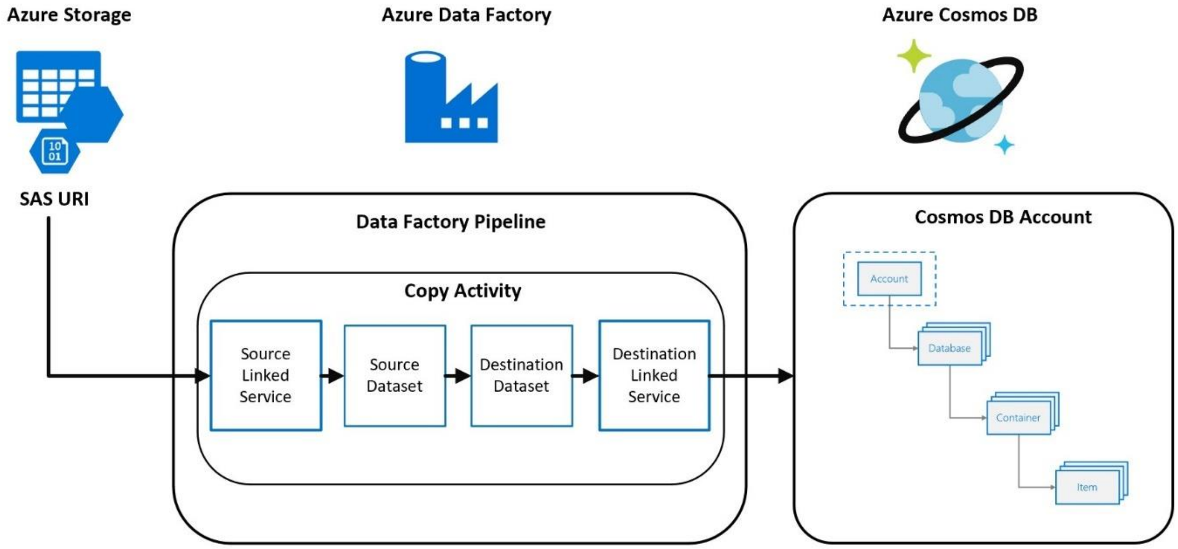
Azure Cosmos DB Lab

Prerequisites

- Microsoft Azure subscription
- Resource Group to deploy Azure services
- Permissions to create the following resource
    - Cosmos DB
    - Data Factory

Reference Architecture

Below is the architecture of the Cosmos DB Lab.



Cosmos DB Lab Guides

It is recommended to complete the labs in the order specified below:

- Lab 1: Importing Data into Azure Cosmos DB with Azure Data Factory
- Lab 2: Querying in Azure Cosmos DB
- Lab 3: Indexing in Azure Cosmos DB

Azure Cosmos DB Lab

Lab 1: Importing data into Azure Cosmos DB with Azure Data Factory

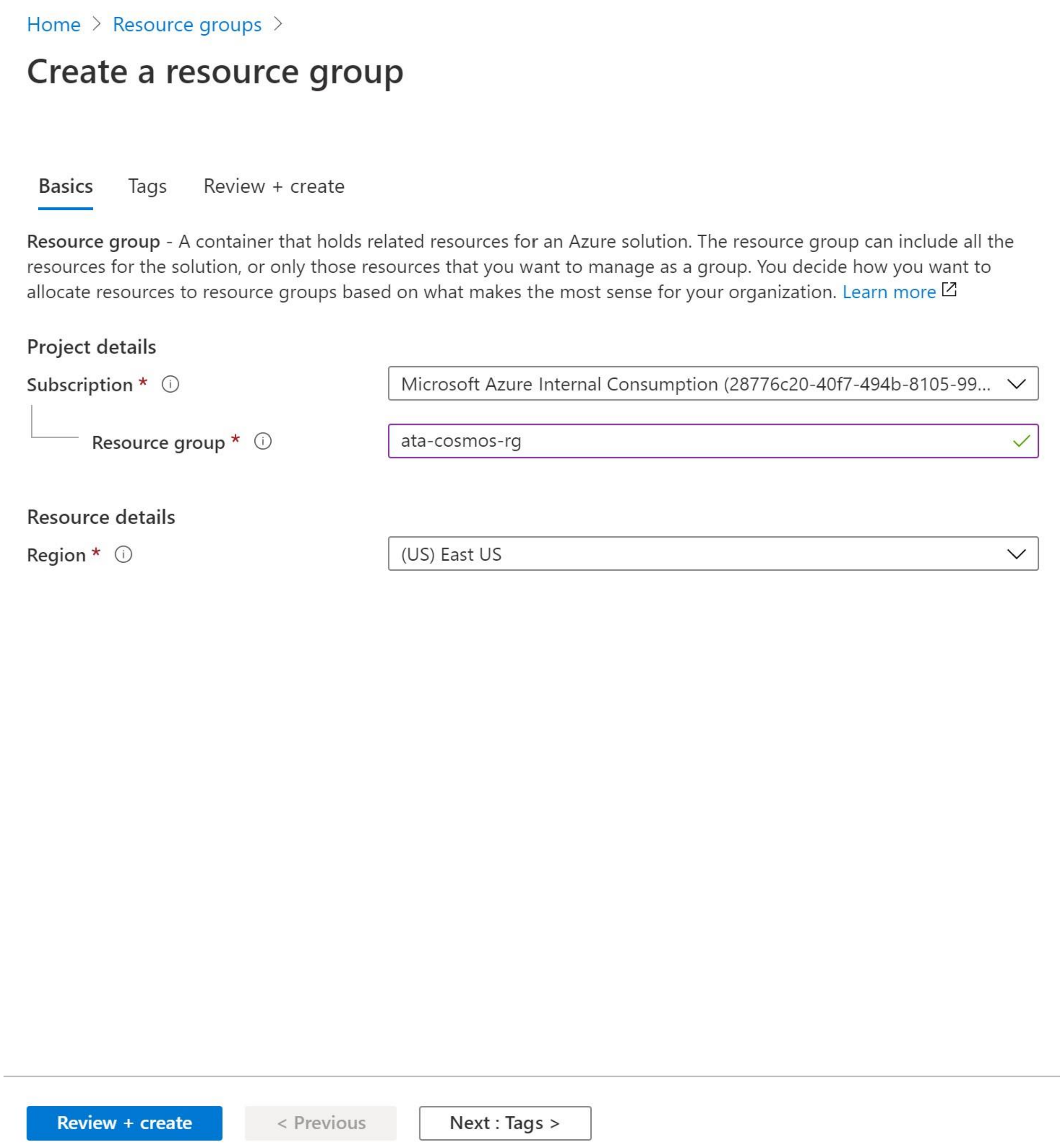
Prerequisites

- Microsoft Azure subscription
- Resource Group to deploy Azure services
- Permissions to create the following resource
  - Cosmos DB
  - Data Factory

Step 1: Create a Resource Group

1. In a new browser window, sign in to the Azure portal.
2. In the Azure Portal, search for **Resource Groups**.
3. Click on the **Add** button.
4. Fill out the **Basics** tab as follows:

- **Subscription:** Choose your subscription
- **Resource group:** Provide a unique name like 'ata-cosmos-<YourName>-rg'
- **Region:** East US



Create a resource group

Resource group - A container that holds related resources for an Azure solution. The resource group can include all the resources for the solution, or only those resources that you want to manage as a group. You decide how you want to allocate resources to resource groups based on what makes the most sense for your organization. Learn more

Project details

Subscription * : Microsoft Azure Internal Consumption (28776c20-40f7-494b-8105-99...

Resource group * : ata-cosmos-rg

Resource details

Region * : (US) East US

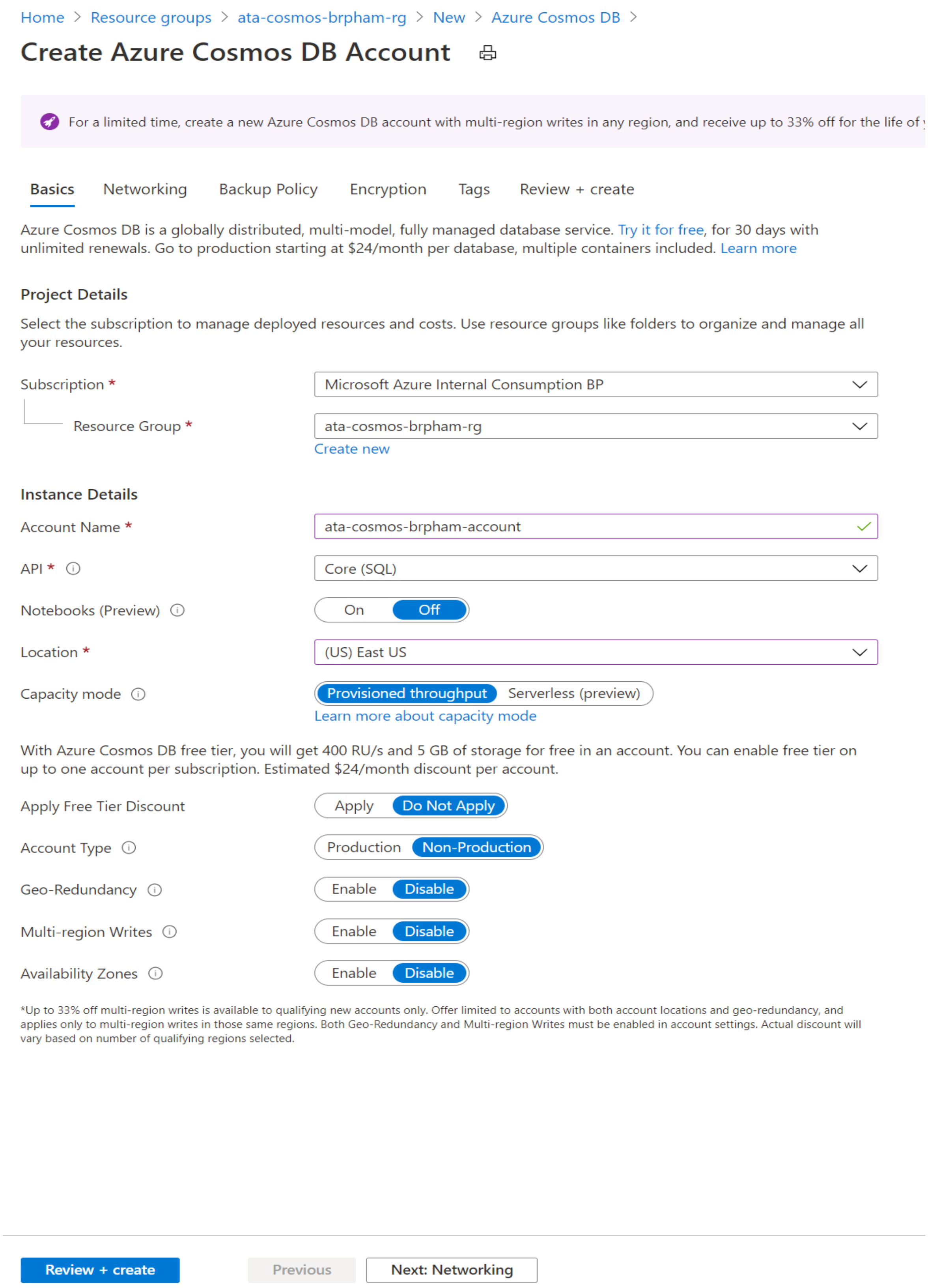
4. Click the **Next: Review + Create** button
5. Click the **Create** button

Step 2: Create an Azure Cosmos DB Account

1. In the Azure Portal, search for **Azure Cosmos DB**
2. Click on the **Create Azure Cosmos DB Account** button
3. Fill out the **Basics** tab as follows:

- **Subscription:** Choose your subscription
- **Resource group:** Select the Resource Group you created for this lab
- **Account name:** Choose a unique name for the Cosmos DB Account: ata-cosmos-<YourName>-account

- **API:** Core (SQL)
- **Location:** (US) East US



Home > Resource groups > ata-cosmos-brpham-rg > New > Azure Cosmos DB >

Create Azure Cosmos DB Account

For a limited time, create a new Azure Cosmos DB account with multi-region writes in any region, and receive up to 33% off for the life of

Azure Cosmos DB is a globally distributed, multi-model, fully managed database service. Try it for free, for 30 days with unlimited renewals. Go to production starting at $24/month per database, multiple containers included. Learn more

Project Details

Select the subscription to manage deployed resources and costs. Use resource groups like folders to organize and manage all your resources.

Subscription *: Microsoft Azure Internal Consumption BP

Resource Group *: ata-cosmos-brpham-rg

Create new

Instance Details

Account Name *: ata-cosmos-brpham-account ✓

API *: Core (SQL)

Notebooks (Preview): On [✓ Off]

Location *: (US) East US

Capacity mode: [✓ Provisioned throughput] Serverless (preview)

Learn more about capacity mode

With Azure Cosmos DB free tier, you will get 400 RU/s and 5 GB of storage for free in an account. You can enable free tier on up to one account per subscription. Estimated $24/month discount per account.

Apply Free Tier Discount: Apply [✓ Do Not Apply]

Account Type: Production [✓ Non-Production]

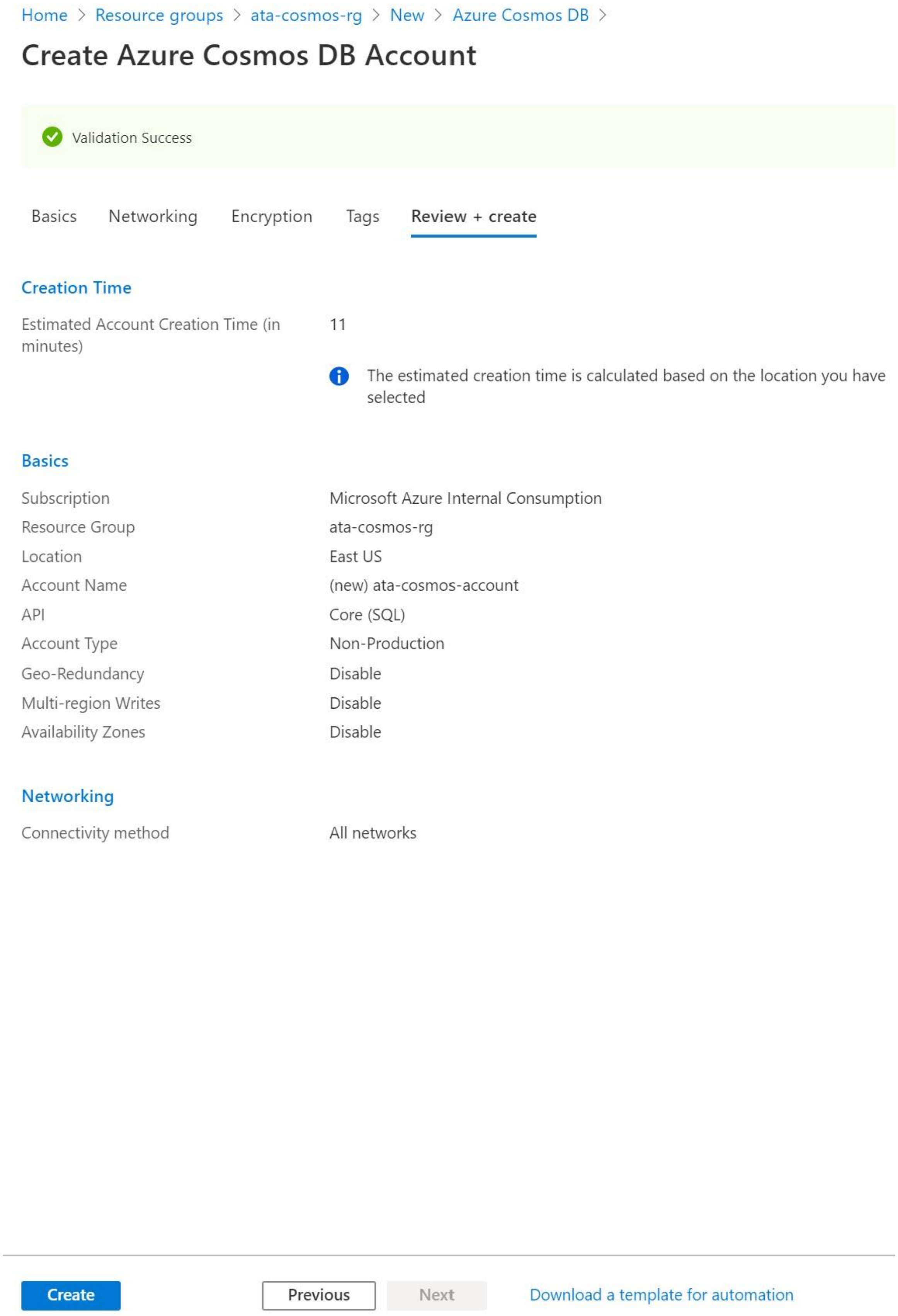
Geo-Redundancy: Enable [✓ Disable]

Multi-region Writes: Enable [✓ Disable]

Availability Zones: Enable [✓ Disable]

*Up to 33% off multi-region writes is available to qualifying new accounts only. Offer limited to accounts with both account locations and geo-redundancy, and applies only to multi-region writes in those same regions. Both Geo-Redundancy and Multi-region Writes must be enabled in account settings. Actual discount will vary based on number of qualifying regions selected.

4. Leave the rest of the fields as default and click the **Review + create** button.


Home > Resource groups > ata-cosmos-rg > New > Azure Cosmos DB >
Basics
Networking
Encryption
Tags
Review + create
Create
Previous
Next
Download a template for automation

Create Azure Cosmos DB Account

✓ Validation Success

Creation Time

Estimated Account Creation Time (in minutes)	11

ⓘ The estimated creation time is calculated based on the location you have selected

Basics

Subscription	Microsoft Azure Internal Consumption
Resource Group	ata-cosmos-rg
Location	East US
Account Name	(new) ata-cosmos-account
API	Core (SQL)
Account Type	Non-Production
Geo-Redundancy	Disable
Multi-region Writes	Disable
Availability Zones	Disable

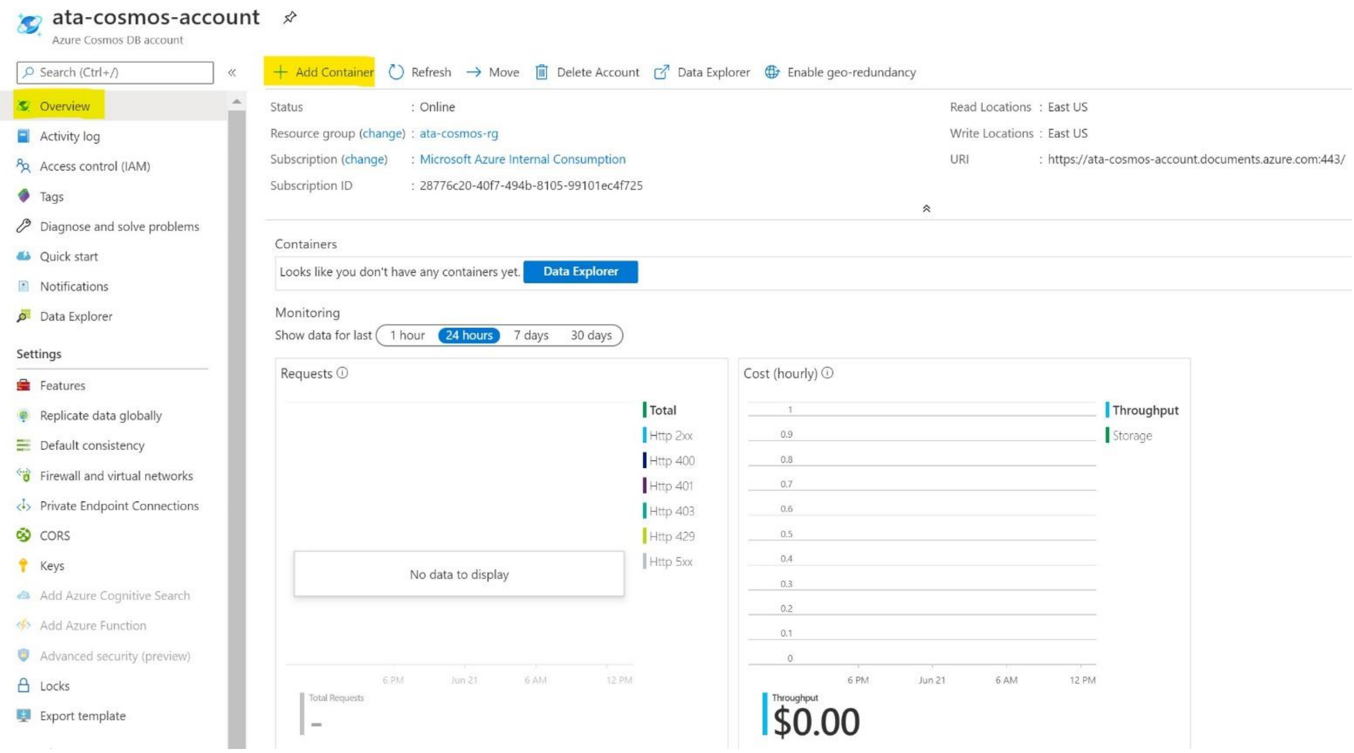
Networking

Connectivity method	All networks

5. Click the **Create** button.

Step 3: Create a container

1. In the Azure Portal, search for the Cosmos DB Account that was created for the lab.
2. In the Azure Cosmos DB blade, locate and select the **Overview** link on the left side of the blade. At the top select the **Add Container** button.

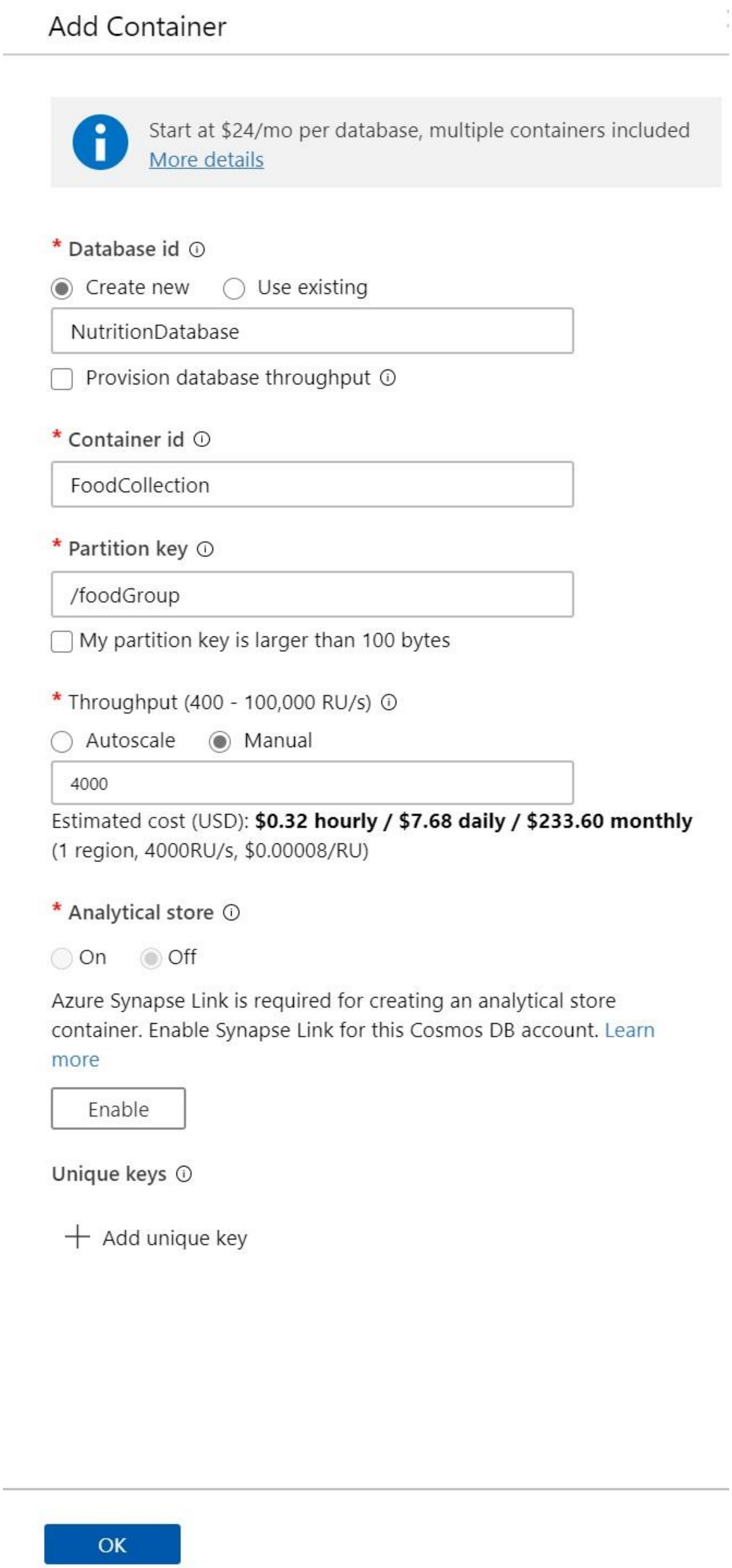


Status : Online
Resource group (change) : ata-cosmos-rg
Subscription (change) : Microsoft Azure Internal Consumption
Subscription ID : 28776c20-40f7-494b-8105-99101ec4f725
Read Locations : East US
Write Locations : East US
URI : https://ata-cosmos-account.documents.azure.com:443/

Looks like you don't have any containers yet.

3. In the **Add Container** popup, fill out the following fields and click the **OK** button.

- **Database id:** Select the **Create new** option and enter the value **NutritionDatabase**
- **Provision database throughput:** Do not check this option.
- **Container id:** FoodCollection
- **Partition key:** /foodGroup
- **My partition key is larger than 100 bytes:** Do not check this option.
- **Throughput (400 - 100,000 RU/s) More information:** Select **Manual** and enter the value 4000.



Add Container

Start at $24/mo per database, multiple containers included
More details

* Database id ⓘ

● Create new ○ Use existing

NutritionDatabase

□ Provision database throughput ⓘ

* Container id ⓘ

FoodCollection

* Partition key ⓘ

/foodGroup

□ My partition key is larger than 100 bytes

* Throughput (400 - 100,000 RU/s) ⓘ

○ Autoscale ● Manual

4000

Estimated cost (USD): **$0.32 hourly / $7.68 daily / $233.60 monthly** (1 region, 4000RU/s, $0.00008/RU)

* Analytical store ⓘ

○ On ● Off

Azure Synapse Link is required for creating an analytical store container. Enable Synapse Link for this Cosmos DB account. Learn more

Unique keys ⓘ

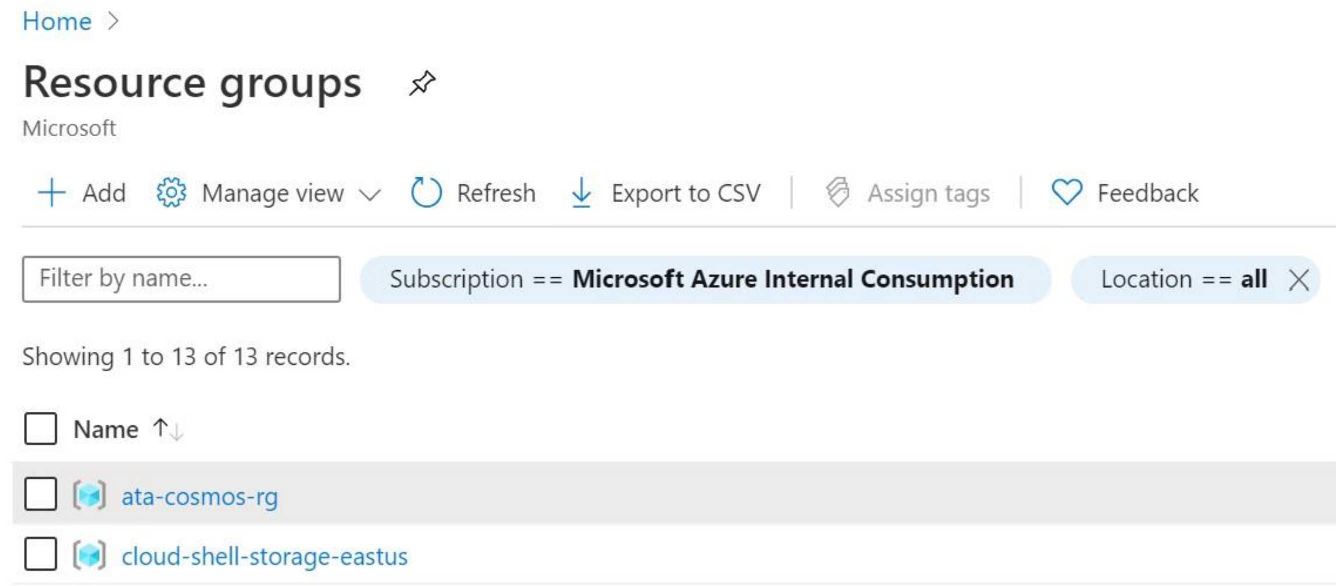
Step 4: Import Lab Data into Container

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

You do not need to do Steps 1-4 in this section and can proceed to Step 5 by opening your Data Factory if you have created a Data Factory resource for the previous lab.

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

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



Subscription == **Microsoft Azure Internal Consumption**

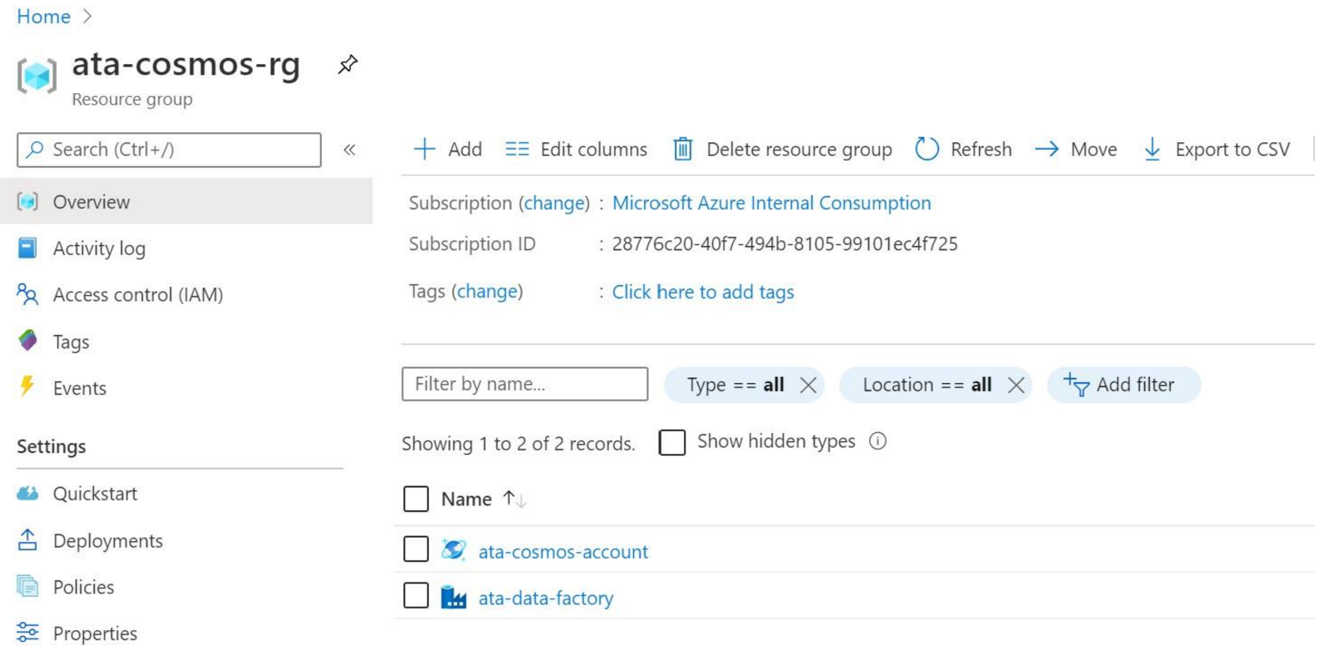
Location == **all**

Showing 1 to 13 of 13 records.

- ata-cosmos-rg
- cloud-shell-storage-eastus

2. In the **Resource groups** blade, locate and select the resource group created for the lab.

3. If you see a Data Factory resource, you can skip to step 5, otherwise select **Add** to add a new resource



Subscription (change) : Microsoft Azure Internal Consumption

Subscription ID : 28776c20-40f7-494b-8105-99101ec4f725

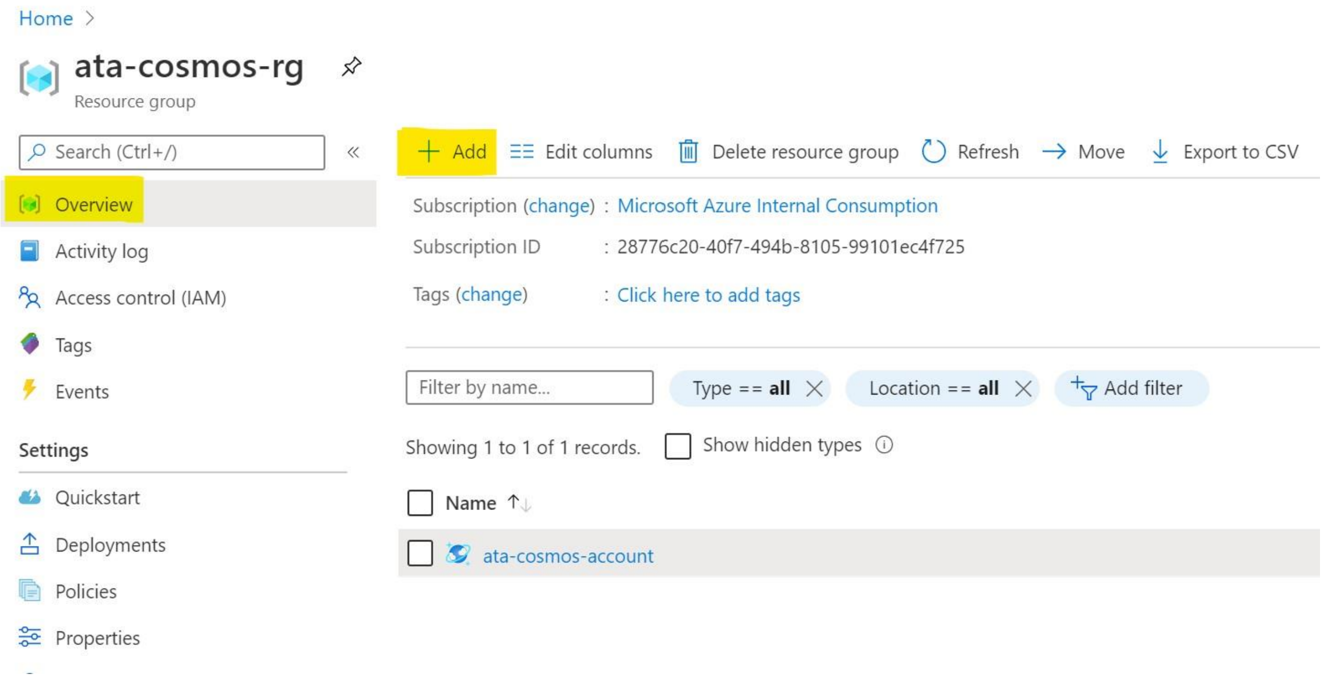
Tags (change) : Click here to add tags

Type == **all**

Location == **all**

Showing 1 to 2 of 2 records.

- ata-cosmos-account
- ata-data-factory



Subscription (change) : Microsoft Azure Internal Consumption

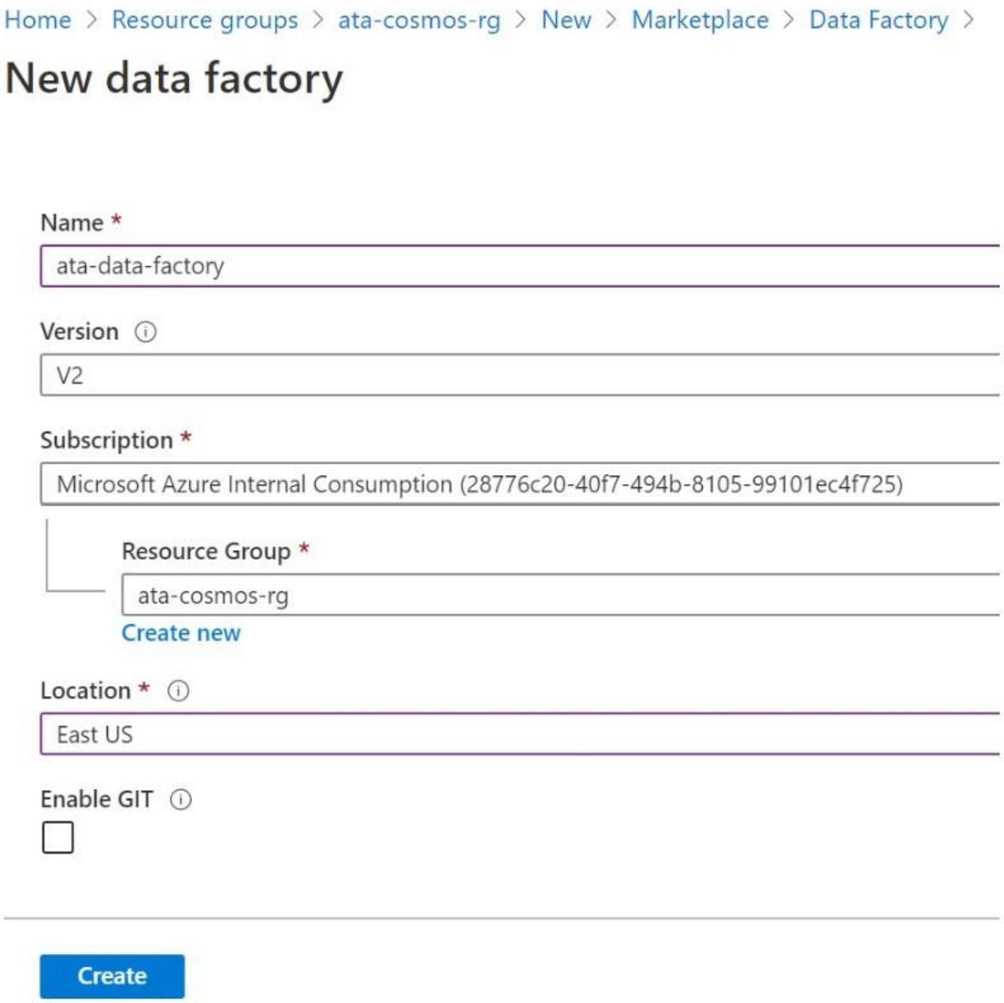
Subscription ID : 28776c20-40f7-494b-8105-99101ec4f725

Tags (change) : Click here to add tags

Showing 1 to 1 of 1 records.

□	Name
□	ata-cosmos-account

- Search for **Data Factory** and select it.
- Create a new **Data Factory**. You should give this data factory a unique name and select the relevant Azure subscription. You should ensure your existing resource group is selected as well as a Version **V2**.
- Select **East US** as the region. Do not select **Enable GIT** (this may be checked by default).
- Select **Create**.



Home > Resource groups > ata-cosmos-rg > New > Marketplace > Data Factory >

New data factory

Name *
ata-data-factory

Version ⓘ
V2

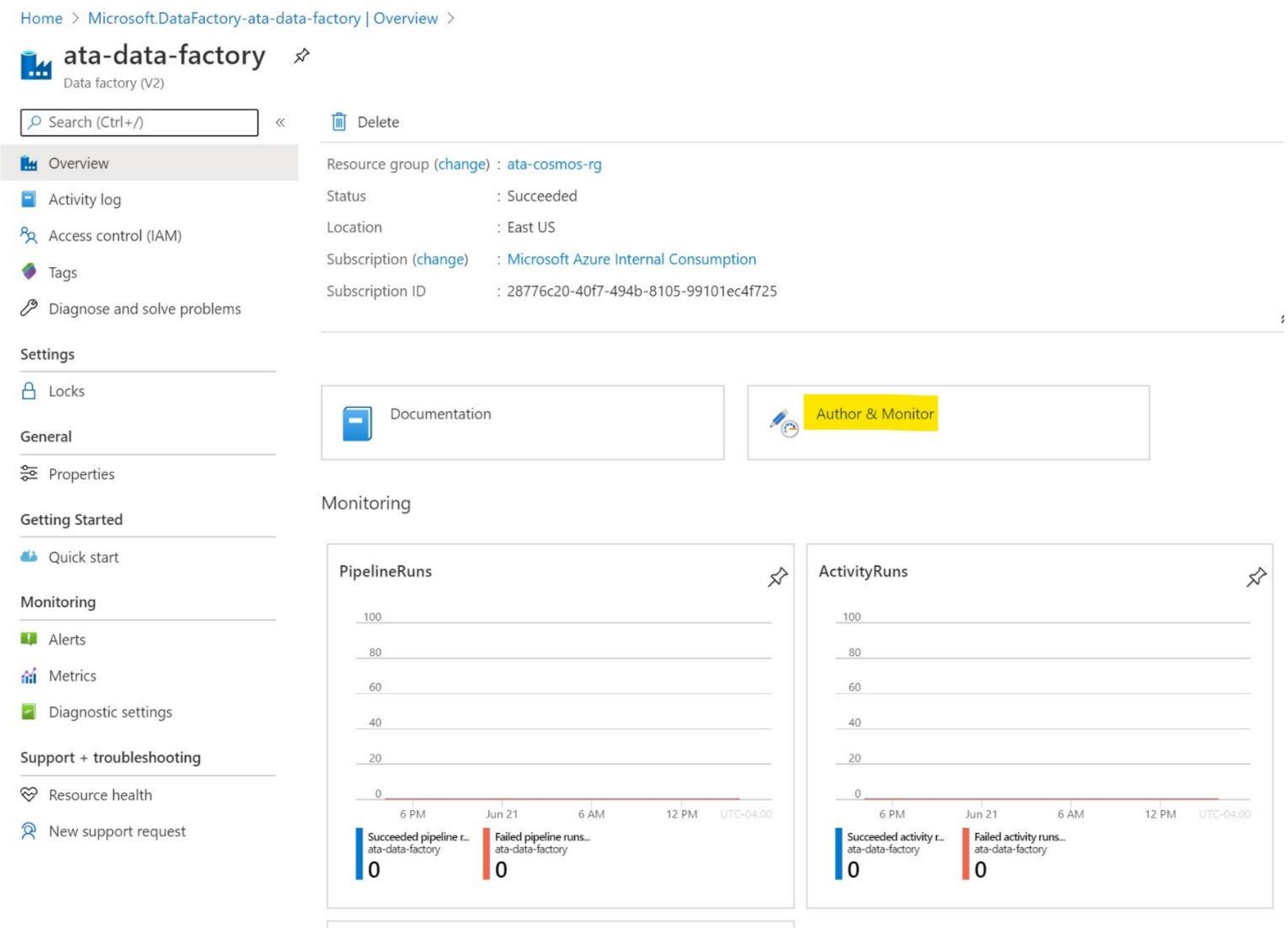
Subscription *
Microsoft Azure Internal Consumption (28776c20-40f7-494b-8105-99101ec4f725)

Resource Group *
ata-cosmos-rg

Location * ⓘ
East US

Enable GIT ⓘ
☐

4. After creation, open your newly created Data Factory. Select **Author & Monitor** and you will launch ADF.


Home > Microsoft.DataFactory-ata-data-factory | Overview >

ata-data-factory

Data factory (V2)

Resource group (change) : ata-cosmos-rg

Status : Succeeded

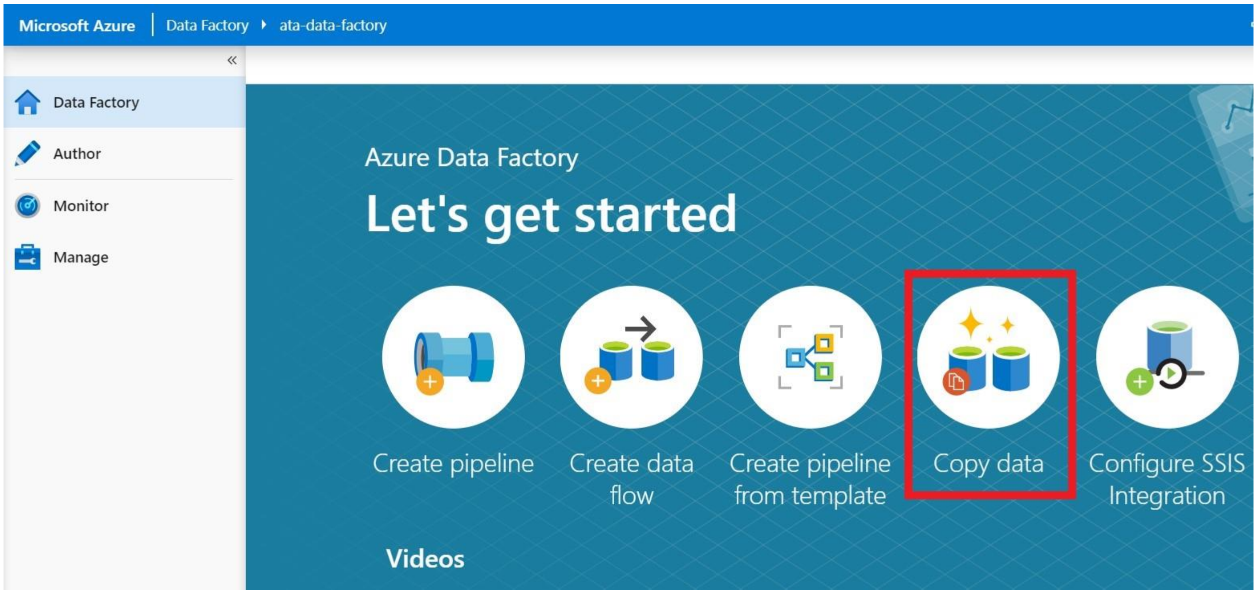
Location : East US

Subscription (change) : Microsoft Azure Internal Consumption

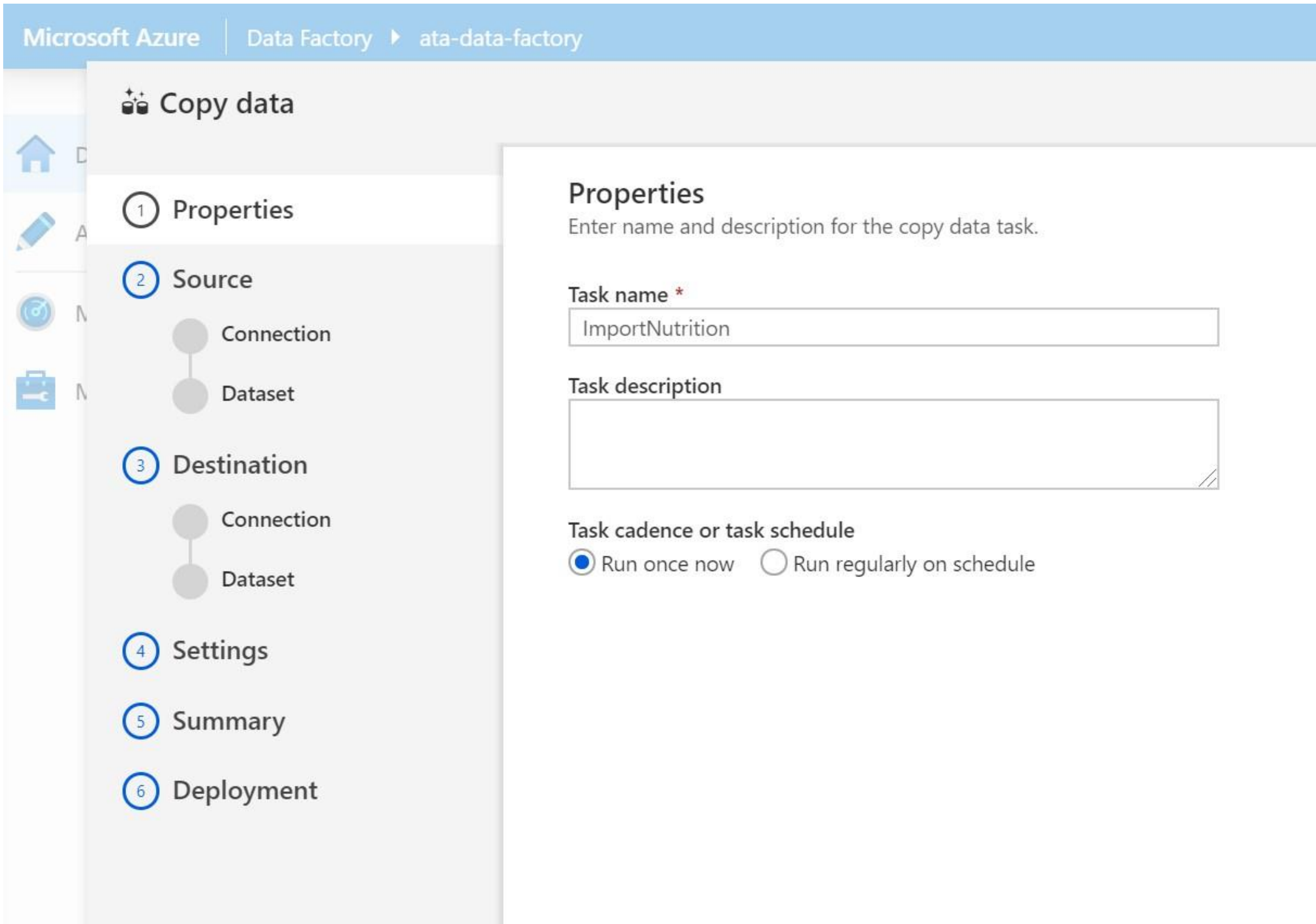
Subscription ID : 28776c20-40f7-494b-8105-99101ec4f725

5. 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.



6. Edit basic properties for this data copy. You should name the task **ImportNutrition** and select to **Run once now**, then select **Next**



Properties

Enter name and description for the copy data task.

Task name *

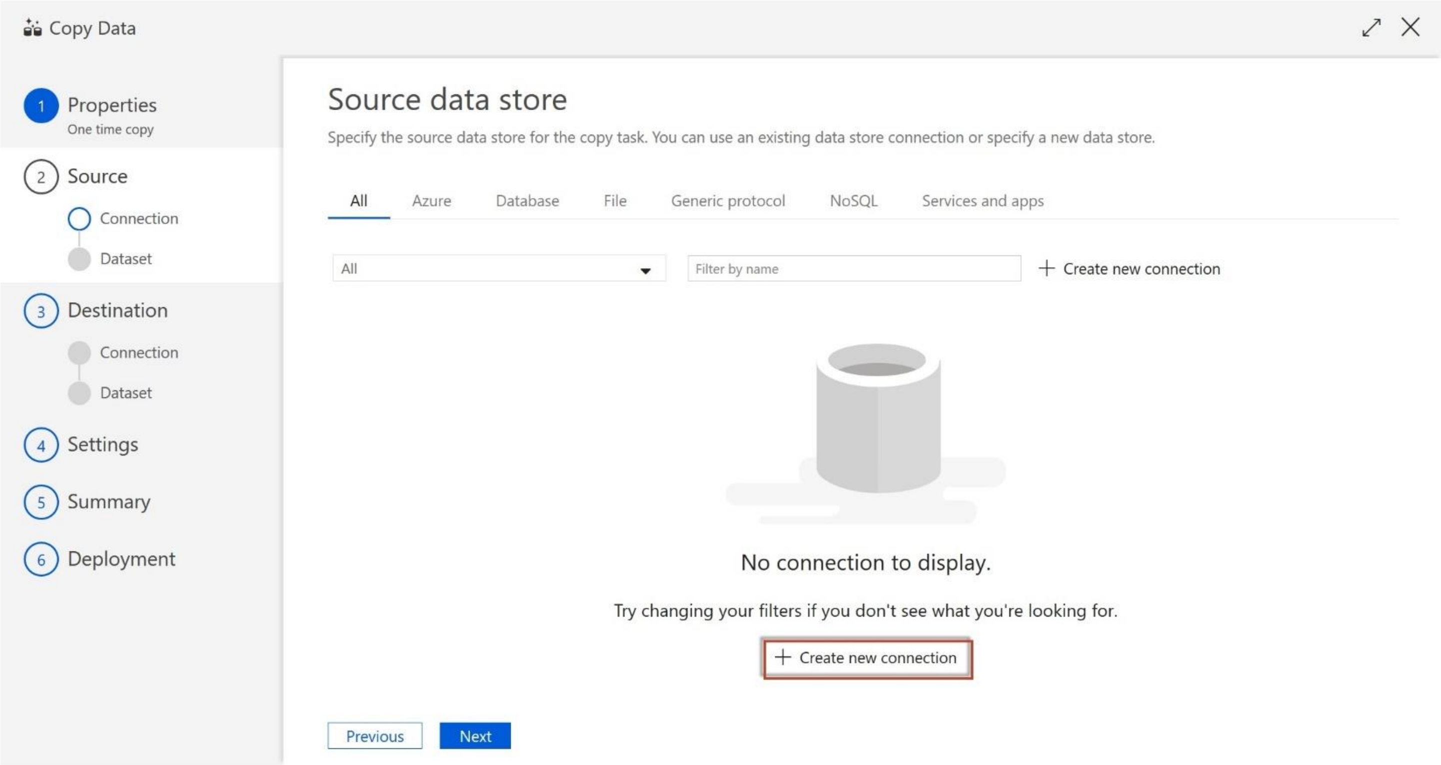
ImportNutrition

Task description

Task cadence or task schedule

● Run once now ○ Run regularly on schedule

7. **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.

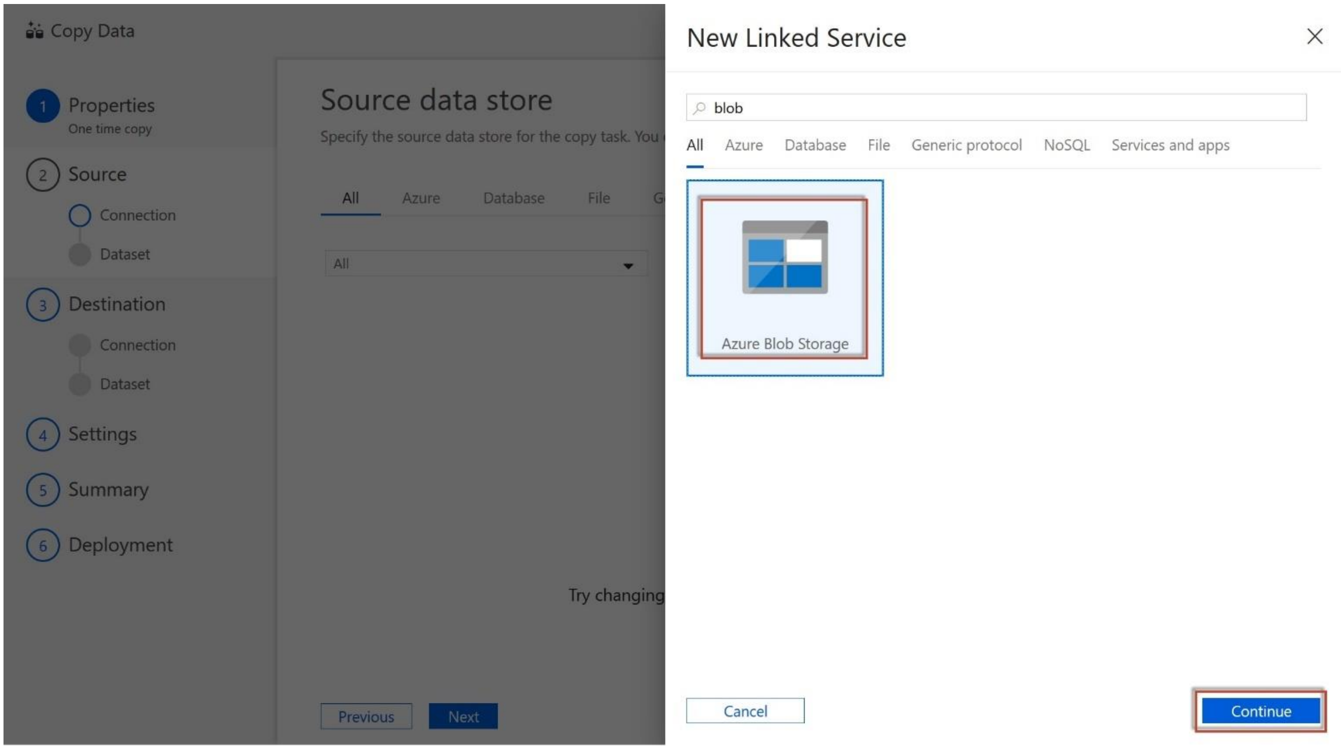


Source data store

Specify the source data store for the copy task. You can use an existing data store connection or specify a new data store.

No connection to display.

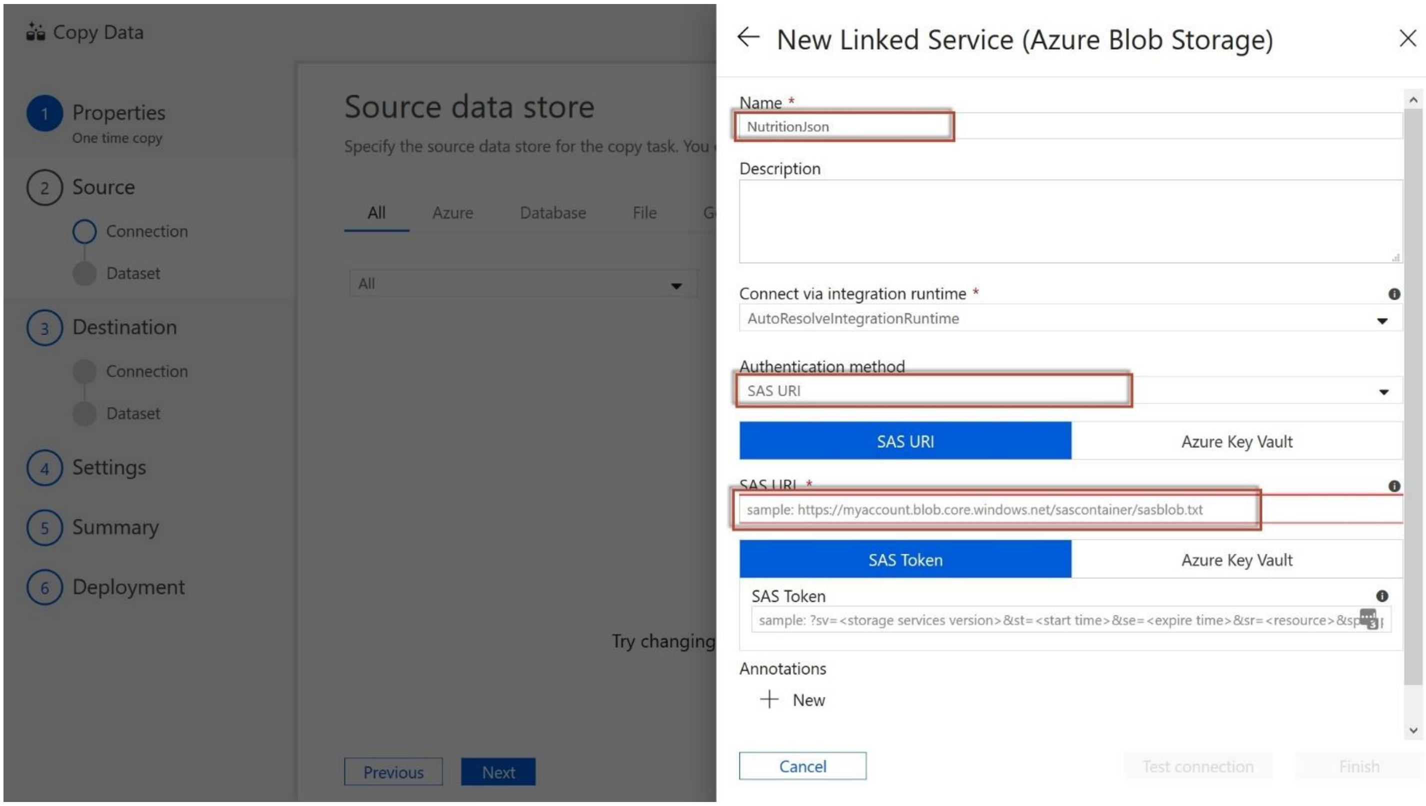
Try changing your filters if you don't see what you're looking for.



blob

8. Name the source **NutritionJson** and select **SAS URI** as the Authentication method. Please use the following SAS URI for read-only access to this Blob Storage container:

   - Enter the following URL into SQL URI:
   https://azuredatadays.blob.core.windows.net/ata?sp=rl&st=2020-10-28T17:25:03Z&se=2021-12-31T17:25:00Z&sv=2019-12-12&sr=c&sig=JBvw%2FqlDzsp5Rk6zSkJrLmNg79QcRoRpzUXl3zNr8os%3D



Name *: NutritionJson

Connect via integration runtime *: AutoResolveIntegrationRuntime

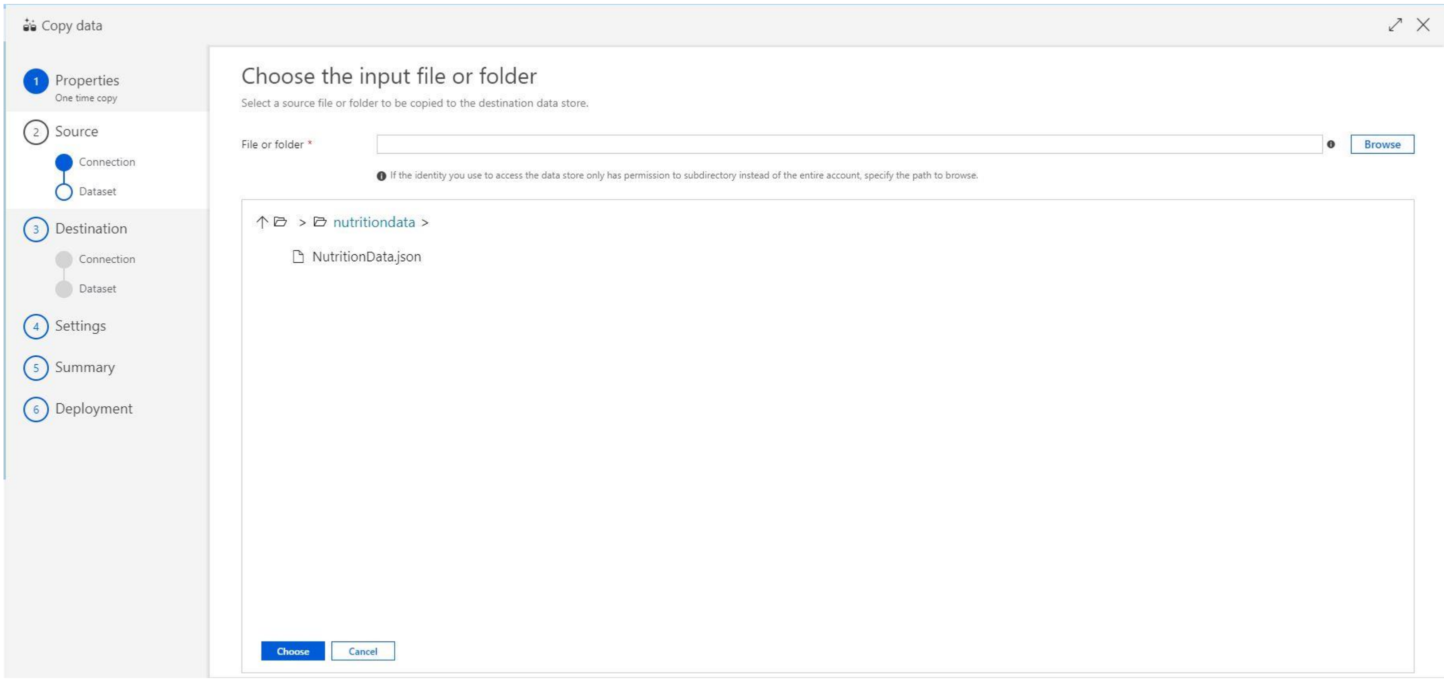
Authentication method: SAS URI

9. Select **Create**

10. Select **Next**

11. Enter ‘ata/NutritionData.json’ into the File and folder* field, click Browse

12. Select **Choose**



Choose the input file or folder

Select a source file or folder to be copied to the destination data store.

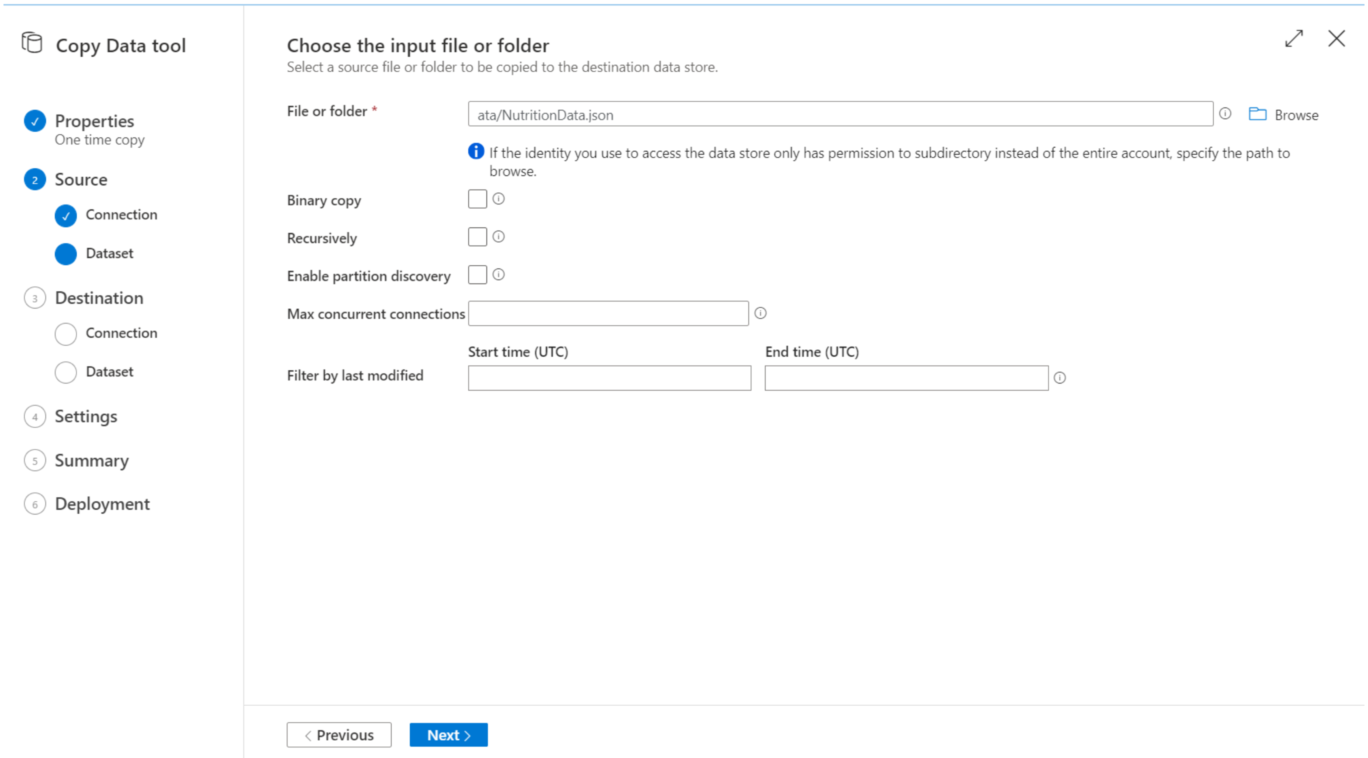
File or folder *

If the identity you use to access the data store only has permission to subdirectory instead of the entire account, specify the path to browse.

> nutritiondata >

NutritionData.json

13. Un-check **Copy file recursively** or **Binary Copy** if they are checked. Also ensure that other fields are empty.



Choose the input file or folder

Select a source file or folder to be copied to the destination data store.

File or folder * ata/NutritionData.json

If the identity you use to access the data store only has permission to subdirectory instead of the entire account, specify the path to browse.

Binary copy □

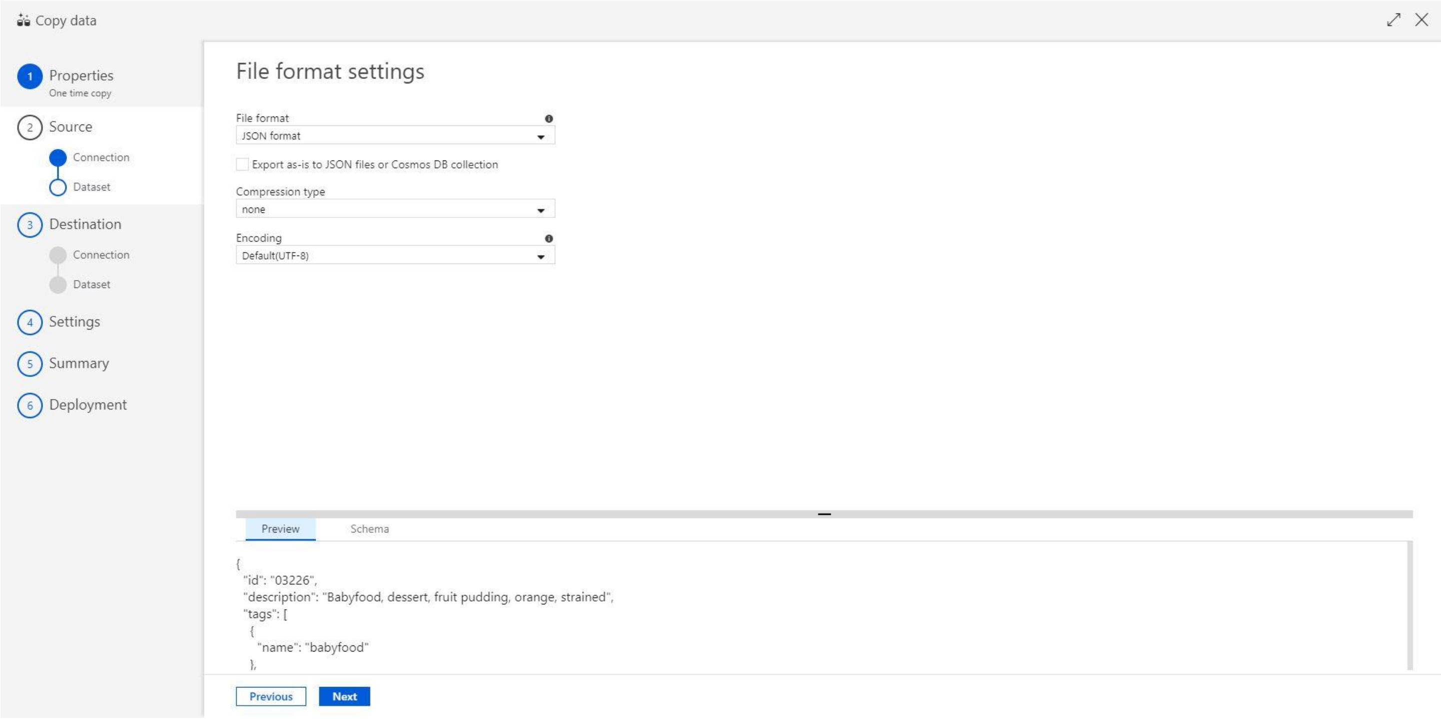
Recursively □

Enable partition discovery □

Max concurrent connections

Filter by last modified Start time (UTC) End time (UTC)

14. Select the file format as **JSON format**. Then select **Next**.



File format settings

File format: JSON format

☐ Export as-is to JSON files or Cosmos DB collection

Compression type: none

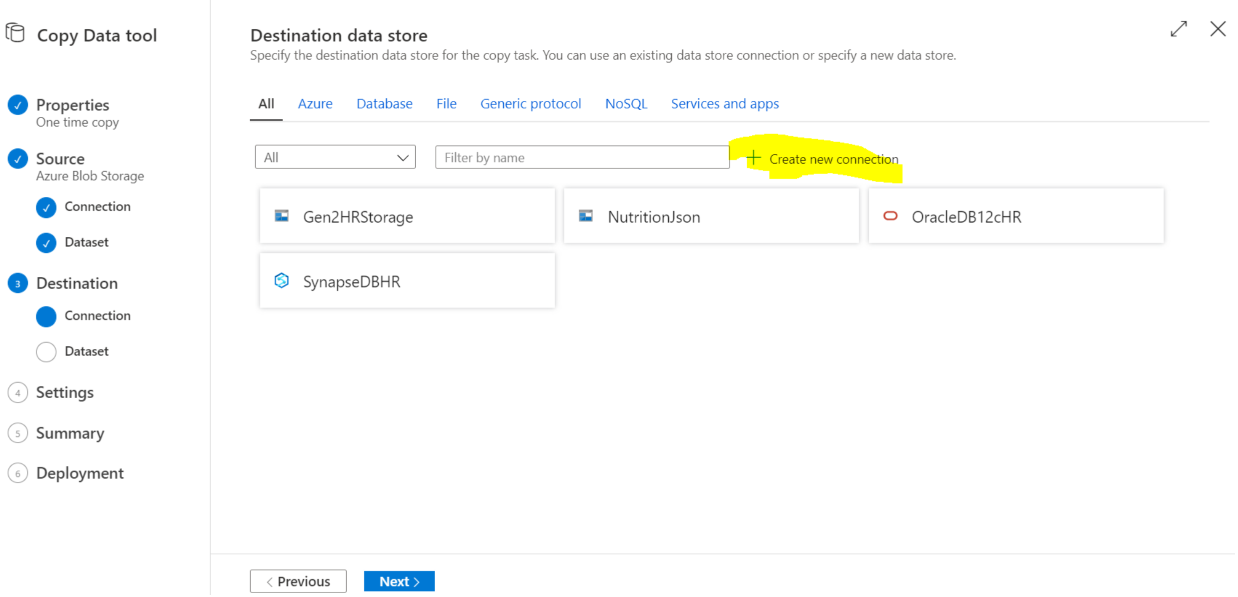
Encoding: Default(UTF-8)

```
{
  "id": "03226",
  "description": "Babyfood, dessert, fruit pudding, orange, strained",
  "tags": [
    {
      "name": "babyfood"
    },
```

15. You have now successfully connected the Blob Storage container with the NutritionData.json file as the source.

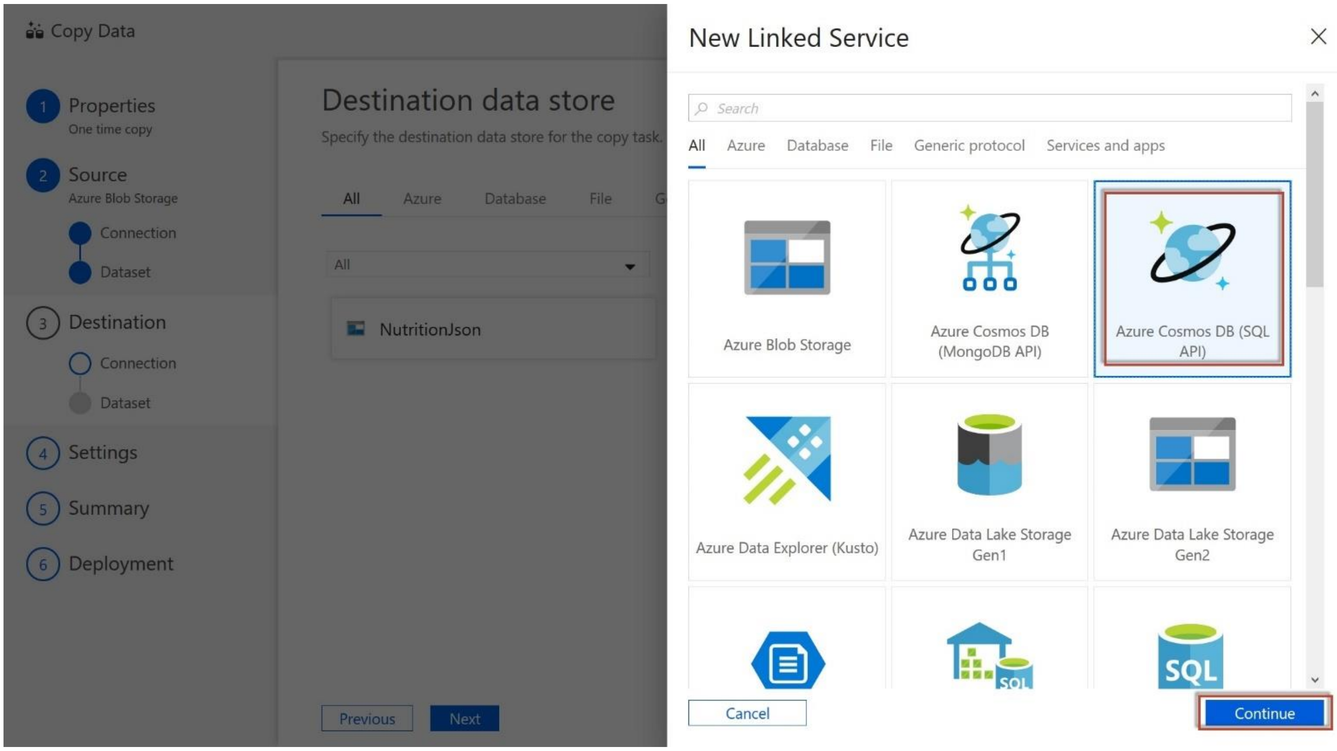
16. For the **Destination data store** add the Cosmos DB target data store by selecting **Create new connection** and selecting **Azure Cosmos DB (SQL API)**.

Click on '+ Create new connection'

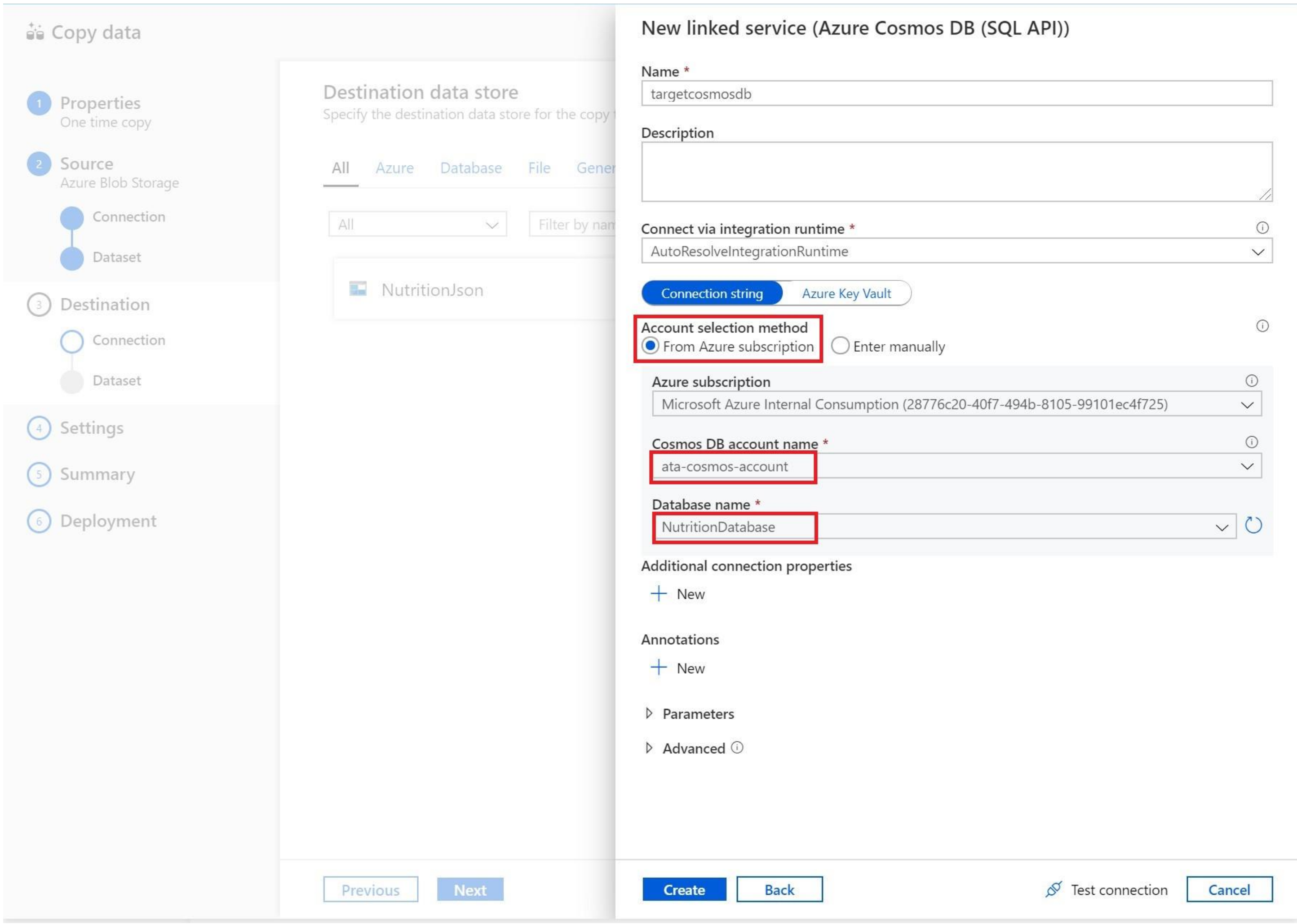


Destination data store

Specify the destination data store for the copy task. You can use an existing data store connection or specify a new data store.



17. Name the linked service **targetcosmosdb** and select your Azure subscription and Cosmos DB account (ata-cosmos-<YourName>-account). You should also select the Cosmos DB **NutritionDatabase** that you created earlier.


New linked service (Azure Cosmos DB (SQL API))

Name * targetcosmosdb

Description

Connect via integration runtime * AutoResolveIntegrationRuntime

Connection string | Azure Key Vault

Account selection method: ● From Azure subscription ○ Enter manually

Azure subscription: Microsoft Azure Internal Consumption (28776c20-40f7-494b-8105-99101ec4f725)

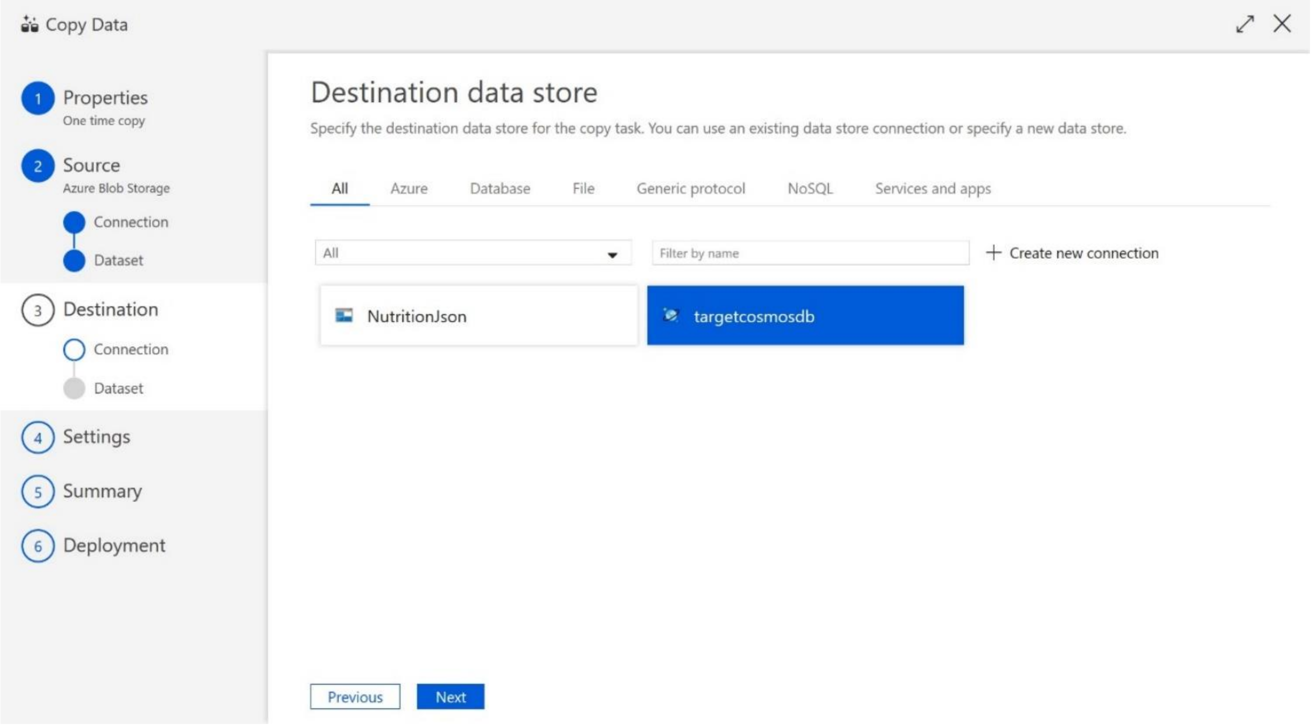
Cosmos DB account name * ata-cosmos-account

Database name * NutritionDatabase

Additional connection properties

Annotations

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


Destination data store

Specify the destination data store for the copy task. You can use an existing data store connection or specify a new data store.

19. Select your **FoodCollection** container from the drop-down menu. You will map your Blob storage file to the correct Cosmos DB container. Select **Next** to continue.

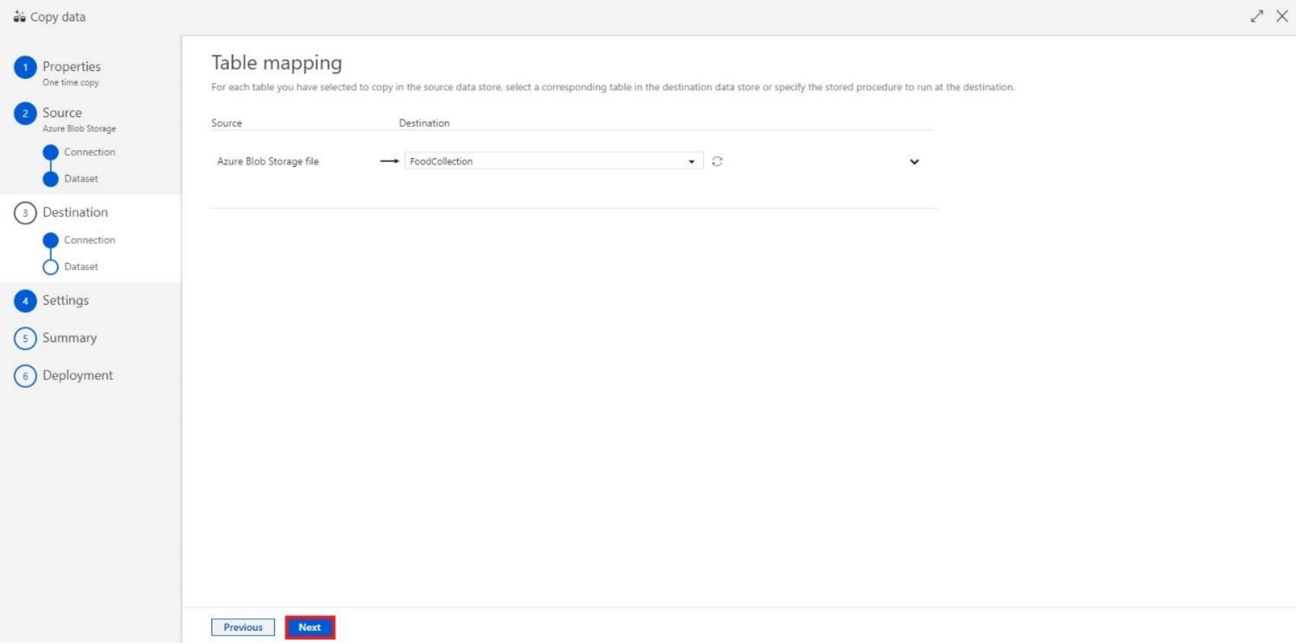
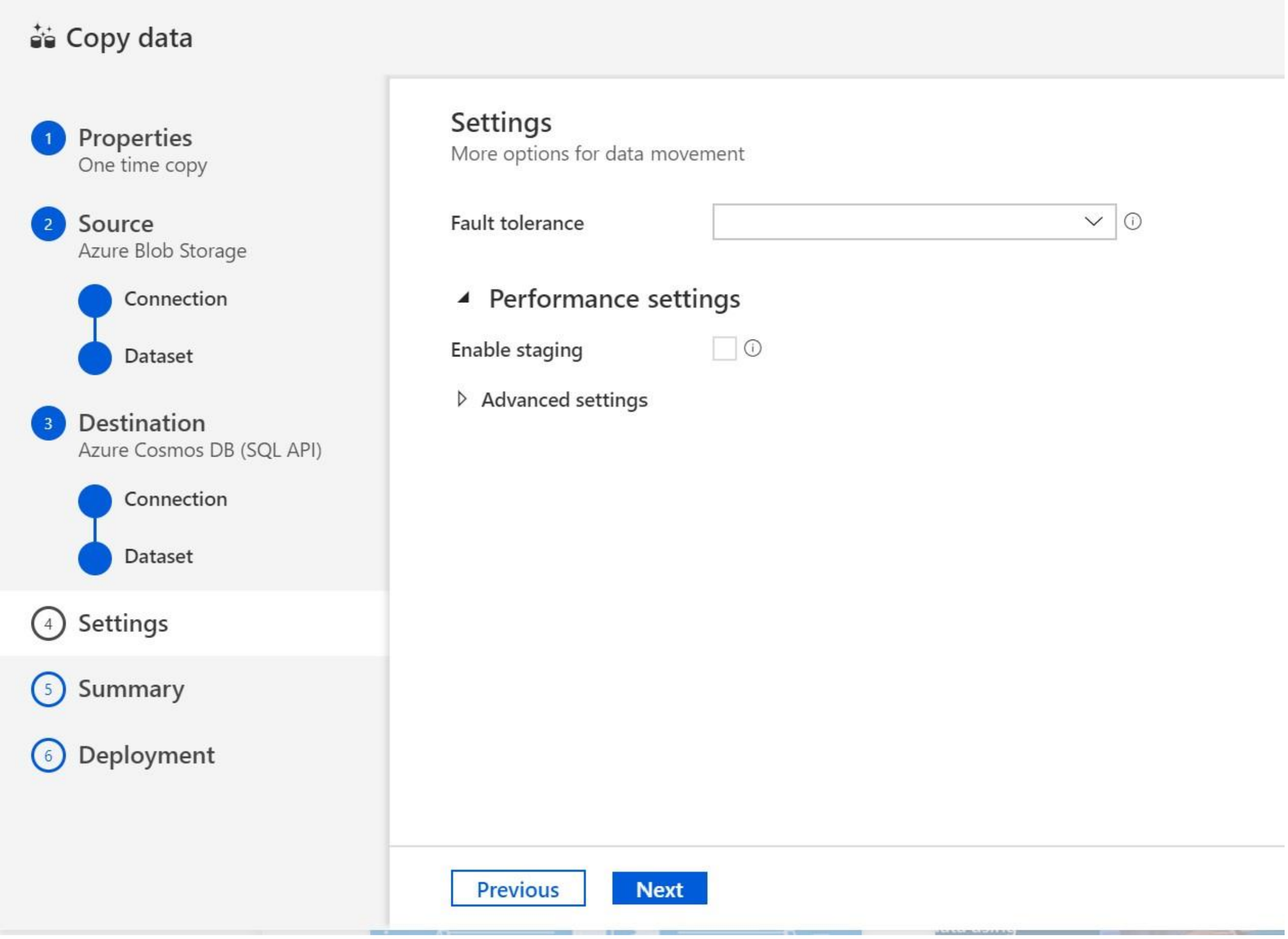


Table mapping

For each table you have selected to copy in the source data store, select a corresponding table in the destination data store or specify the stored procedure to run at the destination.

Source		Destination
Azure Blob Storage file	→	FoodCollection

20. There is no need to change any Settings. Select **next**.



Settings

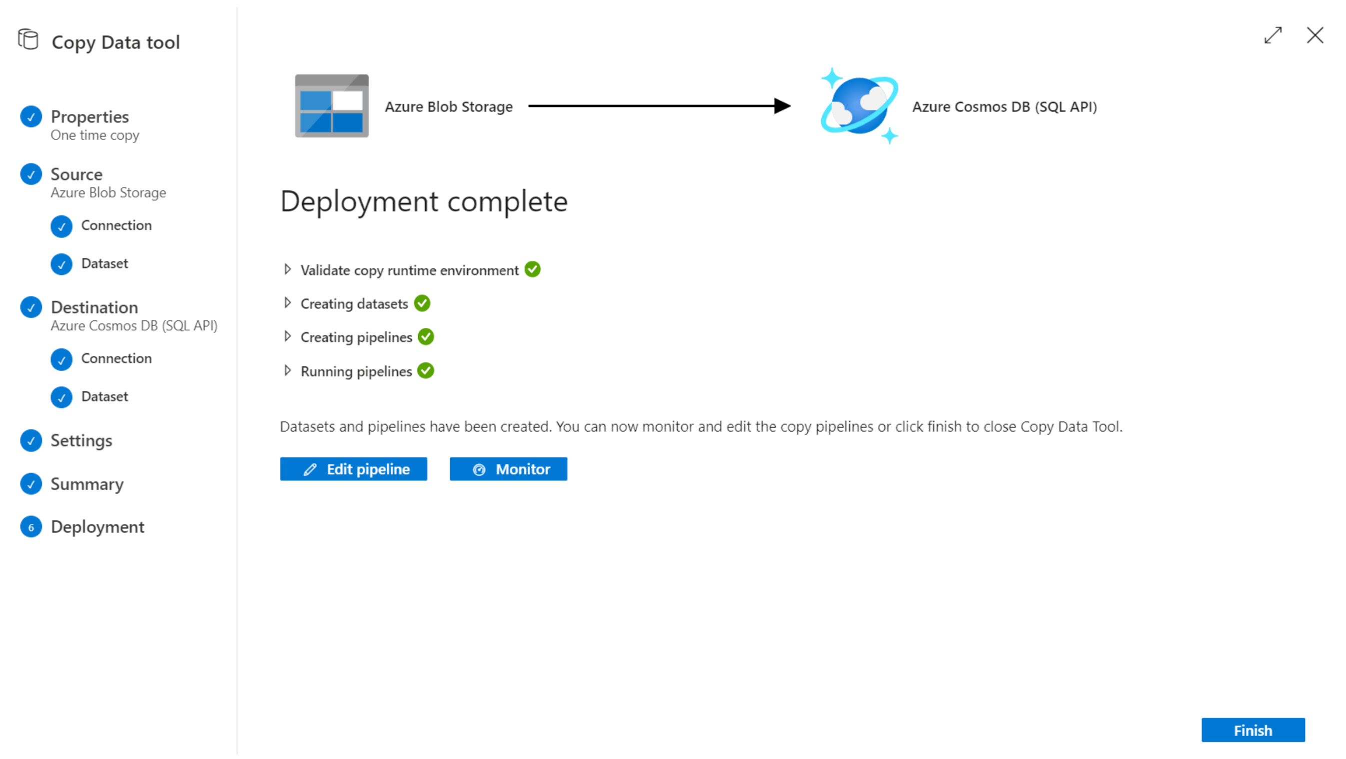
More options for data movement

Fault tolerance

Performance settings

Enable staging □

Advanced settings

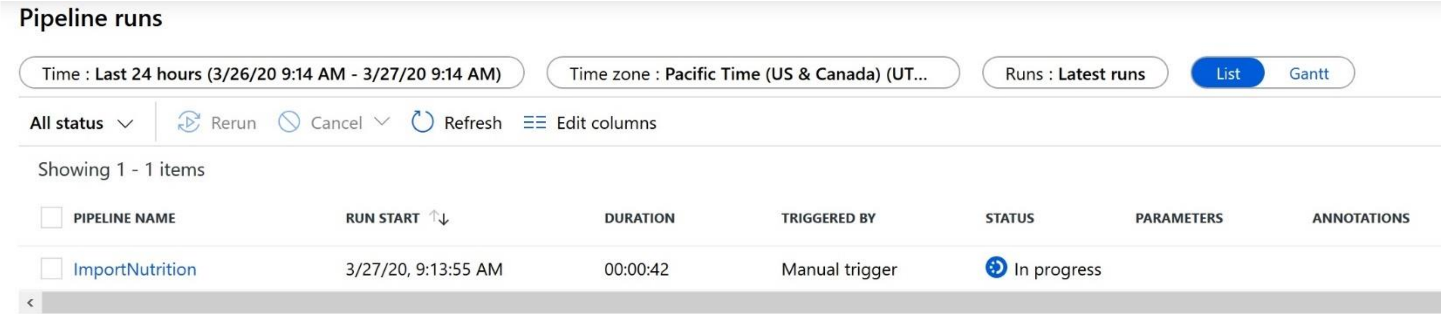


Deployment complete

- Validate copy runtime environment ✓
- Creating datasets ✓
- Creating pipelines ✓
- Running pipelines ✓

Datasets and pipelines have been created. You can now monitor and edit the copy pipelines or click finish to close Copy Data Tool.

21. Select **Next** to begin deployment After deployment is complete, select **Monitor**.



Time : Last 24 hours (3/26/20 9:14 AM - 3/27/20 9:14 AM)

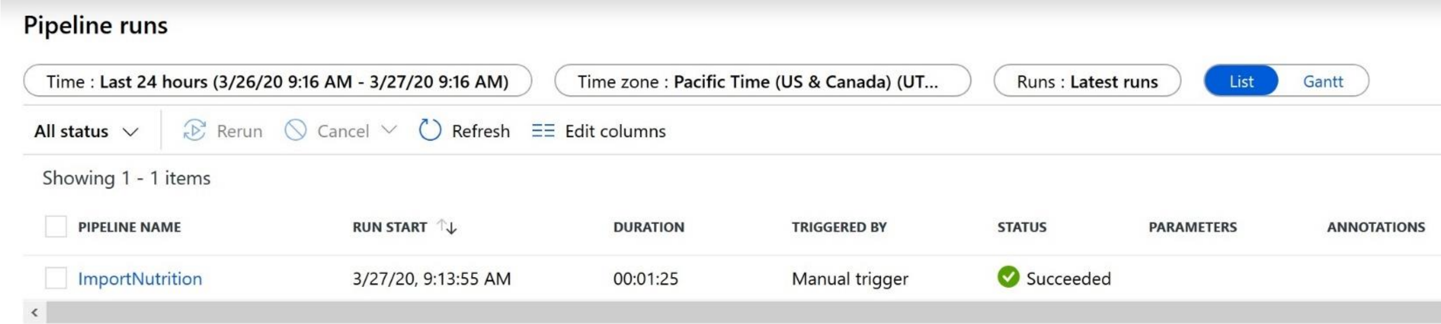
Time zone : Pacific Time (US & Canada) (UT...

Runs : Latest runs

Showing 1 - 1 items

PIPELINE NAME	RUN START	DURATION	TRIGGERED BY	STATUS	PARAMETERS	ANNOTATIONS
ImportNutrition	3/27/20, 9:13:55 AM	00:00:42	Manual trigger	In progress		

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



Time : Last 24 hours (3/26/20 9:16 AM - 3/27/20 9:16 AM)

Time zone : Pacific Time (US & Canada) (UT...

Runs : Latest runs

Showing 1 - 1 items

PIPELINE NAME	RUN START	DURATION	TRIGGERED BY	STATUS	PARAMETERS	ANNOTATIONS
ImportNutrition	3/27/20, 9:13:55 AM	00:01:25	Manual trigger	✓ Succeeded		

23. Once the import process has completed, close the ADF. You will now proceed to validate your imported data.

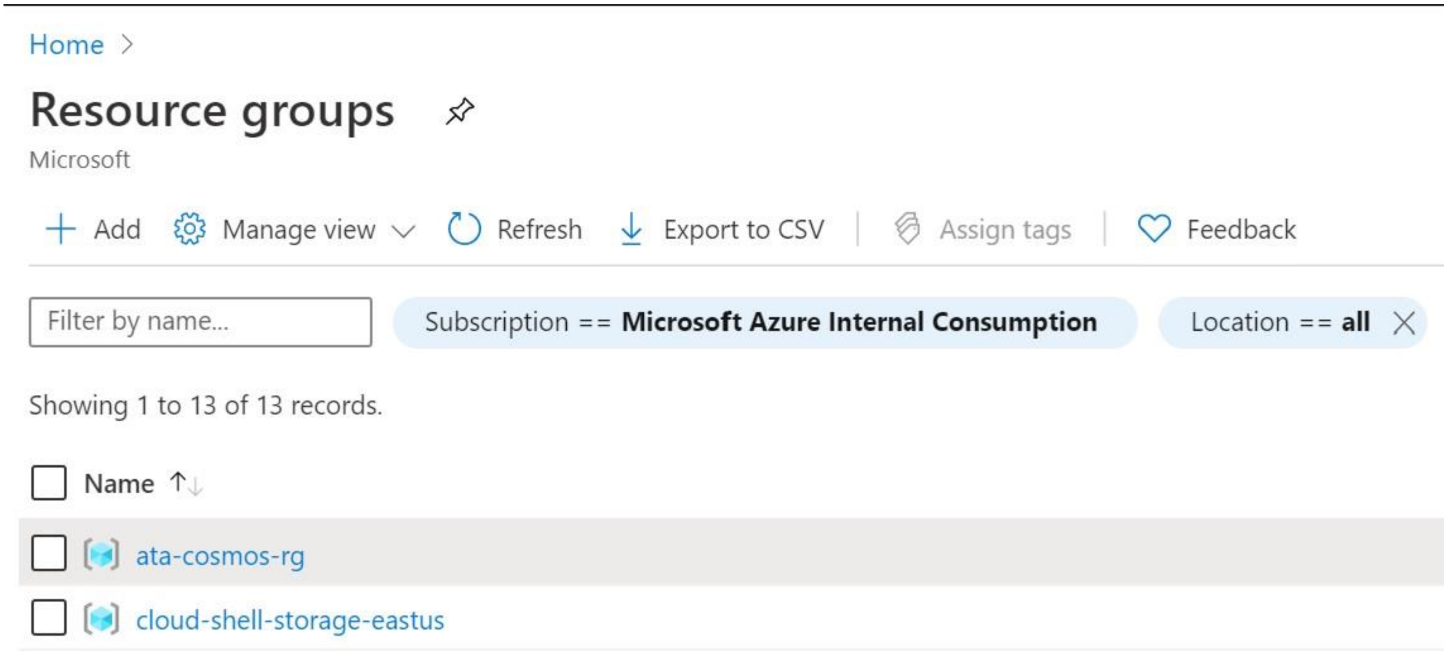
Validate Imported Data

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 view the data stored in our container.

You will validate that the data was successfully imported into your container using the **Items** view in the **Data Explorer**.

1. Return to the **Azure Portal** (http://portal.azure.com).

2. Select the **Resource groups** link, locate and select the resource group created for the lab.



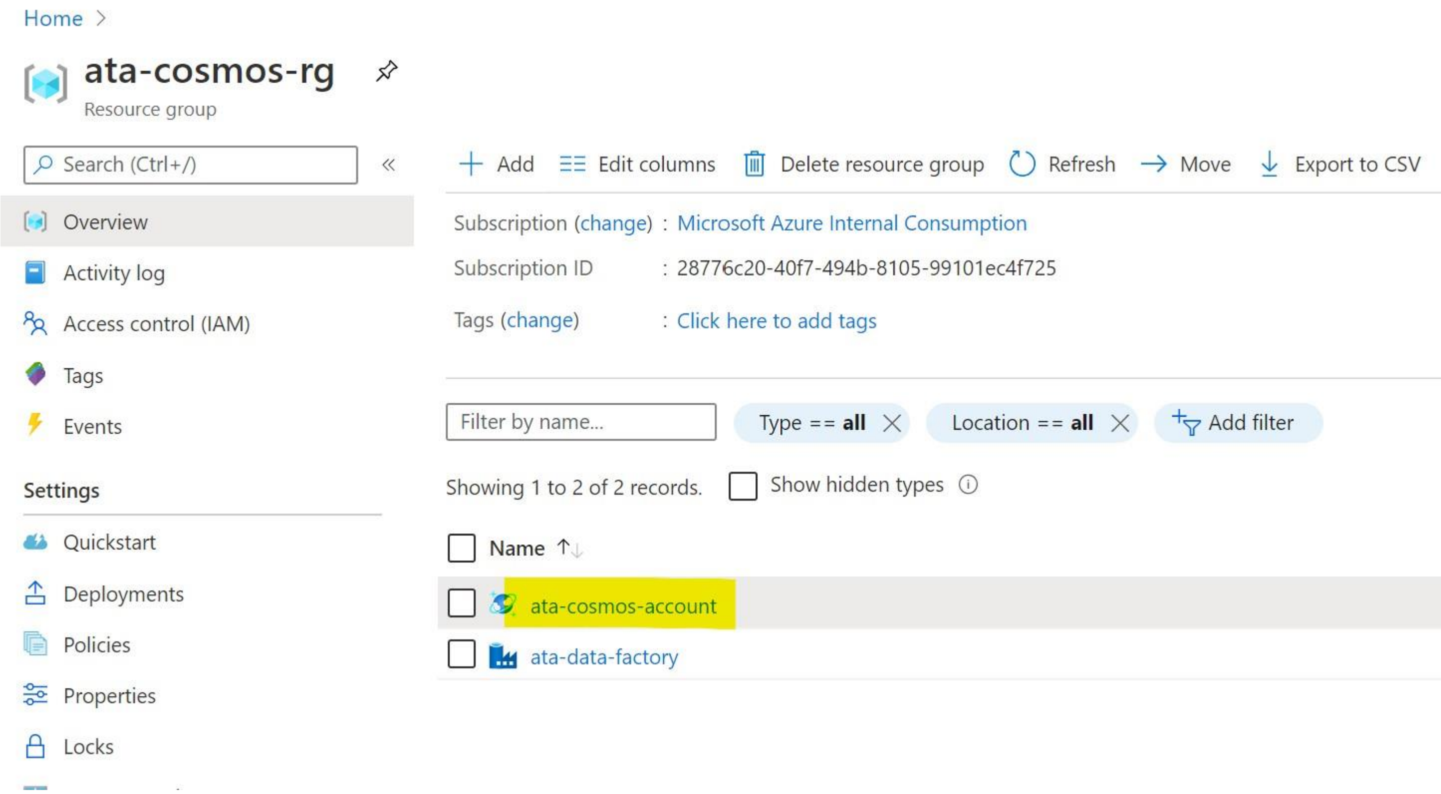
Subscription == **Microsoft Azure Internal Consumption**

Location == **all**

Showing 1 to 13 of 13 records.

□	Name ↑↓
□	ata-cosmos-rg
□	cloud-shell-storage-eastus

3. Inside the resource group, select the **Azure Cosmos DB** account you created for the lab.

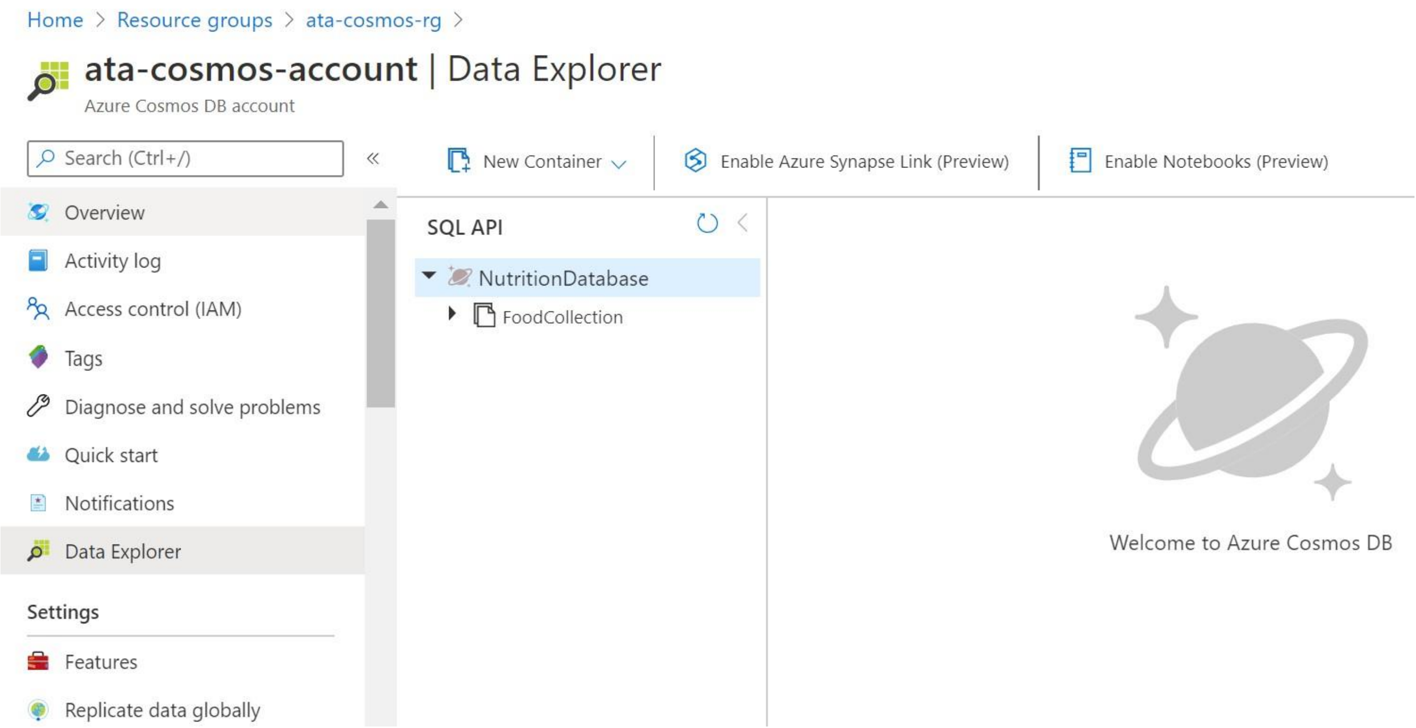


Subscription (change) : Microsoft Azure Internal Consumption

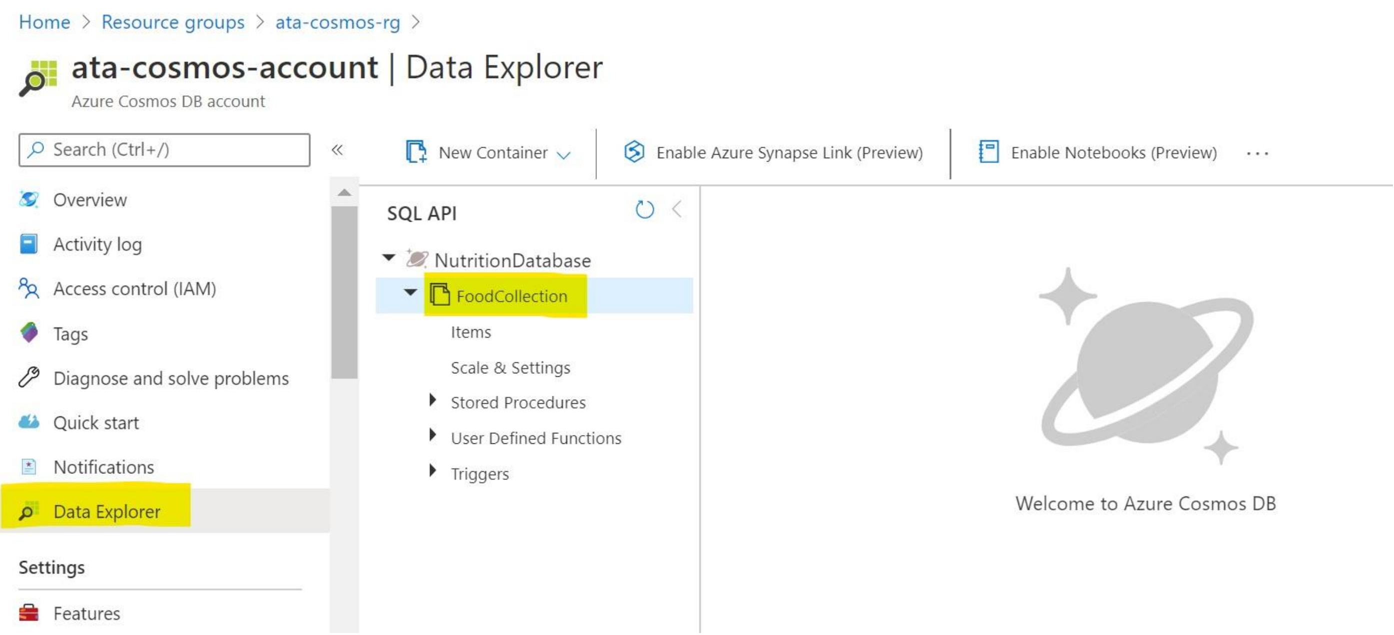
Subscription ID : 28776c20-40f7-494b-8105-99101ec4f725

Tags (change) : Click here to add tags

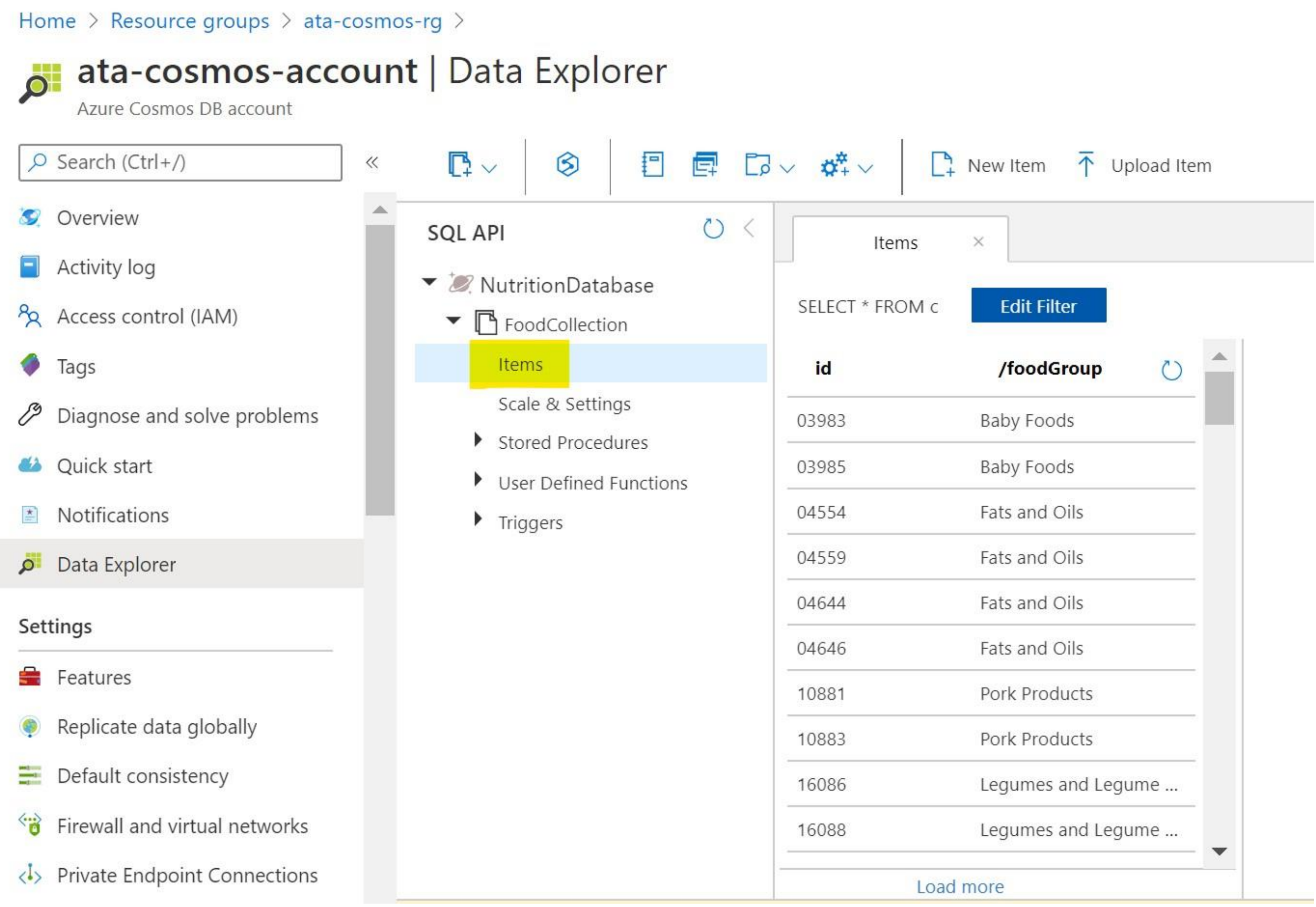
4. In the **Azure Cosmos DB** blade, locate and select the **Data Explorer** link on the left side of the blade.



5. In the **Data Explorer** section, expand the **NutritionDatabase** database node and then expand the **FoodCollection** container node.

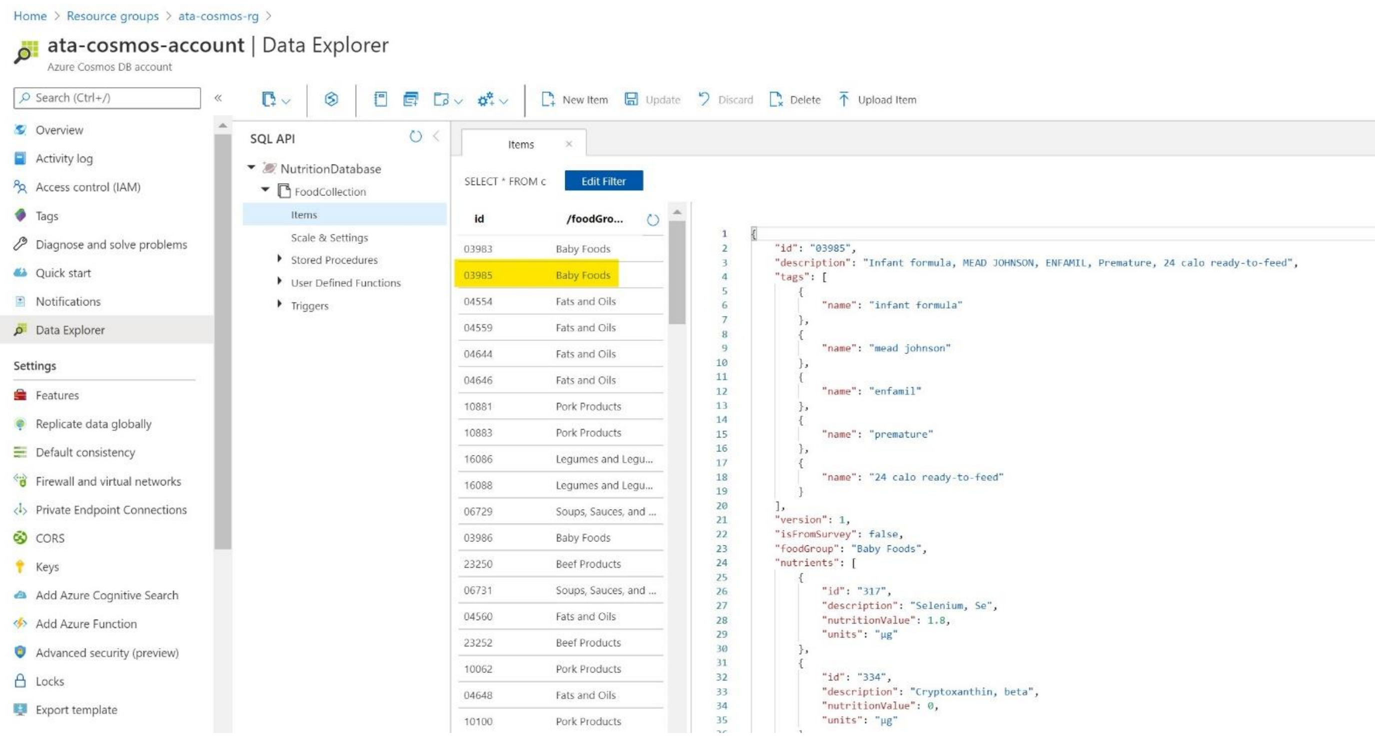


6. Within the **FoodCollection** node, select the **Items** link to view a subset of the various documents in the container. Select a few of the documents and observe the properties and structure of the documents.



SELECT * FROM c

id	/foodGroup
03983	Baby Foods
03985	Baby Foods
04554	Fats and Oils
04559	Fats and Oils
04644	Fats and Oils
04646	Fats and Oils
10881	Pork Products
10883	Pork Products
16086	Legumes and Legume ...
16088	Legumes and Legume ...



```
SELECT * FROM c
```

id	/foodGro...
03983	Baby Foods
03985	Baby Foods
04554	Fats and Oils
04559	Fats and Oils
04644	Fats and Oils
04646	Fats and Oils
10881	Pork Products
10883	Pork Products
16086	Legumes and Legu...
16088	Legumes and Legu...
06729	Soups, Sauces, and ...
03986	Baby Foods
23250	Beef Products
06731	Soups, Sauces, and ...
04560	Fats and Oils
23252	Beef Products
10062	Pork Products
04648	Fats and Oils
10100	Pork Products

```
{
    "id": "03985",
    "description": "Infant formula, MEAD JOHNSON, ENFAMIL, Premature, 24 calo ready-to-feed",
    "tags": [
        {
            "name": "infant formula"
        },
        {
            "name": "mead johnson"
        },
        {
            "name": "enfamil"
        },
        {
            "name": "premature"
        },
        {
            "name": "24 calo ready-to-feed"
        }
    ],
    "version": 1,
    "isFromSurvey": false,
    "foodGroup": "Baby Foods",
    "nutrients": [
        {
            "id": "317",
            "description": "Selenium, Se",
            "nutritionValue": 1.8,
            "units": "µg"
        },
        {
            "id": "334",
            "description": "Cryptoxanthin, beta",
            "nutritionValue": 0,
            "units": "µg"
```

Lab 2: Querying in Azure Cosmos DB

Azure Cosmos DB SQL API accounts provide support for querying items using the Structured Query Language (SQL), one of the most familiar and popular query languages, as a JSON query language. In this lab, you will explore how to use these rich query capabilities directly through the Azure Portal. No separate tools or client side code are required.

Query Overview

Querying JSON with SQL allows Azure Cosmos DB to combine the advantages of a legacy relational databases with a NoSQL database. You can use many rich query capabilities such as sub-queries or aggregation functions but still retain the many advantages of modeling data in a NoSQL database.

Azure Cosmos DB supports strict JSON items only. The type system and expressions are restricted to deal only with JSON types. For more information, see the JSON specification.

Running your first query

In this lab section, you will query your **FoodCollection**. If you prefer, you can also complete all lab steps using the Azure Cosmos DB Query Playground.

You will begin by running basic queries with `SELECT`, `WHERE`, and `FROM` clauses.

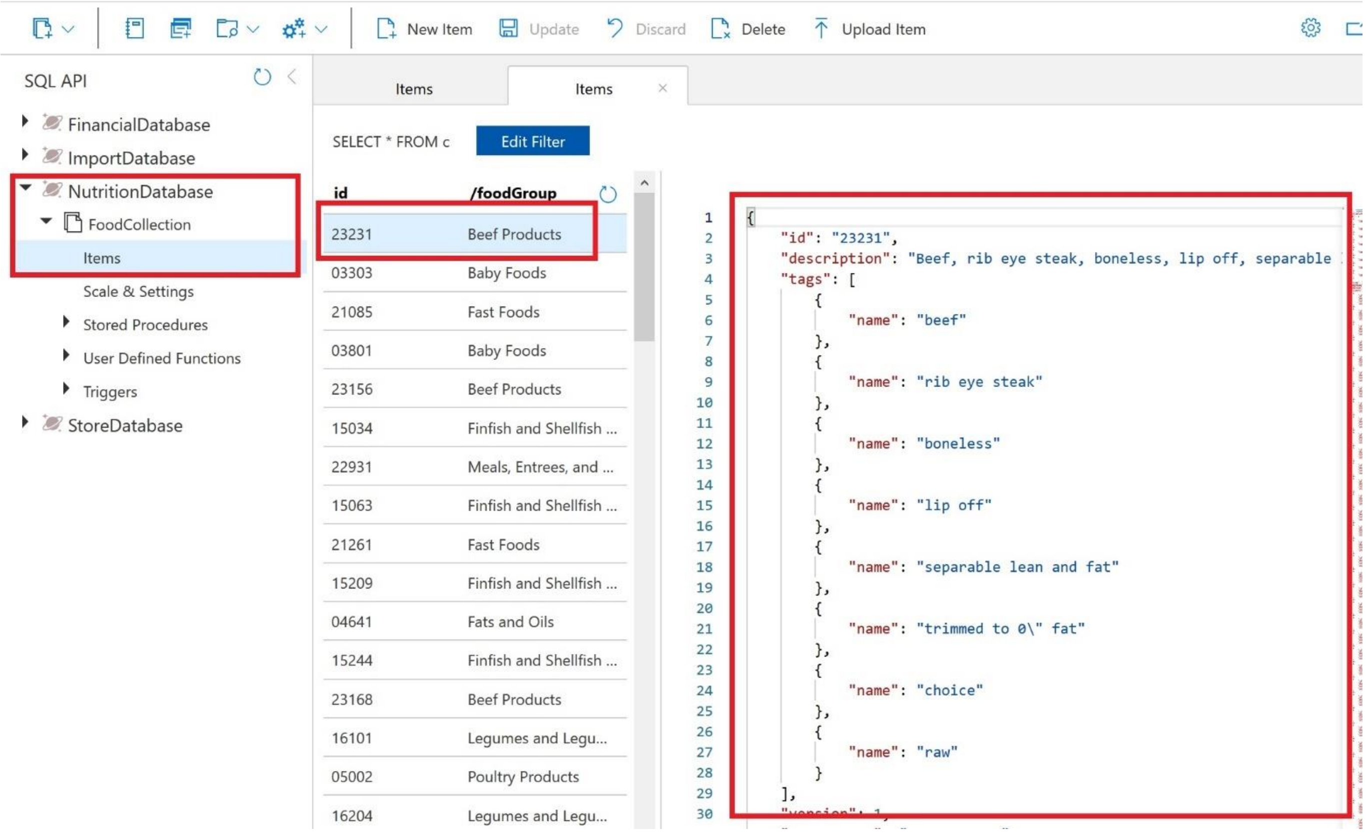
Open Data Explorer

1. In the **Azure Cosmos DB** blade, locate and select the **Data Explorer** link on the left side of the blade.

2. In the **Data Explorer** section, expand the **NutritionDatabase** database node and then expand the **FoodCollection** container node.

3. Within the **FoodCollection** node, select the **Items** link.

4. View the items within the container. Observe how these documents have many properties, including arrays.

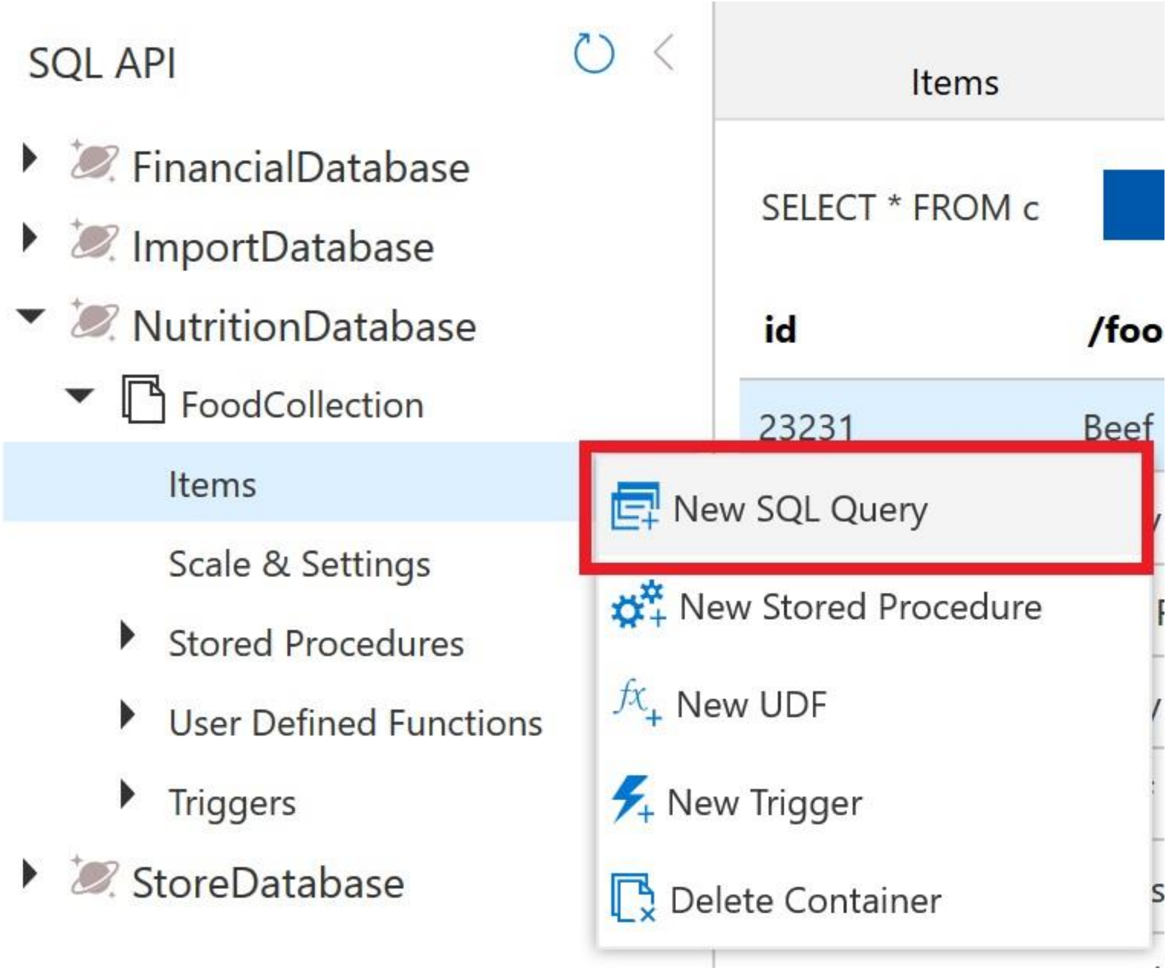


SELECT * FROM c

id	/foodGroup
23231	Beef Products
03303	Baby Foods
21085	Fast Foods
03801	Baby Foods
23156	Beef Products
15034	Finfish and Shellfish ...
22931	Meals, Entrees, and ...
15063	Finfish and Shellfish ...
21261	Fast Foods
15209	Finfish and Shellfish ...
04641	Fats and Oils
15244	Finfish and Shellfish ...
23168	Beef Products
16101	Legumes and Legu...
05002	Poultry Products
16204	Legumes and Legu...

```
{
    "id": "23231",
    "description": "Beef, rib eye steak, boneless, lip off, separable
    "tags": [
        {
            "name": "beef"
        },
        {
            "name": "rib eye steak"
        },
        {
            "name": "boneless"
        },
        {
            "name": "lip off"
        },
        {
            "name": "separable lean and fat"
        },
        {
            "name": "trimmed to 0\" fat"
        },
        {
            "name": "choice"
        },
        {
            "name": "raw"
        }
    ],
    "version": 1,
```

5. Select **New SQL Query**.

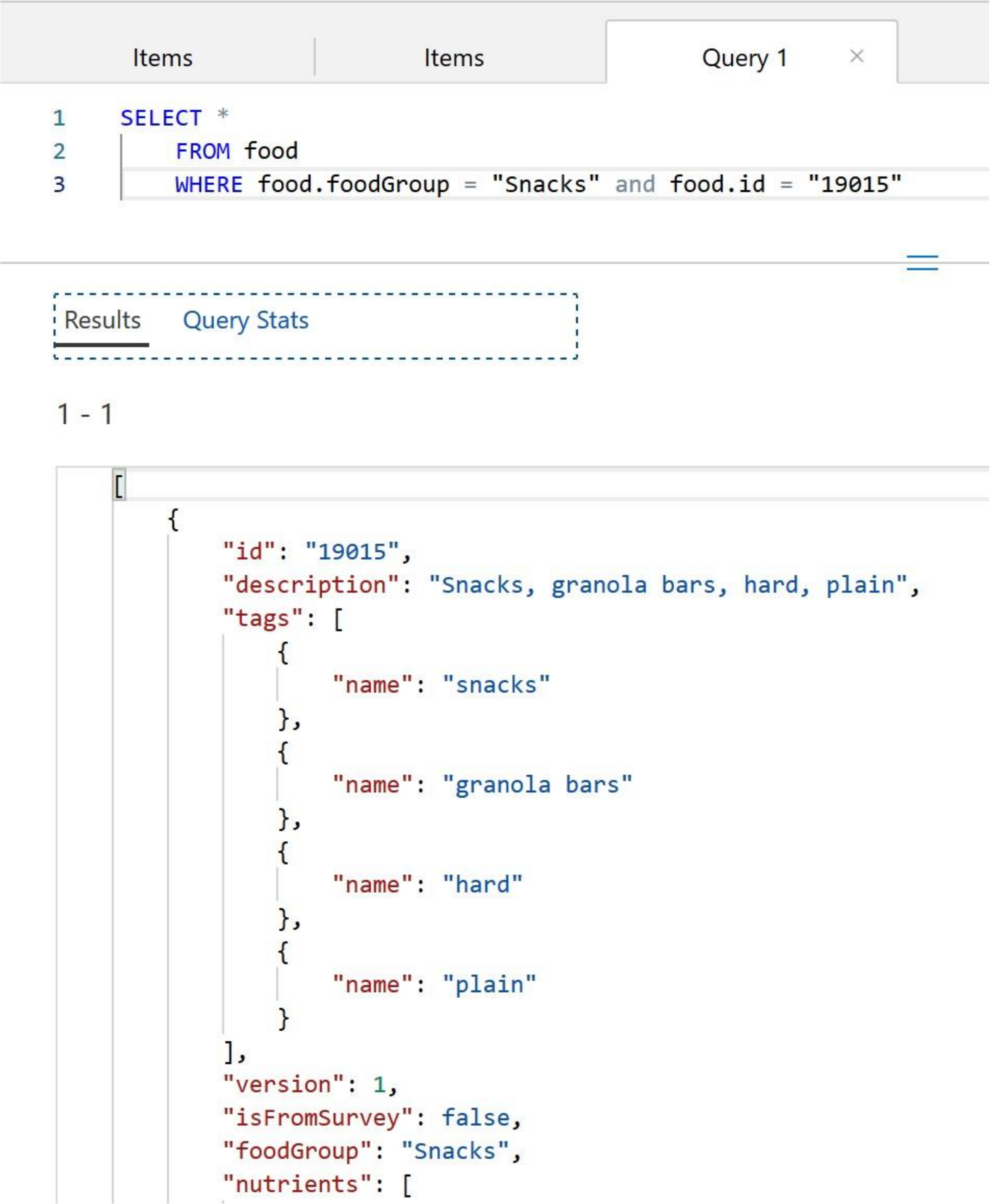


```
SELECT * FROM c
```

6. Paste the following SQL query and select **Execute Query**.

```
SELECT * FROM food
WHERE food.foodGroup = "Snacks" and food.id = "19015"
```

7. You will see that the query returned the single document where id is "19015" and the foodGroup is "Snacks". Explore the structure of this item as it is representative of the items within the **FoodCollection** container that we will be working with for the remainder of this section.



```
SELECT *
    FROM food
    WHERE food.foodGroup = "Snacks" and food.id = "19015"
```

1 - 1

```
[
    {
        "id": "19015",
        "description": "Snacks, granola bars, hard, plain",
        "tags": [
            {
                "name": "snacks"
            },
            {
                "name": "granola bars"
            },
            {
                "name": "hard"
            },
            {
                "name": "plain"
            }
        ],
        "version": 1,
        "isFromSurvey": false,
        "foodGroup": "Snacks",
        "nutrients": [
```

Dot and quoted property projection accessors

You can choose which properties of the document to project into the result using the dot notation. If you wanted to return only the item's id you could run the query below.

Select **New SQL Query**. Paste the following SQL query and selecting **Execute Query**.

```
SELECT food.id
FROM food
WHERE food.foodGroup = "Snacks" and food.id = "19015"
```

Though less common, you can also access properties using the quoted property operator [""]. For example, SELECT food.id and SELECT food["id"] are equivalent. This syntax is useful to escape a property that contains spaces, special characters, or has the same name as a SQL keyword or reserved word.

```sql
SELECT food["id"]
FROM food
WHERE food["foodGroup"] = "Snacks" and food["id"] = "19015"
```

WHERE clauses

Let's explore WHERE clauses. You can add complex scalar expressions including arithmetic, comparison and logical operators in the WHERE clause.

Run the below query by selecting the **New SQL Query**.

Paste the following SQL query and then select **Execute Query**.

```sql
SELECT food.id,
food.description,
food.tags,
food.foodGroup,
food.manufacturerName,
food.version
FROM food
WHERE (food.manufacturerName = "The Coca-Cola Company" AND food.version > 0)
```

This query will return the id, description, servings, tags, foodGroup, manufacturerName and version for items with "The Coca-Cola Company" for manufacturerName and a version greater than 0.

Your first result document should be:

```json
{
  "id": "14026",
  "description": "Beverages, Energy Drink, sugar-free with guarana",
  "tags": [
    {
      "name": "beverages"
    },
    {
      "name": "energy drink"
    },
    {
      "name": "sugar-free with guarana"
    }
  ],
  "foodGroup": "Beverages",
  "manufacturerName": "The Coca-Cola Company",
  "version": 1
}
```

You should note that where the query returned the results of tags node it projected the entire contents of the property which in this case is an array.

Advanced projection

Azure Cosmos DB supports several forms of transformation on the resultant JSON. One of the simplest is to alias your JSON elements using the AS aliasing keyword as you project your results.

By running the query below you will see that the element names are transformed. In addition, the projection is accessing only the first element in the servings array for all items specified by the WHERE clause.

Run the below query by selecting the **New SQL Query**.

Paste the following SQL query and then select **Execute Query**.

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

ORDER BY clause

Azure Cosmos DB supports adding an ORDER BY clause to sort results based on one or more properties

Run the below query by selecting the **New SQL Query**.

Paste the following SQL query and then select **Execute Query**.

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

You can learn more about configuring the required indexes for an Order By clause in the later Indexing Lab or by reading our docs.

Limiting query result size

Azure Cosmos DB supports the TOP keyword. TOP can be used to limit the number of returning values from a query.

Run the query below to see the top 20 results.

```
SELECT TOP 20 food.id,
food.description,
food.tags,
food.foodGroup
FROM food
WHERE food.foodGroup = "Snacks"
```

The OFFSET LIMIT clause is an optional clause to skip then take some number of values from the query. The OFFSET count and the LIMIT count are required in the OFFSET LIMIT clause.

```
SELECT food.id,
food.description,
food.tags,
food.foodGroup
FROM food
WHERE food.foodGroup = "Snacks"
ORDER BY food.id
OFFSET 10 LIMIT 10
```

When OFFSET LIMIT is used in conjunction with an ORDER BY clause, the result set is produced by doing skip and take on the ordered values. If no ORDER BY clause is used, it will result in a deterministic order of values.

More advanced filtering

Let's add the IN and BETWEEN keywords into our queries. IN can be used to check whether a specified value matches any element in a given list and BETWEEN can be used to run queries against a range of values.

Run the query below:

```
SELECT food.id,
       food.description,
       food.tags,
       food.foodGroup,
       food.version
FROM food
WHERE food.foodGroup IN ("Poultry Products", "Sausages and Luncheon Meats")
    AND (food.id BETWEEN "05740" AND "07050")
```

More advanced projection

Azure Cosmos DB supports JSON projection within its queries. Let's project a new JSON Object with modified property names.

Run the query below to see the results.

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

JOIN within your documents

Azure Cosmos DB's JOIN supports intra-document and self-joins. Azure Cosmos DB does not support JOINs across documents or containers.

In an earlier query example we returned a result with attributes of just the first serving of the food.servings array. By using the join syntax below, we can now return an item in the result for every item within the serving array while still being able to project the attributes from elsewhere in the item.

Run the query below to iterate on the food document's servings.

```
SELECT
food.id as FoodID,
serving.description as ServingDescription
FROM food
JOIN serving IN food.servings
WHERE food.id = "03226"
```

JOINs are useful if you need to filter on properties within an array. Run the below example that has filter after the intra-document JOIN.

```
SELECT VALUE COUNT(1)
FROM c
JOIN t IN c.tags
JOIN s IN c.servings
WHERE t.name = 'infant formula' AND s.amount > 1
```

System functions

Azure Cosmos DB supports a number of built-in functions for common operations. They cover mathematical functions like ABS, FLOOR and ROUND and type checking functions like IS_ARRAY, IS_BOOL and IS_DEFINED. Learn more about supported system functions.

Run the query below to see example use of some system functions

```
SELECT food.id,
food.commonName,
food.foodGroup,
ROUND(nutrient.nutritionValue) AS amount,
nutrient.units
FROM food JOIN nutrient IN food.nutrients
WHERE IS_DEFINED(food.commonName)
AND nutrient.description = "Water"
AND food.foodGroup IN ("Sausages and Luncheon Meats", "Legumes and Legume Products")
AND food.id > "42178"
```

Correlated sub-queries

In many scenarios, a sub-query may be effective. A correlated sub-query is a query that references values from an outer query. We will walk through some of the most useful examples here. You can learn more about subqueries.

There are two types of sub-queries: Multi-value sub-queries and scalar sub-queries. Multi-value sub-queries return a set of documents and are always used within the FROM clause. A scalar sub-query expression is a sub-query that evaluates to a single value.

Multi-value subqueries

You can optimize JOIN expressions with a sub-query.

Consider the following query which performs a self-join and then applies a filter on name, nutritionValue, and amount. We can use a sub-query to filter out the joined array items before joining with the next expression.

```
SELECT VALUE COUNT(1)
FROM c
JOIN t IN c.tags
JOIN n IN c.nutrients
JOIN s IN c.servings
WHERE t.name = 'infant formula' AND (n.nutritionValue > 0
AND n.nutritionValue < 10) AND s.amount > 1
```

We could rewrite this query using three sub-queries to optimize and reduce the Request Unit (RU) charge. Observe that the multi-value sub-query always appears in the FROM clause of the outer query.

```
SELECT VALUE COUNT(1)
FROM c
JOIN (SELECT VALUE t FROM t IN c.tags WHERE t.name = 'infant formula')
JOIN (SELECT VALUE n FROM n IN c.nutrients WHERE n.nutritionValue > 0 AND n.nutritionValue < 10)
JOIN (SELECT VALUE s FROM s IN c.servings WHERE s.amount > 1)
```

Scalar sub-queries

One use case of scalar sub-queries is rewriting ARRAY_CONTAINS as EXISTS.

Consider the following query that uses ARRAY_CONTAINS:

```
SELECT TOP 5 f.id, f.tags
FROM food f
WHERE ARRAY_CONTAINS(f.tags, {name: 'orange'})
```

Run the following query which has the same results but uses EXISTS:

```
SELECT TOP 5 f.id, f.tags
FROM food f
WHERE EXISTS(SELECT VALUE t FROM t IN f.tags WHERE t.name = 'orange')
```

The major advantage of using EXISTS is the ability to have complex filters in the EXISTS function, rather than just the simple equality filters which ARRAY_CONTAINS permits.

Here is an example:

```
SELECT VALUE c.description FROM c

JOIN n IN c.nutrients
WHERE n.units= "mg" AND n.nutritionValue > 0
AND EXISTS(SELECT VALUE t FROM t IN c.tags WHERE t.name = 'orange')
```

Lab 3: Indexing in Azure Cosmos DB

In this lab, you will modify the indexing policy of an Azure Cosmos DB container. You will explore how you can optimize indexing policy for write or read heavy workloads as well as understand the indexing requirements for different SQL API query features.

Indexing Overview

Azure Cosmos DB is a schema-agnostic database that allows you to iterate on your application without having to deal with schema or index management. By default, Azure Cosmos DB automatically indexes every property for all items in your container without the need to define any schema or configure secondary indexes. If you chose to leave indexing policy at the default settings, you can run most queries with optimal performance and never have to explicitly consider indexing. However, if you want control over adding or removing properties from the index, modification is possible through the Azure Portal or any SQL API SDK.

Azure Cosmos DB uses an inverted index, representing your data in a tree form. For a brief introduction on how this works, read our indexing overview before continuing with the lab.

Customizing the indexing policy

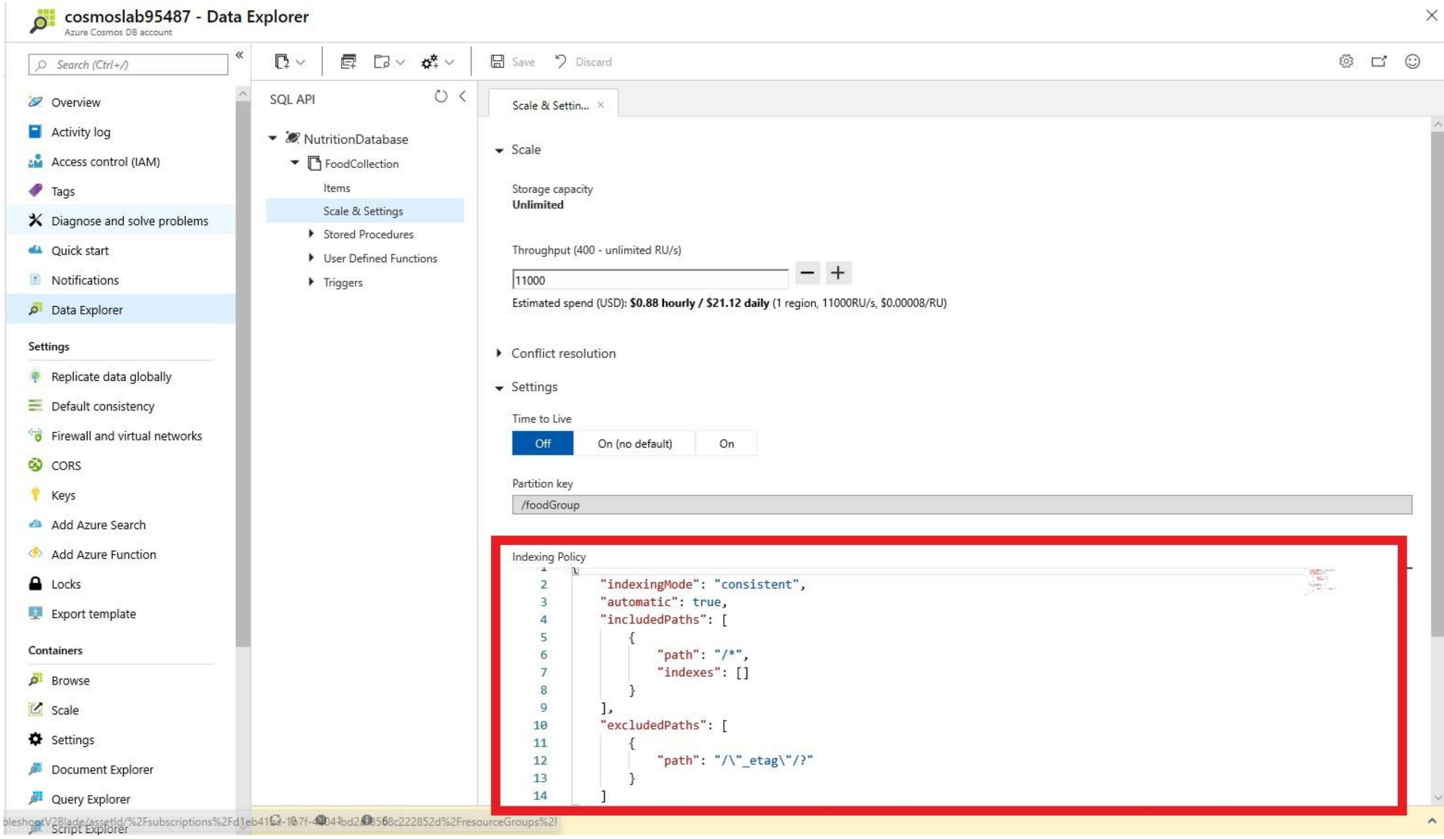
In this lab section, you will view and modify the indexing policy for your **FoodCollection**.

Open Data Explorer

1. On the left side of the portal, select the **Resource groups** link.
2. In the **Resource groups** blade, locate and select the resource group created for this lab.
3. In the resource group, select your **Azure Cosmos DB** account (ata-cosmos-<YourName>-account).
4. In the **Azure Cosmos DB** blade, locate and select the **Data Explorer** link on the left side of the blade.
5. In the **Data Explorer** section, expand the **NutritionDatabase** database node and then expand the **FoodCollection** container node.
6. Within the **FoodCollection** node, select the **Items** link.
7. View the items within the container. Observe how these documents have many properties, including arrays. If we do not use a particular property in the WHERE clause, ORDER BY clause, or a JOIN, indexing the property does not provide any performance benefit.
8. Still within the **FoodCollection** node, select the **Scale & Settings** link.

9. Review the **Indexing Policy** section

    - Notice you can edit the JSON file that defines your container's index
    - An Indexing policy can also be modified through any Azure Cosmos DB SDK
    - During this lab we will modify the indexing policy through the Azure Portal



Storage capacity
Unlimited

Throughput (400 - unlimited RU/s)

11000

Estimated spend (USD): **$0.88 hourly / $21.12 daily** (1 region, 11000RU/s, $0.00008/RU)

Time to Live

Off | On (no default) | On

Partition key

/foodGroup

Indexing Policy

```
    "indexingMode": "consistent",
    "automatic": true,
    "includedPaths": [
        {
            "path": "/*",
            "indexes": []
        }
    ],
    "excludedPaths": [
        {
            "path": "/\"_etag\"/?"
        }
    ]
```

Including and excluding Range Indexes

Instead of including a range index on every property by default, you can choose to either include or exclude specific paths from the index. Let's go through some simple examples (no need to enter these into the Azure Portal, we can just review them here).

Within the **FoodCollection**, documents have this schema (some properties were removed for simplicity):

```
{
    "id": "36000",
    "_rid": "LYwNAKzLG9ADAAAAAAAAAA==",
    "_self": "dbs/LYwNAA==/colls/LYwNAKzLG9A=/docs/LYwNAKzLG9ADAAAAAAAAAA==/",
    "_etag": "\"0b008d85-0000-0700-0000-5d1a47e60000\"",
    "description": "APPLEBEE'S, 9 oz house sirloin steak",
    "tags": [
        {
            "name": "applebee's"
        },
        {
            "name": "9 oz house sirloin steak"
        }
    ],

    "manufacturerName": "Applebee's",
    "foodGroup": "Restaurant Foods",
    "nutrients": [
        {
            "id": "301",
            "description": "Calcium, Ca",
            "nutritionValue": 16,
            "units": "mg"
        },
        {
            "id": "312",
            "description": "Copper, Cu",
            "nutritionValue": 0.076,
            "units": "mg"
        },
    ]
}
```

If you wanted to only index the manufacturerName, foodGroup, and nutrients array with a range index, you should define the following index policy. In this example, we use the wildcard character `*` to indicate that we would like to index all paths within the nutrients array.

```
{
        "indexingMode": "consistent",
        "includedPaths": [
            {
                "path": "/manufacturerName/*"
            },
            {
                "path": "/foodGroup/*"
            },
            {
                "path": "/nutrients/[]/*"
            }
        ],
        "excludedPaths": [
            {
                "path": "/*"
            }
        ]
    }
```

However, it's possible we may just want to index the nutritionValue of each array element.

In this next example, the indexing policy would explicitly specify that the nutritionValue path in the nutrition array should be indexed. Since we don't use the wildcard character `*`, no additional paths in the array are indexed.

```
{
        "indexingMode": "consistent",
        "includedPaths": [
            {
                "path": "/manufacturerName/*"
            },
            {
                "path": "/foodGroup/*"
            },
            {
                "path": "/nutrients/[]/nutritionValue/*"
            }
        ],
        "excludedPaths": [
            {
                "path": "/*"
            }
        ]
    }
```

Finally, it's important to understand the difference between the `*` and `?` characters.
The `*` character indicates that Azure Cosmos DB should index every path beyond that specific node.
The `?` character indicates that Azure Cosmos DB should index no further paths beyond this node.
In the above example, there are no additional paths under nutritionValue. If we were to modify the document and add a path here, having the wildcard character `*` in the above example would ensure that the property is indexed without explicitly mentioning the name.

Understand query requirements

Before modifying indexing policy, it's important to understand how the data is used in the collection.

If your workload is write-heavy or your documents are large, you should only index necessary paths. This will significantly decrease the amount of RU's required for inserts, updates, and deletes.

Let's imagine that the following queries are the only read operations that are executed on the **FoodCollection** container.

Query #1

```
SELECT * FROM c WHERE c.manufacturerName = <manufacturerName>
```

Query #2

```
SELECT * FROM c WHERE c.foodGroup = <foodGroup>
```

These queries only require that a range index be defined on **manufacturerName** and **foodGroup**, respectively. We can modify the indexing policy to index only these properties.

Edit the indexing policy by including paths

1. In the Azure Portal, navigate back to the **FoodCollection** container
2. Select the **Scale & Settings** link
3. In the **Indexing Policy** section, replace the existing json file with the following:

```
{
    "indexingMode": "consistent",
    "includedPaths": [
        {
            "path": "/manufacturerName/*"
        },
        {
            "path": "/foodGroup/*"
        }
    ],
    "excludedPaths": [
        {
            "path": "/*"
        }
    ]
}
```

This new indexing policy will create a range index on only the manufacturerName and foodGroup properties. It will remove range indexes on all other properties.

19. Select **Save**. Azure Cosmos DB will update the index in the container, using your excess provisioned throughput to make the updates.

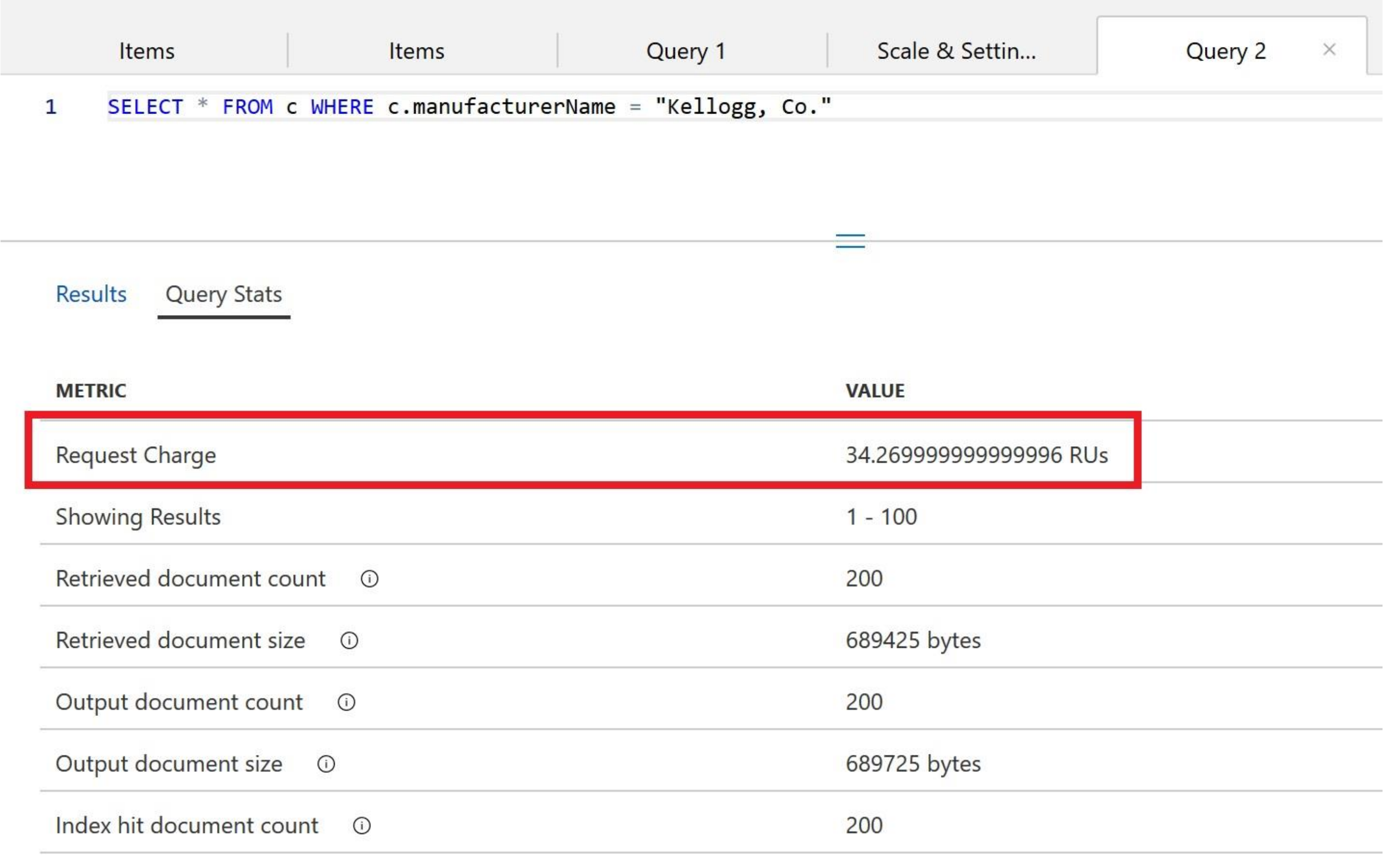
    During the container re-indexing, write performance is unaffected. However, queries may return incomplete results.

20. In the menu, select the **New SQL Query** icon.

21. Paste the following SQL query and select **Execute Query**:

    ```
    SELECT * FROM c WHERE c.manufacturerName = "Kellogg, Co."
    ```

22. Navigate to the **Query Stats** tab. You should observe that this query still has a low RU charge, even after removing some properties from the index. Because the **manufacturerName** was the only property used as a filter in the query, it was the only index that was required.



```
1   SELECT * FROM c WHERE c.manufacturerName = "Kellogg, Co."
```

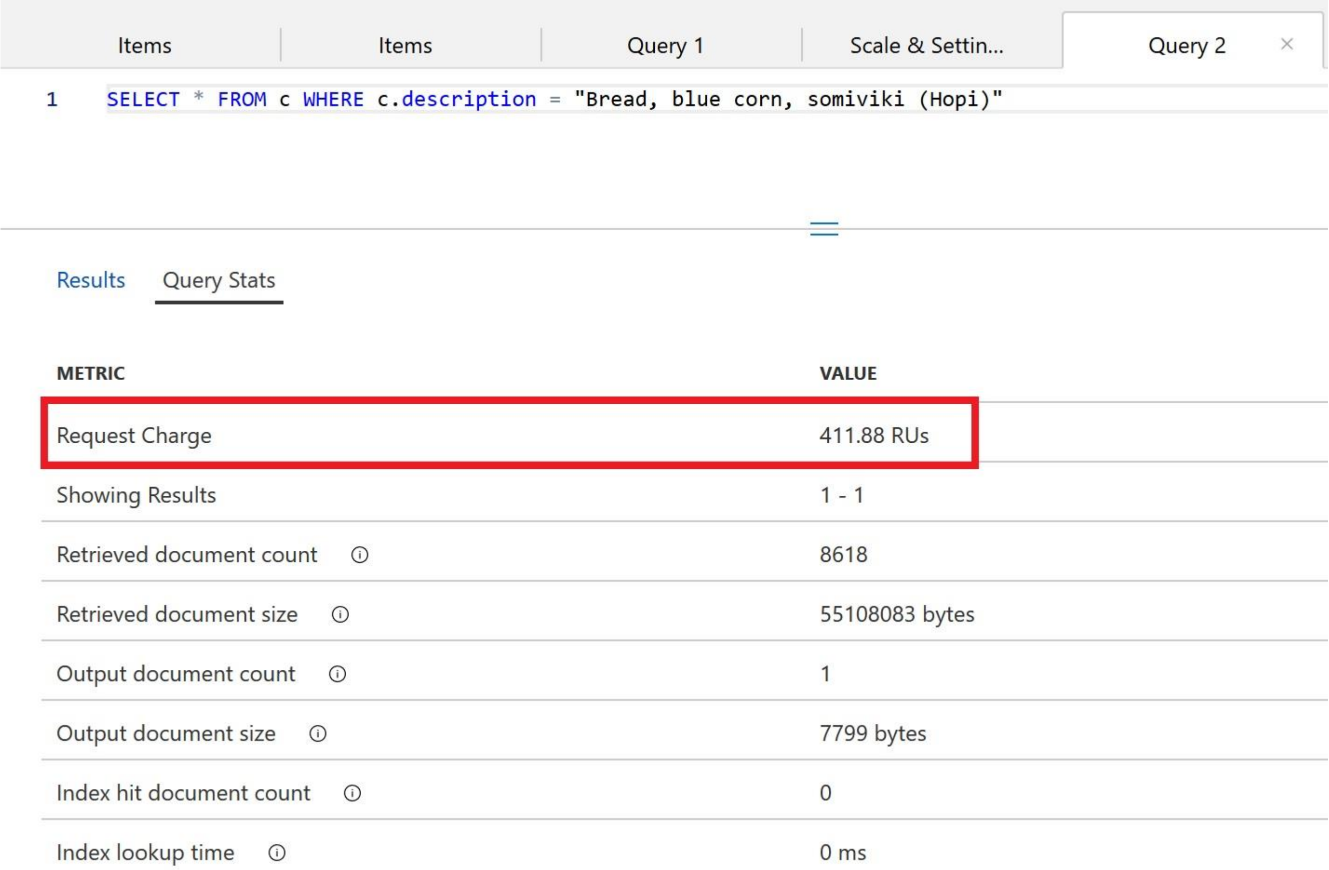
METRIC	VALUE
Request Charge	34.269999999999996 RUs
Showing Results	1 - 100
Retrieved document count ⓘ	200
Retrieved document size ⓘ	689425 bytes
Output document count ⓘ	200
Output document size ⓘ	689725 bytes
Index hit document count ⓘ	200

23. Replace the query text with the following and select **Execute Query**:

    ```
    SELECT * FROM c WHERE c.description = "Bread, blue corn, somiviki (Hopi)"
    ```

24. Observe that this query has a very high RU charge even though only a single document is returned. This is because no range index is currently defined for the `description` property.

25. Observe the **Query Metrics**:



```
SELECT * FROM c WHERE c.description = "Bread, blue corn, somiviki (Hopi)"
```

METRIC	VALUE
Request Charge	411.88 RUs
Showing Results	1 - 1
Retrieved document count	8618
Retrieved document size	55108083 bytes
Output document count	1
Output document size	7799 bytes
Index hit document count	0
Index lookup time	0 ms

If a query does not use the index, the **Index hit document count** will be 0. We can see above that the query needed to retrieve 5,187 documents and ultimately ended up only returning 1 document.

Edit the indexing policy by excluding paths

In addition to manually including certain paths to be indexed, you can exclude specific paths. In many cases, this approach can be simpler since it will allow all new properties in your document to be indexed by default. If there is a property that you are certain you will never use in your queries, you should explicitly exclude this path.

We will create an indexing policy to index every path except for the **description** property.

1. Navigate back to the **FoodCollection** in the Azure Portal

2. Select the **Scale & Settings** link

3. In the **Indexing Policy** section, replace the existing json file with the following:

   ```
   {
       "indexingMode": "consistent",
       "includedPaths": [
           {
               "path": "/*"
           }
       ],
       "excludedPaths": [
           {
               "path": "/description/*"
           }
       ]

   }
   ```

   This new indexing policy will create a range index on every property **except** for the description.

4. Select **Save**. Azure Cosmos DB will update the index in the container, using your excess provisioned throughput to make the updates.

   During the container re-indexing, write performance is unaffected. However, queries may return incomplete results.

5. After defining the new indexing policy, navigate to your **FoodCollection** and select the **Add New SQL Query** icon. Paste the following SQL query and select **Execute Query**:

   ```
   SELECT * FROM c WHERE c.manufacturerName = "Kellogg, Co."
   ```

6. Navigate to the **Query Stats** tab. You should observe that this query still has a low RU charge since manufacturerName is indexed.

7. Replace the query text with the following and select **Execute Query**:

   ```
   SELECT * FROM c WHERE c.description = "Bread, blue corn, somiviki (Hopi)"
   ```

8. Observe that this query has a very high RU charge even though only a single document is returned. This is because the `description` property is explicitly excluded in the indexing policy.

Adding a Composite Index

For ORDER BY queries that order by multiple properties, a composite index is required. A composite index is defined on multiple properties and must be manually created.

1. In the **Azure Cosmos DB** blade, locate and select the **Data Explorer** link on the left side of the blade.

2. In the **Data Explorer** section, expand the **NutritionDatabase** database node and then expand the **FoodCollection** container node.

3. Select the icon to add a **New SQL Query**.

4. Paste the following SQL query and select **Execute Query**

   ```
   SELECT * FROM c ORDER BY c.foodGroup ASC, c.manufacturerName ASC
   ```

   This query will fail with the following error:

   ```
   "The order by query does not have a corresponding composite index that it can be served
   from."
   ```

   In order to run a query that has an ORDER BY clause with one property, the default range index is sufficient. Queries with multiple properties in the ORDER BY clause require a composite index.

5. Still within the **FoodCollection** node, select the **Scale & Settings** link. In the **Indexing Policy** section, you will add a composite index.

6. Replace the **Indexing Policy** with the following text:

```
{
    "indexingMode": "consistent",
    "automatic": true,
    "includedPaths": [
        {
            "path": "/manufacturerName/*"
        },
        {
            "path": "/foodGroup/*"
        }
    ],
    "excludedPaths": [
        {
            "path": "/*"
        },
        {
            "path": "/\"_etag\"/?"
        }
    ],
    "compositeIndexes": [
        [
            {
                "path": "/foodGroup",
                "order": "ascending"
            },
            {
                "path": "/manufacturerName",
                "order": "ascending"
            }
        ]
    ]
}
```

38. **Save** this new indexing policy. The update should take approximately 10-15 seconds to apply to your container.

    This indexing policy defines a composite index that allows for the following ORDER BY queries. Test each of these by running them in your existing open query tab in the **Data Explorer**. When you define the order for properties in a composite index, they must either exactly match the order in the ORDER BY clause or be, in all cases, the opposite value.

39. Run the following queries.

```sql
SELECT * FROM c ORDER BY c.foodGroup ASC, c.manufacturerName ASC
```

40. Run the following query, which the current composite index does not support.

```sql
SELECT * FROM c ORDER BY c.foodGroup DESC, c.manufacturerName ASC
```

41. This query will not run without an additional composite index. Modify the indexing policy to include an additional composite index.

```json
{
    "indexingMode": "consistent",
    "automatic": true,
    "includedPaths": [
        {
            "path": "/manufacturerName/*"
        },
        {
            "path": "/foodGroup/*"
        }
    ],
    "excludedPaths": [
        {
            "path": "/*"
        },
        {
            "path": "/\"_etag\"/?"
        }
    ],
    "compositeIndexes": [
        [
            {
                "path": "/foodGroup",
                "order": "ascending"
            },
            {
                "path": "/manufacturerName",
                "order": "ascending"
            }
        ],
        [
            {
                "path": "/foodGroup",
                "order": "descending"
            },
            {
                "path": "/manufacturerName",
                "order": "ascending"
            }
        ]
    ]
}
```

83. Re-run the query, it should succeed.

Learn more about defining composite indexes.