**Module Introduction - Warm Storage**

In this module, you will learn about

* The role of warm storage in an IoT architecture
* Cosmos DB as a warm storage technology
* Integrating business applications with Cosmos DB

This module introduces you to the role warm storage plays in an IoT architecture. You get hands-on training with Azure Cosmos DB – a recommended technology for warm storage. You learn about the process of populating and querying Cosmos DB, and you learn about integrating Cosmos DB with an Azure Logic App

During this module, you will complete the following hands-on labs:

* Lab 1: Getting started with warm storage
* Lab 2: Implementing Business System integration

**Provision Cosmos DB**

**Provision Cosmos DB**

In this task, you will provision Cosmos DB, which you will use to examine warm storage concepts. The process will involve creating a Cosmos DB account, then a database, and finally a collection that will store simulated wind turbine data.

Cosmos DB is a very versatile and complex platform, so we will not come close to covering all the possible configurations and features. Instead, you will provision a straightforward test instance for warm storage purposes. Along the way, we will mention some of the configuration options that are possible, even if they are not part of the test configuration we build.

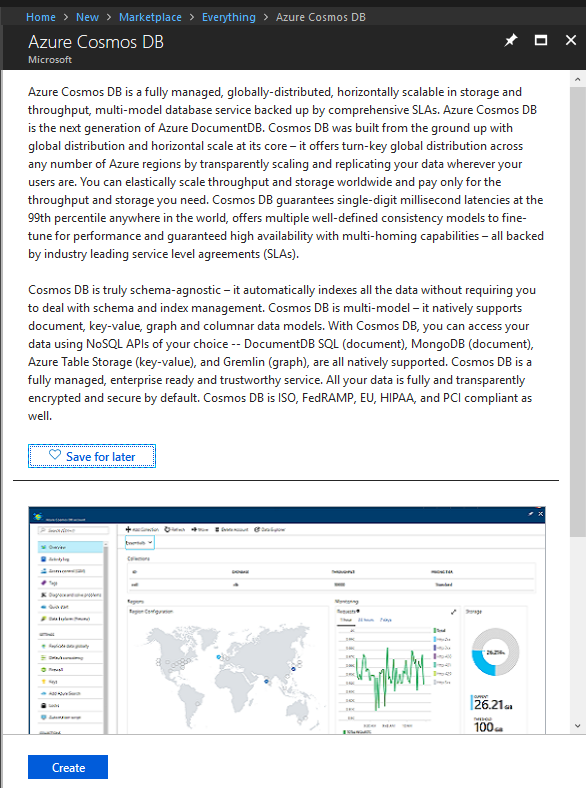
**Note:** You will sometimes see Cosmos DB referred to as DocumentDB in documentation, or on the Azure Portal. That technology was the precursor to Cosmos DB, and Cosmos DB retains backwards compatibility to DocumentDB.

**Create a Cosmos DB Account**

1. In your Web browser, to open your Azure portal, navigate to the [portal.azure.com](http://portal.azure.com).

When you log into Azure you will arrive at the Azure portal home page for your Azure account.

1. In the top-left corner of your Azure portal, click **+Create a resource**
2. In the Search box, type **Cosmos**
3. Press Enter.
4. In the list of filtered results, click **Azure Cosmos DB**
5. After reading through the text description, click **Create**



1. On the **New Account** blade, enter a unique name to use for the **ID** field.

Something like **iotstoragemod02XXcosmosDDDDDD** (where **XX** represents your initials and **DDDDDD** represents the current date in numbers only).

1. Under **API**, select **SQL**

Cosmos gives you several API choices as a multi-model storage solution. We will just be working with the SQL API for this module.

1. Under **Subscription**, select the subscription that you are using for this course.
2. Under **Resource Group**, select **Create new** and enter the name of the resource oup.

A good name to use would be the course ID - **DEV326-RG**

1. Under **Location**, select a region location that is near you.
2. make sure **Enable geo-redundancy** is unchecked.

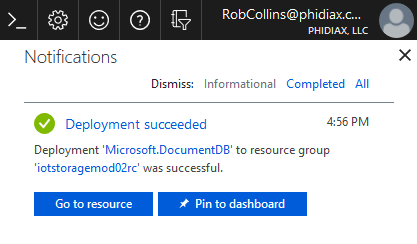
In most production configurations **Enable geo-redundancy** would be selected. One of Cosmos DB’s biggest selling points is its availability across the world and its ability to fulfill requests with low latency across regions. That enables flexibility for all types of topologies and architectures. Even if an application does not have clients across geographies, having geo-redundancy provides more data safety. In our test case, it is not necessary, especially given its cost.

1. Click **Create**

After a few minutes processing period, you will get a notification that your Cosmos DB has been deployed successfully.

1. On the Notifications pane, to navigate to the **Cosmos DB account overview** pane.
2. Click **Go to resource**

You can also use your Resource group to open your Azure Cosmos DB account.

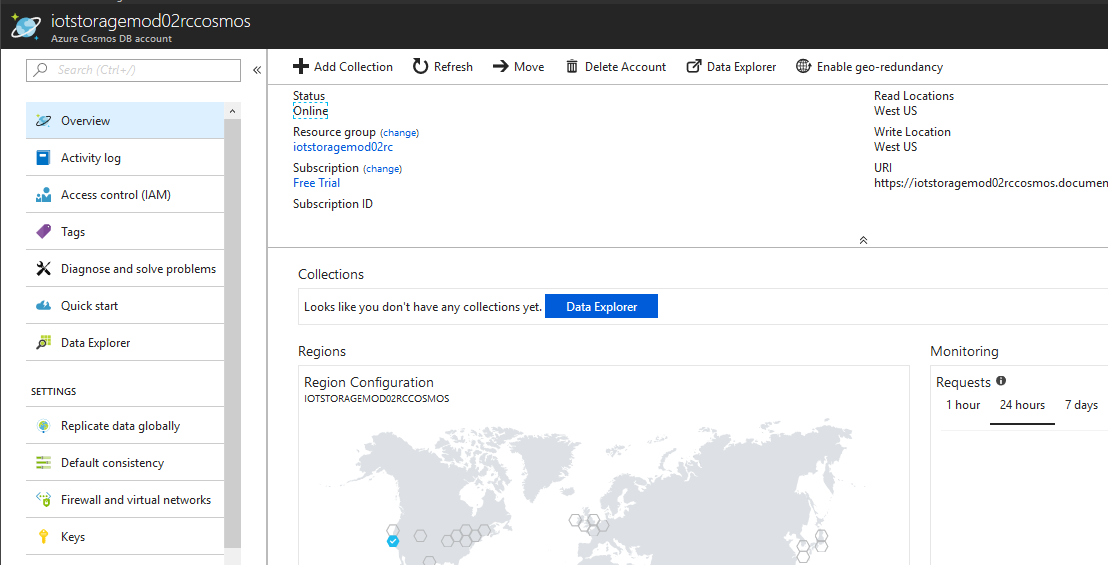


**Note**: At the time of writing, the notification refers to the legacy name **Microsoft.DocumentDB** rather than **Cosmos DB**

1. On the **Cosmos DB Account** blade, in the left hand navigation area, click **Overview**
2. On the **Overview** pane, under **Regions** notice a World Map.

An icon indicates which region(s) your account is deployed to.

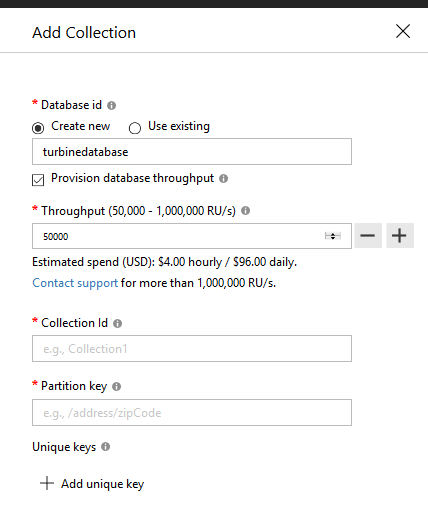
**Note**: It is possible to enable geo-redundancy and add other regions, even after you have provisioned your Cosmos DB account.



**Add a Collection**

In Cosmos DB, data is stored within a collection. Complete the following steps to add a collection.

1. At the top of the **Overview** pane, click **+ Add Collection**
2. On the **Add Collection** blade, in **Database id**, enter **turbinedatabase**



You have the option on this screen to not only add a collection, but to add a database as well. The database name only has to be unique in the context of your account.

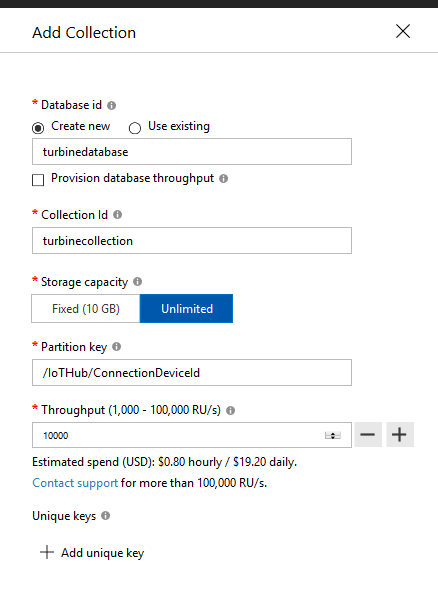
The concept of **Request Units** is an important one for provisioning Cosmos DB. Since Cosmos DB is a completely managed cloud solution, you don’t need to reserve read/write capacities or provision CPU, memory, and IOPS. Instead, you reserve a number of guaranteed request units to be available to your application on a per-second basis. Each operation in Azure Cosmos DB, including writing a document, performing a query, and updating a document, consumes resources, which are distilled to a single number – request units per second.

The concept may feel unfamiliar, but it replaces many other, more difficult capacity planning concepts.

We will not go too deep into RU provisioning in this course, but take notice of one thing on the **Add Collection** screen. If you click the **Provision database throughput** checkbox, you will be provisioning throughput on a database level. This is generally used in production setups with massive scale. The minimum you can choose is 50,000 RU/second (or 180,000,000 1KB writes per hour). There is some other nuance to that throughput number, but suffice it to say that it is much more than we would need for our test scenario. We will provision RUs on the collection level. So make certain the **Provision database throughput** checkbox is unchecked.

1. In **Collection Id**, enter **turbinecollection**

This will be the storage artifact which contains all your data.



1. Under **Storage Capacity**, click **Fixed (10 GB)**

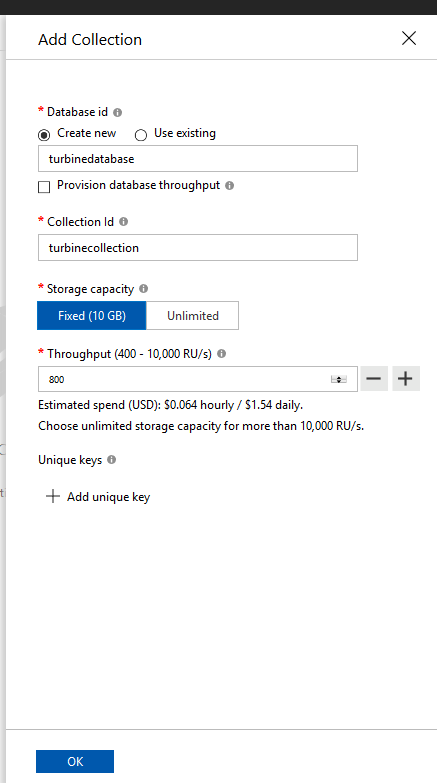
Most production warm storage scenarios would require **Unlimited**. In that configuration, Azure Cosmos DB manages the scaling of your data and allocates storage as it grows.

Also notice that when you choose **Fixed**, you no longer need to define a **Partition key** (in fact, the UI element disappears). A partition key is a value used for logically grouping data for the purpose of making more efficient queries. This also allows for allowing horizontal scaling of data. Using wind farm data as an example, the unique turbine identifier (ConnectionDeviceId in our case) would make a good partition key. However, as we don't need **Unlimited** storage for the test scenario, we don't need to supply a **Partition key**.

1. Under **Throughput**, enter **800**

With a fixed capacity collection, you can use lower minimums for RUs.

1. Click **OK**

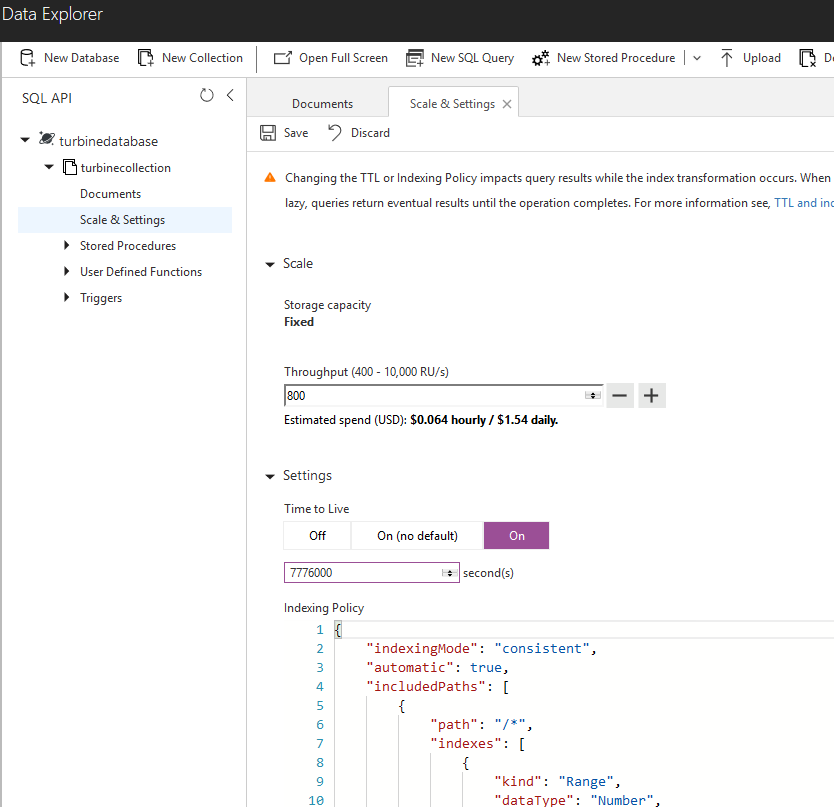


After a short delay, the database and collection will be created. The **Data Explorer** pane will then open and you will see a tree view of the new **turbinedatabase** and **turbinecollection**

1. In the **Data Explorer** pane, click on **turbinecollection**

This expands the **turbine collection** node.

1. Click **Scale & Settings**



One of Cosmos DB’s features is a **Time to Live** (TTL) property that you can apply across an entire database. That is, a record will be automatically deleted a certain amount of time after it was last inserted or updated.

This is a handy feature when using Cosmos DB as warm storage. You generally don’t want to retain information indefinitely in warm storage, so you can leverage the TTL to automatically remove old data.

1. On the **Scale & Settings** tab, under **Settings**, in **Time to live**, click **On**
2. Under **Time to live**, in **second(s)**, enter **7776000**

In the real world, you would use a value that is relevant to your use case, however, in this lab we will use 7776000 seconds (three months). That could be a realistic TTL for IOT warm storage.

1. Click **Save**

This applies your change to the TTL.

**Add a Document**

One of the fundamental traits of Cosmos DB is its usage of JSON documents for storage. You can use the **Data Explorer** pane to add a document and get a sense of how that works.

1. To expand the **turbinedatabase** node, on the **Data Explorer** pane, click **turbinedatabase**
2. To expand the **turbinecollection** node, under **turbinedatabase**, click **turbinecollection**
3. Under **turbinecollection**, click **Documents**
4. In the **Documents** tab, click **New Document**

You will see a placeholder JSON document that simply has an id property.

{

"id": "replace\_with\_new\_document\_id"

}

That underlying artifact – the JSON document – does not have a fixed schema. All it really requires is a unique id property. You could replace the placeholder id property with a unique string and insert it into your Cosmos DB collection. You would have a valid document that you could query. But let's use a richer JSON document.

1. In the **Documents** tab, replace the existing JSON:
2. {
3. "id": "replace\_with\_new\_document\_id"
4. }

With the following:

{

"id": "636557E0-15BE-4773-AD88-976A0E989FE1",

"bearingstemperature": -8.84585293421794,

"windingstemperature": -8.7767400470454,

"towersway": 33.066986251188,

"positionsensor": 14.4828496875627,

"bladestraingauge": 3186.4525118896,

"mainshaftstraingauge": 8388.65394144722,

"shroudaccelerometer": 1540.42864075882,

"gearboxfluid levels": 995,

"powergeneration": 5,

"EventProcessedUtcTime": "2018-06-26T14:53:50.0458380Z",

"PartitionId": 1,

"EventEnqueuedUtcTime": "2018-06-26T12:10:00.9620000Z",

"IoTHub": {

"ConnectionDeviceId": "Simulated.custom.36",

"ConnectionDeviceGenerationId": "636655659591924012",

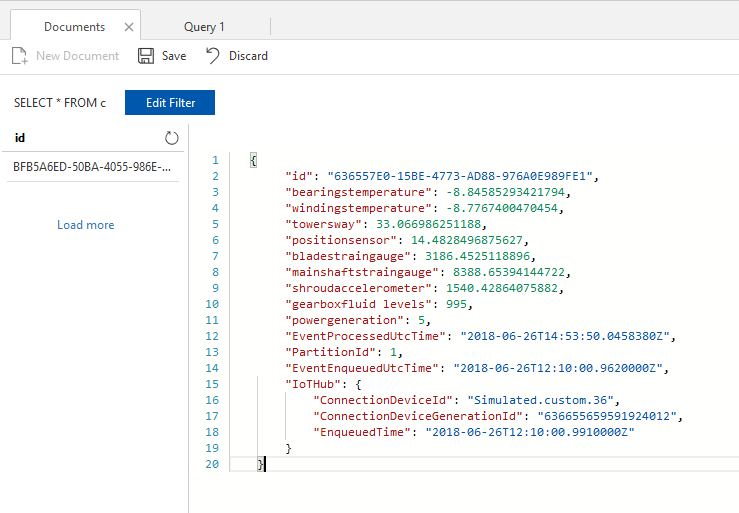
"EnqueuedTime": "2018-06-26T12:10:00.9910000Z"

}

}

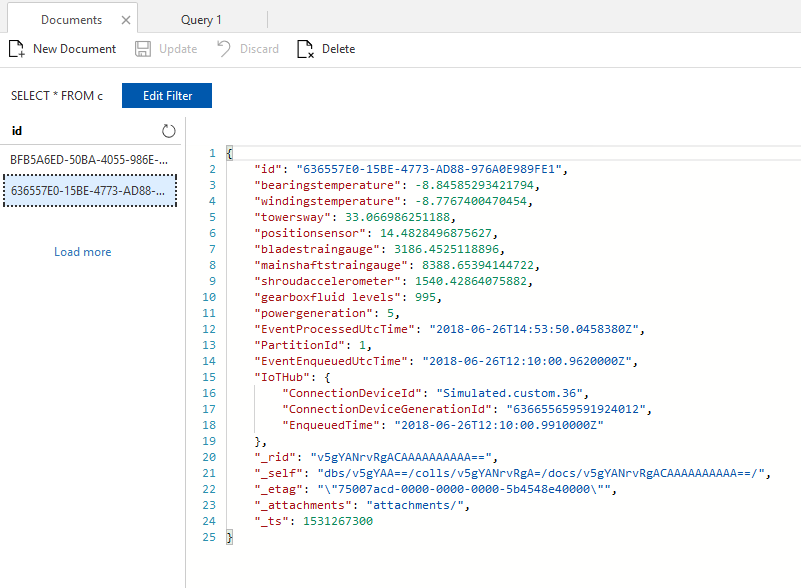
This JSON is similar to the one produced by our device simulator.

1. Click **Save**



Once the document is saved, notice that the id property appears on the left side of the window. You can click it to see the document properties. The document saves properties with various data types – integer, string, double, etc.

Look at the difference between the document you saved and its final form in the database. Cosmos DB adds metadata to each document such \_ts, which is a timestamp.



By default, Cosmos DB will index every property in a JSON document, so you could write a query against any property in your newly created document. Let's try it with the simple id property.

1. Click **New SQL Query**
2. Update the existing query:
3. SELECT \* FROM c

to:

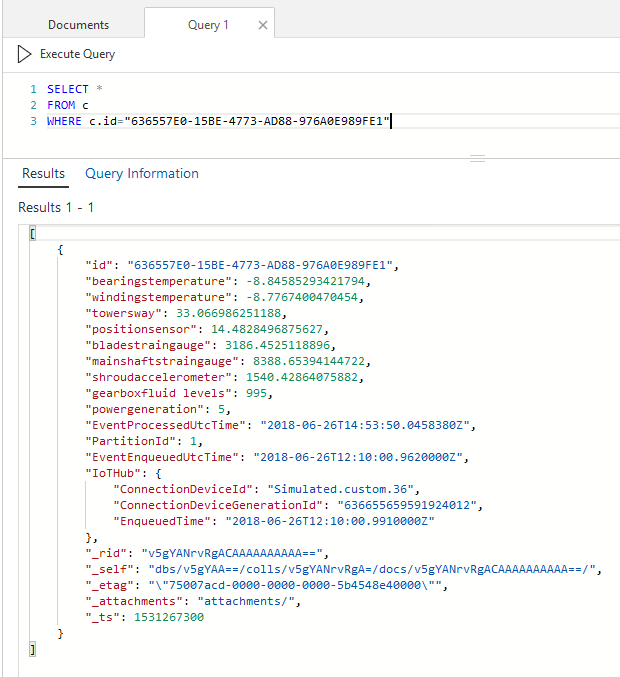
SELECT \*

FROM c

WHERE c.id="636557E0-15BE-4773-AD88-976A0E989FE1"

1. Click **Execute Query**

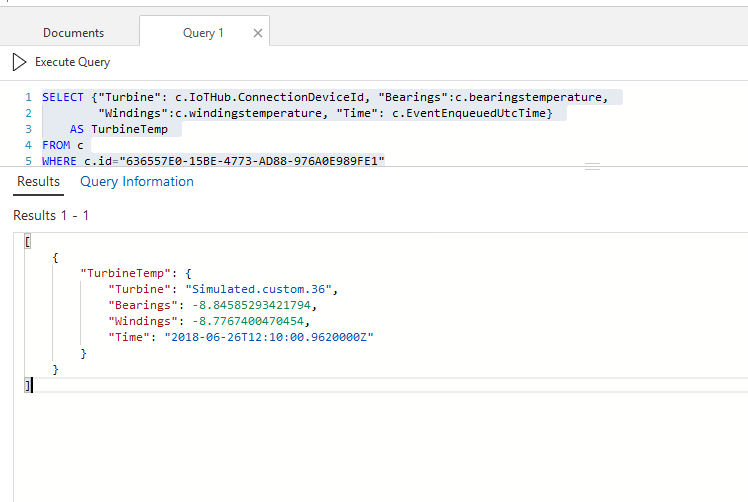
The query will execute and the results are displayed.



Now try with a slightly more sophisticated query of the same record. Instead of SELECT \*, use a SQL query to extract a new JSON document with different property names, a different data shape, and a flattened data structure. The SQL variation that Cosmos DB uses handles JSON data natively. The query will target temperature readings.

1. Replace the query with the following:
2. SELECT {"Turbine": c.IoTHub.ConnectionDeviceId, "Bearings":c.bearingstemperature,
3. "Windings":c.windingstemperature, "Time": c.EventEnqueuedUtcTime}
4. AS TurbineTemp
5. FROM c
6. WHERE c.id="636557E0-15BE-4773-AD88-976A0E989FE1"
7. Click **Execute Query**

The query will execute and the results are displayed



Notice that JSON properties are set and queried directly within the sql statement, such as "Bearings":c.bearingstemperature. The statement also queries a child JSON property, using dot notation: "Turbine": c.IoTHub.ConnectionDeviceId.

You can experiment on queries within the **Document** window. You can further explore Cosmos DB queries on a query playground that Microsoft provides: <https://www.documentdb.com/sql/demo>

In a production scenario, however, you would more likely query Cosmos DB with one of the many client APIs – such as Java, .NET, Node.js or Python.

**Summary**

In this module, you have learned how to provision Cosmos DB to use as warm storage for an IoT architecture. You created an account, added a database, then created a collection for serializing IoT data. As part of the process, you learned about some of the options for capacity planning Cosmos DB using Request Units (RUs). You also examined how Cosmos DB stores data in JSON documents, and some basic techniques for querying Cosmos DB data.