbUnitPrice,

"quantity" : fbQuantity,

"totalPrice" : fbTotalPrice }

)

def viewMap():

""" Return a random map viewing event"""

return(

{"type" : "viewMap",

"minutesViewed" : random.randint(1, 25) }

)

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)

client = cosmos\_client.CosmosClient(config['endPoint'],

{'masterKey': config['key']})

databaseID = 'EntertainmentDatabase'

collectionID = 'DefaultCollection'

database\_link = 'dbs/' + databaseID

collection\_link = database\_link + '/colls/' + collectionID

NofDocsToCreate = 1000

for cnt in range(NofDocsToCreate +1):

# Create a random entertainment event instance and persist it to Cosmos

entTypes = ["viewMap", "beverage", "food", "Watch live TV"]

entInstType = randListElem(entTypes)

if entInstType == "viewMap":

entInst = viewMap()

elif entInstType == "food" or entInstType == "beverage":

entInst = purchaseFoodBeverage()

else:

entInst = watchLiveTelevisionChannel()

client.CreateItem(collection\_link, entInst)

print('.', end = '')

print("\nWritten {0} items into {1} collection".format(cnt, collectionID))

Once you have copied the code into EntertainmentInt.py execute the script using the Visual Studio Code F5 key and the program will create and insert 1000 documents into the collection.

Go to the Azure Portal and look at the documents in the Data Explorer > CustomCollection > Documents tab. Note that the collection contains document of different types with different attributes. This is fine in CosmosDB collections.

Note also that every document has an id field populated by a globally unique user id (GUID). CosmosDB will automatically populate this field – it must be unique for each document and you can specify a value for it in your code (but you need to make sure it is a unique identifier).

## Exercise 5 Benchmark A Simple Collection using a .NET Core Application

In the next part of this lab, you will compare various partition key choices for a large dataset using a special benchmarking tool available on GitHub.com. First, you will learn how to use the benchmarking tool using a simple collection and partition key.

### **Clone Existing .NET Core Project**

On your local machine, create a new folder that will be used to contain the content of your new .NET Core project.

In the new folder, right-click the folder and select the **Open with Code** menu option.

Alternatively, you can run a command prompt in your current directory and execute the code . command.

In the Visual Studio Code window that appears, right-click the **Explorer** pane and select the **Open in Command Prompt** menu option.

In the open terminal pane, enter and execute the following command:

git clone https://github.com/seesharprun/cosmos-benchmark.git .

This command will create a copy of a .NET Core project located on GitHub (<https://github.com/seesharprun/cosmos-benchmark>) in your local folder.

Visual Studio Code will most likely prompt you to install various extensions related to **.NET Core** or **Azure Cosmos DB** development. None of these extensions are required to complete the labs.

In the terminal pane, enter and execute the following command:

dotnet restore

This command will restore all packages specified as dependencies in the project.

1. In the terminal pane, enter and execute the following command:

dotnet build

This command will build the project.

1. Click the **🗙** symbol to close the terminal pane.
2. Observe the **Program.cs** and **benchmark.csproj** files created by the .NET Core CLI.
3. Double-click the **sample.json** link in the **Explorer** pane to open the file in the editor.
4. Observe the sample JSON file

This file will show you a sample of the types of JSON documents that will be uploaded to your collection. Pay close attention to the **Submit\*** fields, the **DeviceId** field and the **LocationId** field.

### **Update the Application’s Settings**

1. Double-click the **appsettings.json** link in the **Explorer** pane to open the file in the editor.
2. Locate the **/cosmosSettings.endpointUri** JSON path:

"endpointUri": ""

1. Update the **endPointUri** property by setting it’s value to the **URI** value from your Azure Cosmos DB account that you recorded earlier in this lab:

For example, if your **uri** is https://cosmosacct.documents.azure.com:443/,   
your new property will look like this: "endpointUri": "https://cosmosacct.documents.azure.com:443/".

1. Locate the **/cosmosSettings.primaryKey** JSON path:

"primaryKey": ""

Update the **primaryKey** property by setting it’s value to the **PRIMARY KEY** value from your Azure Cosmos DB account that you recorded earlier in this lab:

For example, if your **primary key** is elzirrKCnXlacvh1CRAnQdYVbVLspmYHQyYrhx0PltHi8wn5lHVHFnd1Xm3ad5cn4TUcH4U0MSeHsVykkFPHpQ==, your new property will look like this: "primaryKey": "elzirrKCnXlacvh1CRAnQdYVbVLspmYHQyYrhx0PltHi8wn5lHVHFnd1Xm3ad5cn4TUcH4U0MSeHsVykkFPHpQ==".

### **Configure a Simple Collection for Benchmarking**

Double-click the **appsettings.json** link in the **Explorer** pane to open the file in the editor.

Locate the **/collectionSettings** JSON path:

"collectionSettings": [],

Update the **collectionSettings** property by setting it’s value to the following array of JSON objects:

"collectionSettings": [

{

"id": "CollectionWithHourKey",

"throughput": 10000,

"partitionKeys": [ "/SubmitHour" ]

}

],

The object above will instruct the benchmark tool to create a single collection and set it’s throughput and partition key to the specified values. For this simple demo, we will use the **hour** when an IoT device recording was submitted as our partition key.

| Collection Name | Throughput | Partition Key |
| --- | --- | --- |
| CollectionWithHourKey | 10000 | /SubmitHour |

Save all of your open editor tabs.

### **Run the Benchmark Application**

1. In the Visual Studio Code window, right-click the **Explorer** pane and select the **Open in Command Prompt** menu option.
2. In the open terminal pane, enter and execute the following command:

dotnet run

1. Observe the results of the application’s execution. Your results should look very similar to the code sample below:

DocumentDBBenchmark starting...

Database Validated: dbs/MOEFAA==/

Collection Validated: dbs/MOEFAA==/colls/MOEFAN6FoQU=/

Summary:

---------------------------------------------------------------------

Endpoint: https://cosmosacct.documents.azure.com/

Database IoTDeviceData

Collection CollectionWithHourKey

Partition Key: /SubmitHour

Throughput: 10000 Request Units per Second (RU/s)

Insert Operation: 100 Tasks Inserting 1000 Documents Total

---------------------------------------------------------------------

Starting Inserts with 100 tasks

Inserted 1000 docs @ 997 writes/s, 7220 RU/s (19B max monthly 1KB reads)

Summary:

---------------------------------------------------------------------

Total Time Elapsed: 00:00:01.0047125

Inserted 1000 docs @ 995 writes/s, 7209 RU/s (19B max monthly 1KB reads)

---------------------------------------------------------------------

The benchmark tool tells you how long it takes to write a specific number of documents to your collection. You also get useful metadata such as the amount of **RU/s** being used and the total execution time. We are not tuning our partition key choice quite yet, we are simply learning to use the tool.

1. Press the **ENTER** key to complete the execution of the console application.

### **Update the Application’s Settings**

Double-click the **appsettings.json** link in the **Explorer** pane to open the file in the editor.

Locate the **/cosmosSettings.numberOfDocumentsToInsert** JSON path:

"numberOfDocumentsToInsert": 1000

Update the **numberOfDocumentsToInsert** property by setting it’s value to **50,000**:

"numberOfDocumentsToInsert": 50000

Save all of your open editor tabs.

### **Run the Benchmark Application**

1. In the Visual Studio Code window, right-click the **Explorer** pane and select the **Open in Command Prompt** menu option.
2. In the open terminal pane, enter and execute the following command:

dotnet run

1. Observe the results of the application’s execution.
2. Observe the amount of time required to import multiple records.
3. Press the **ENTER** key to complete the execution of the console application.

## Benchmark Various Partition Key Choices using a .NET Core Application

Now you will use multiple collections and partition key options to compare various strategies for partitioning a large dataset.

### **Configure Multiple Collections for Benchmarking**

1. Double-click the **appsettings.json** link in the **Explorer** pane to open the file in the editor.
2. Locate the **/collectionSettings** JSON path:

"collectionSettings": [],

Update the **collectionSettings** property by setting it’s value to the following array of JSON objects:

"collectionSettings": [

{

"id": "CollectionWithMinuteKey",

"throughput": 10000,

"partitionKeys": [ "/SubmitMinute" ]

},

{

"id": "CollectionWithDeviceKey",

"throughput": 10000,

"partitionKeys": [ "/DeviceId" ]

}

],

The object above will instruct the benchmark tool to create multiple collections and set their throughput and partition key to the specified values. For this demo, we will compare the results using each partition key.

| Collection Name | Throughput | Partition Key |
| --- | --- | --- |
| CollectionWithMinuteKey | 10000 | /SubmitMinute |
| CollectionWithDeviceKey | 10000 | /DeviceId |

1. Save all of your open editor tabs.

### **Run the Benchmark Application**

1. In the Visual Studio Code window, right-click the **Explorer** pane and select the **Open in Command Prompt** menu option.
2. In the open terminal pane, enter and execute the following command:

dotnet run

1. Observe the results of the application’s execution.

The timestamp on these IoT records is based on the time when the record was created. We submit the records as soon as they are created so there’s very little latency between the client and server timestamp. Most of the records being submitted will be within the same minute so they share the same **SubmitMinute** partition key. This will cause a hot partition key and can constraint throughput. In this context, a hot partition key refers to when requests to the same partition key exceed the provisioned throughput and are rate-limited. A hot partition key causes high volumes of data to be stored within the same partition. Such uneven distribution is inefficient. In this demo, you should expect a total time of >20 seconds.

---------------------------------------------------------------------

Collection CollectionWithMinuteKey

Partition Key: /SubmitMinute

Total Time Elapsed: 00:00:57.4233616

Inserted 50000 docs @ 871 writes/s, 6304 RU/s (16B max monthly 1KB reads)

---------------------------------------------------------------------

The **SubmitMinute** partition key will most likely take longer to execute than the **DeviceId** partition key. Using the **DeviceId**partition key creates a more even distribution of requests across your various partition keys. Because of this behavior, you should notice drastically improved performance.

---------------------------------------------------------------------

Collection CollectionWithDeviceKey

Partition Key: /DeviceId

Total Time Elapsed: 00:00:27.2769234

Inserted 50000 docs @ 1833 writes/s, 13272 RU/s (34B max monthly 1KB reads)

---------------------------------------------------------------------

1. Compare the RU/s and total time for both collections.
2. Press the **ENTER** key to complete the execution of the console application.

### **Observe the New Collections and Database in the Azure Portal**

1. Return to the **Azure Portal** ([http://portal.azure.com](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 **IoTDeviceData** database node and then observe the various collection nodes.
7. Expand the **CollectionWithDeviceKey** node. Within the node, click the **Scale & Settings** link.
8. Observe the following properties of the collection:
9. Storage Capacity
10. Assigned Throughput
11. Indexing Policy
12. Click the **New SQL Query** button at the top of the **Data Explorer** section.
13. In the query tab, replace the contents of the query editor with the following SQL query:

SELECT VALUE COUNT(1) FROM recordings

1. Click the **Execute Query** button in the query tab to run the query.
2. In the **Results** pane, observe the results of your query.
3. Back in the **Data Explorer** section, right-click the **IoTDeviceData** database node and select the **Delete Database** option.

Since you created multiple collections in this database with high throughput, it makes sense to dispose of the database immediately to minimize your Azure subscription consumption.

1. In the **Delete Database** popup enter the name of the database (**IoTDeviceData and EntertainmentDatabase**) in the field and then press the **OK** button.
2. Close your browser application.