**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.

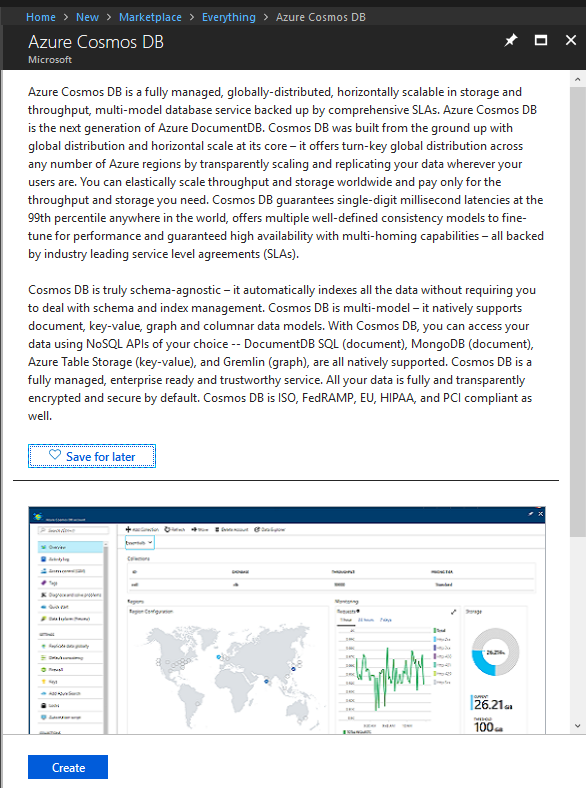
**Note:** You will sometimes see Cosmos DB referred to as DocumentDB in documentation, or on the Azure Portal. That technology was the precursor to Cosmos DB, and Cosmos DB retains backwards compatibility to DocumentDB.

**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.

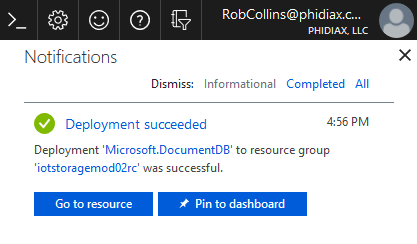
In most production configurations **Enable geo-redundancy** would be selected. One of Cosmos DB’s biggest selling points is its availability across the world and its ability to fulfill requests with low latency across regions. That enables flexibility for all types of topologies and architectures. Even 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.

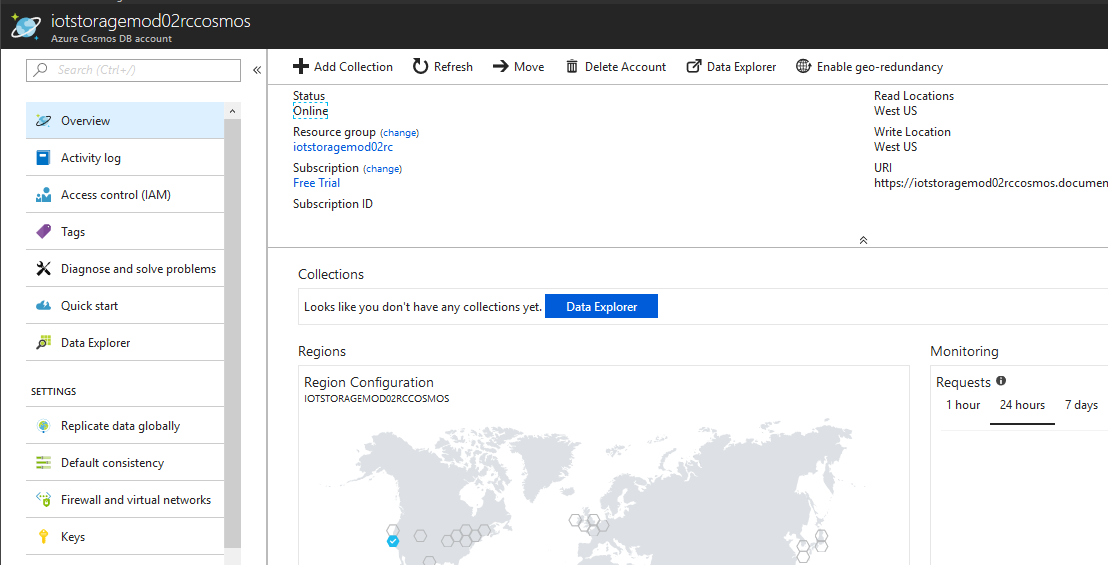


**Note**: At the time of writing, the notification refers to the legacy name **Microsoft.DocumentDB** rather than **Cosmos DB**

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.

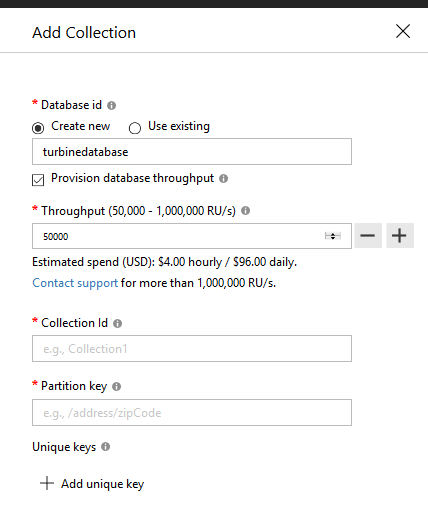
**Note**: It is possible to enable geo-redundancy and add other regions, even after you have provisioned your Cosmos DB account.



**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.

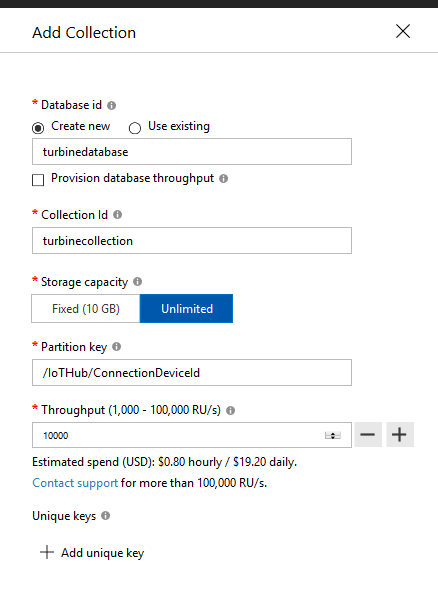
The concept of **Request Units** is an important one for provisioning Cosmos DB. Since Cosmos DB is a completely managed cloud solution, you don’t need to reserve read/write capacities or provision CPU, memory, and IOPS. Instead, you reserve a number of guaranteed request units to be available to your 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)**

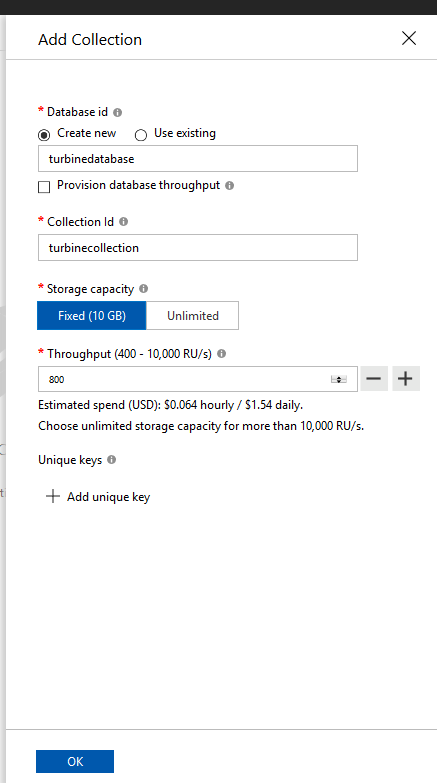
Most production warm storage scenarios would require **Unlimited**. In that configuration, Azure Cosmos DB manages the scaling of your data and allocates storage as it grows.

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**

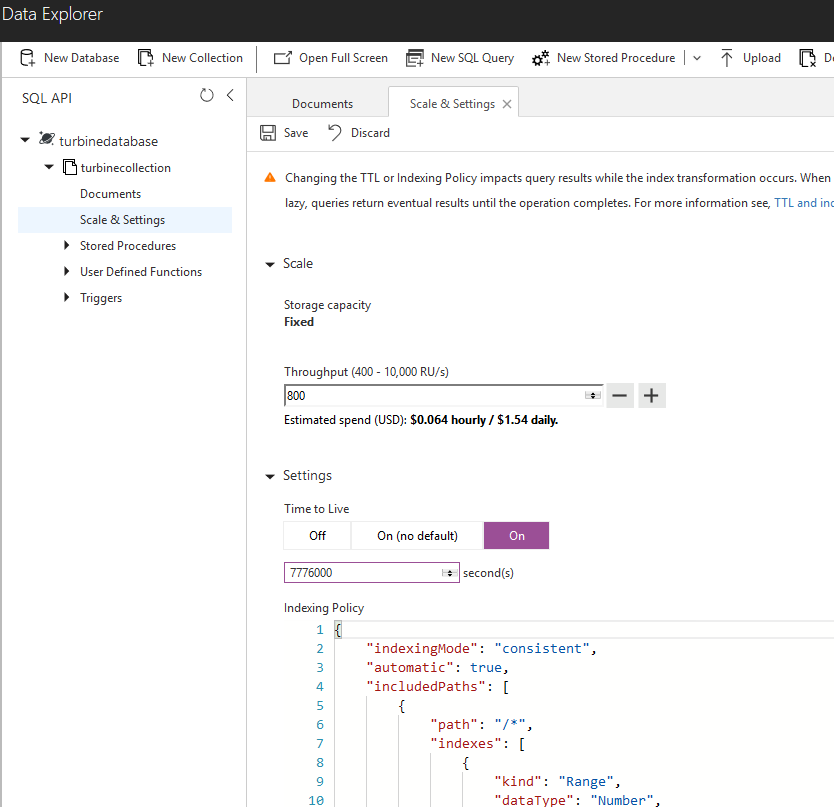


After a short delay, the database and collection will be created. The **Data Explorer** pane will then open and you will see a tree view of the new **turbinedatabase** and **turbinecollection**

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",

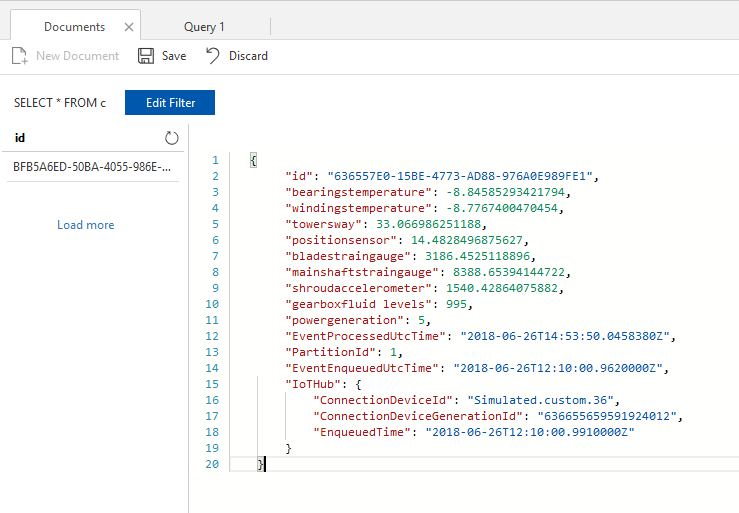
"EnqueuedTime": "2018-06-26T12:10:00.9910000Z"

}

}

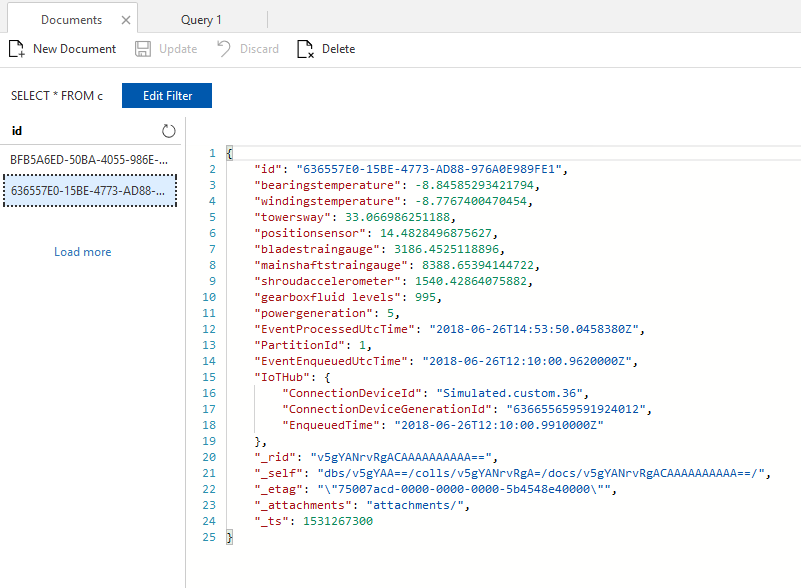
This JSON is similar to the one produced by our device simulator.

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:

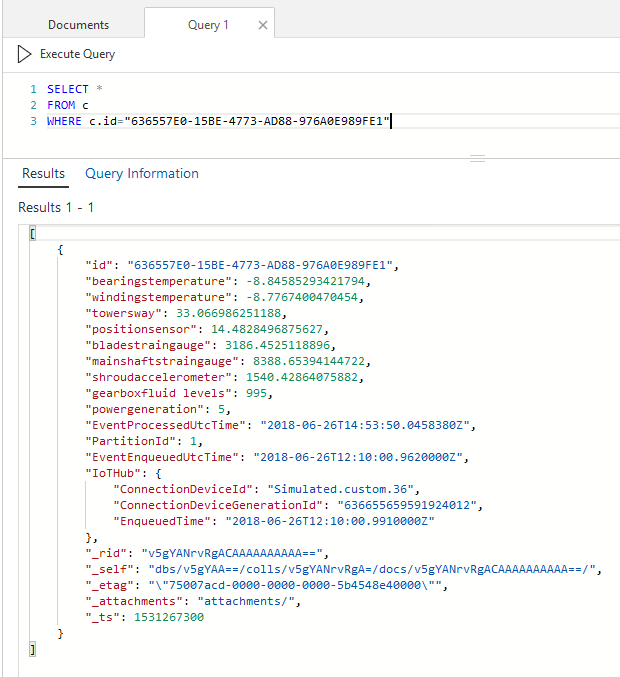
SELECT \*

FROM c

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

1. Click **Execute Query**

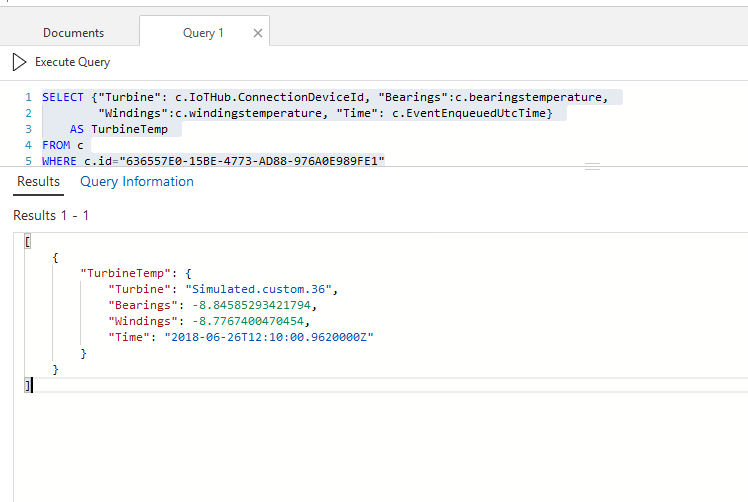
The query will execute and the results are displayed.



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.

**Stream data to Cosmos DB**

In this module, you will stream simulated IoT device data into Cosmos DB to get an understanding of how to create a warm storage path in Azure. You can use the Microsoft Azure IoT Solution Accelerator that you set up in Module 1, Lesson 1. If you closed down the accelerator to save costs, you will need to set it up again.

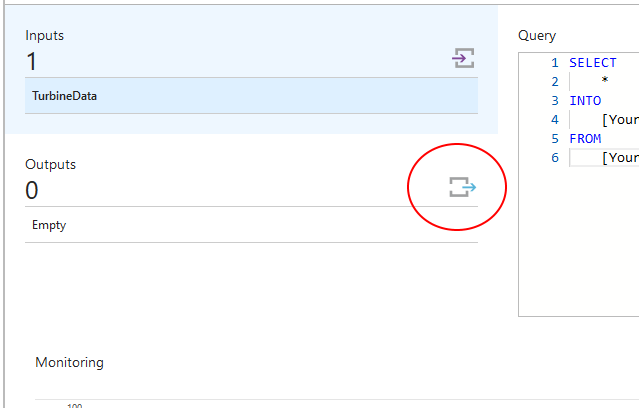
Device data comes in nearly every type of format, some of it obscure or outdated. Both the Azure IoT Hub and Azure Stream Analytics help transform and normalize the data before it is inserted into warm storage. This is a key difference between warm and cold storage. Cold storage is mainly intended for capturing and retaining data, so it’s better to leave data in its original format. Warm storage is intended to be used as transactional data, so some transformation is strategic. For your exercise, the IoT data will be in json format and you use Azure Stream Analytics' querying capabilities for data transformation.

1. If you deleted the device simulation that you created for Module 1, Lesson 1, go back and create a new one with the same parameters.

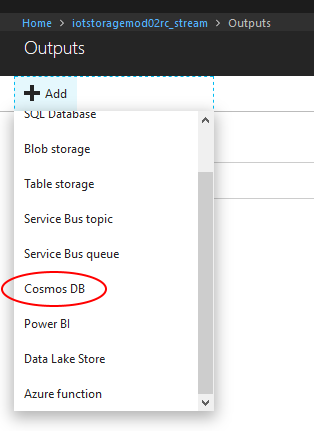
You may need to change the names of some of the Azure artifacts (the ones that require unique names). If you did not delete the device simulation, you can use it as-is.

You should now have an IoT Hub, as well as an Azure Stream Analytics job.

1. Navigate to the **Overview** page of your stream analytics job.
2. In the **Outputs** section, click the icon that looks like a box with an arrow emerging from it.



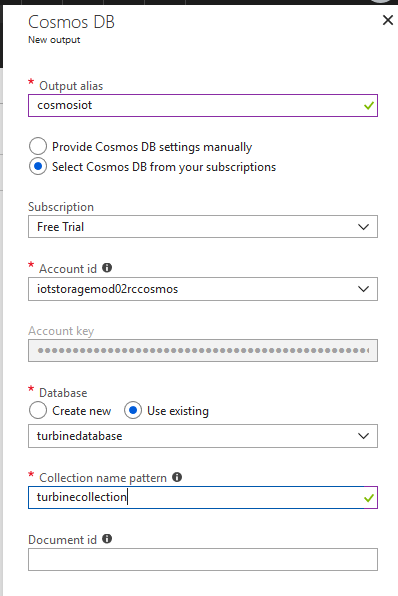
1. On the **Outputs** page, click **+ Add**
2. In the list of output choices, click **Cosmos DB**



1. Under **Output alias**, enter **cosmosiot**.
2. Under **Subscription**, make certain you have the correct **Subscription** selected
3. Under **Account id** select the default admin. That will automatically fill in the **Account key** with the correct value.
4. Under **Database**, click **Use existing**, then select **turbinedatabase**
5. Under **Collection name pattern**, type **turbinecollection**

Do not type anything under the optional **Document id**. As discussed in the last lesson, every json document stored in Cosmos DB requires a unique identifier. The data generated by the Azure IoT the Solution Accelerator does have some unique properties, for instance **IoTHub.ConnectionDeviceGenerationId**, but you won’t use that. Rather, you will leave the **Document id** box blank and allow Cosmos DB to create a unique identifier, as it inserts records.

1. Click **Save**



1. Navigate to the **Overview** page.
2. Click **Edit query**

If you are reusing your stream analytics job from module 1, add the SQL query below to the existing query. If you created your stream analytics job for this module, replace the placeholder query with the SQL query below.

SELECT

[bearings temperature],

[windings temperature],

[tower sway],

[position sensor],

[blade strain gauge],

[main shaft strain gauge],

[shroud accelerometer],

[gearbox fluid levels],

[power generation],

[EventProcessedUtcTime],

[EventEnqueuedUtcTime],

[IoTHub].[CorrelationId],

[IoTHub].[ConnectionDeviceId]

INTO

cosmosiot

FROM

TurbineData

Notice that you can use Stream Analytics query capabilities to change the shape of the source data. In the case of simulated data, we do not include any of the “unit” data, such as *tower sway\_unit*, because that is superfluous test data. Also notice that the source data has a json property that looks like this:

"IoTHub": {

"MessageId": null,

"CorrelationId": null,

"ConnectionDeviceId": "122f2c65-f8b9-4bb8-9314-fbc9fd1cf10e.9471b892-57c8-4260-9172-45533a79cc6f.9",

"ConnectionDeviceGenerationId": "636655659591924012",

"EnqueuedTime": "2018-06-26T12:10:00.9970000Z",

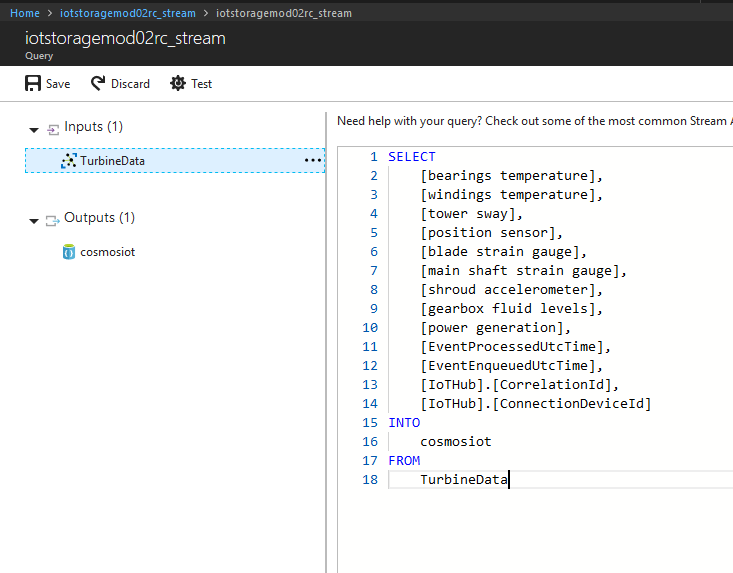
"StreamId": null

}

The data is “flattened” in the query by using dot notation:

[IoTHub].[CorrelationId],

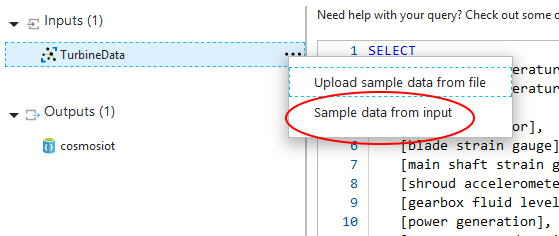
[IoTHub].[ConnectionDeviceId]



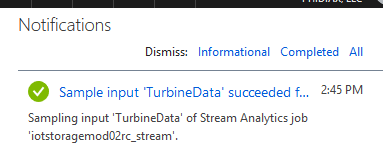
1. Get sample data.

The query page gives you the opportunity to sample data from your input source, so that you can test out your query.

1. Click the **ellipsis** next to your input and select **Sample data from input**.
2. Specify a time window that contains data.

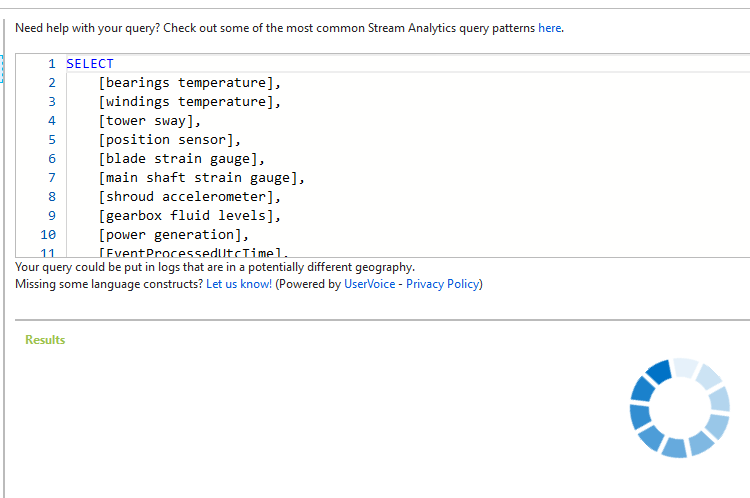


1. After a moment, you should get a notification that your sample input has succeeded. You now have data that you can run a test query against.

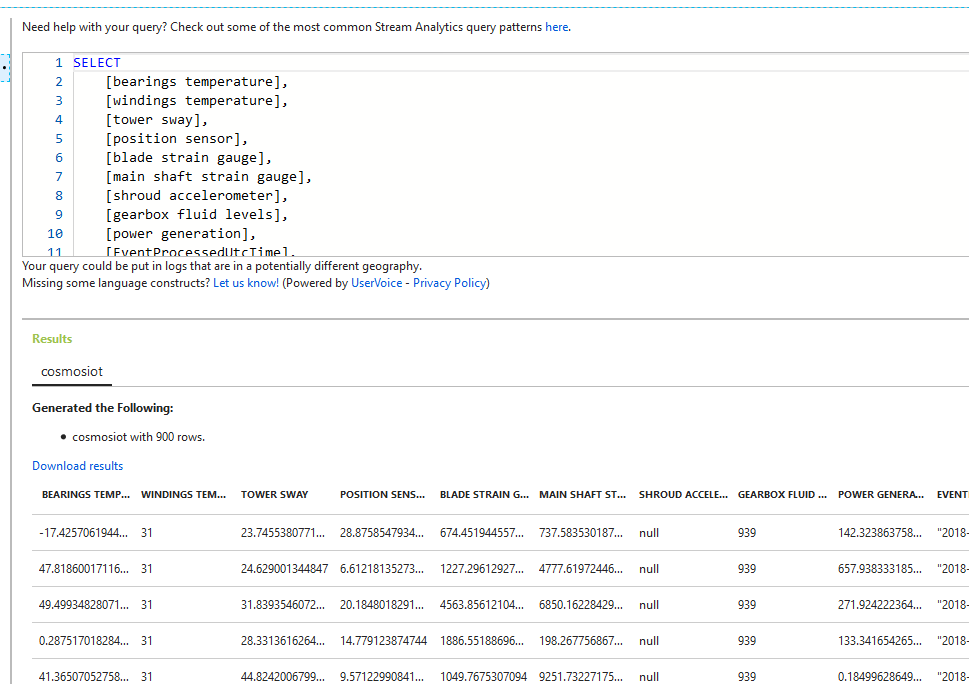


1. In the **Edit query** window, click **Test**.

If there are no query errors, you will see a “working” animation in the **Results** window. Once your query has finished, you will see the results as a tab-delimited list in the **Results** window. If that all looks good, you are ready to start your analytics job.



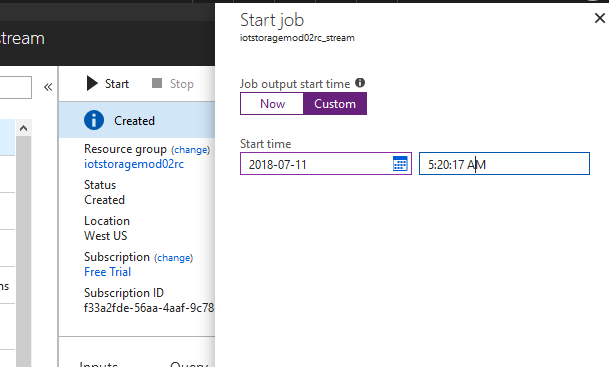
1. Click **Save**



1. Navigate back to the Azure Stream Analytics **Overview** page.
2. Click the **Start** icon.

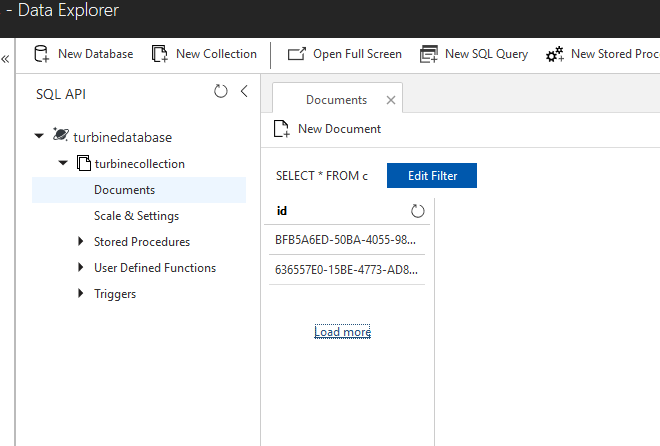
You will be asked for the **Job output start time**.

1. Click **Custom**
2. Set the **Start time** prior to when you started running the Device Simulation Solution Accelerator. Start the job.



1. Navigate to the Cosmos DB **Data Explorer** page.
2. Click **turbinecollection -> Documents** in the treeview.

You can use this to see the new records as they are inserted into the database.



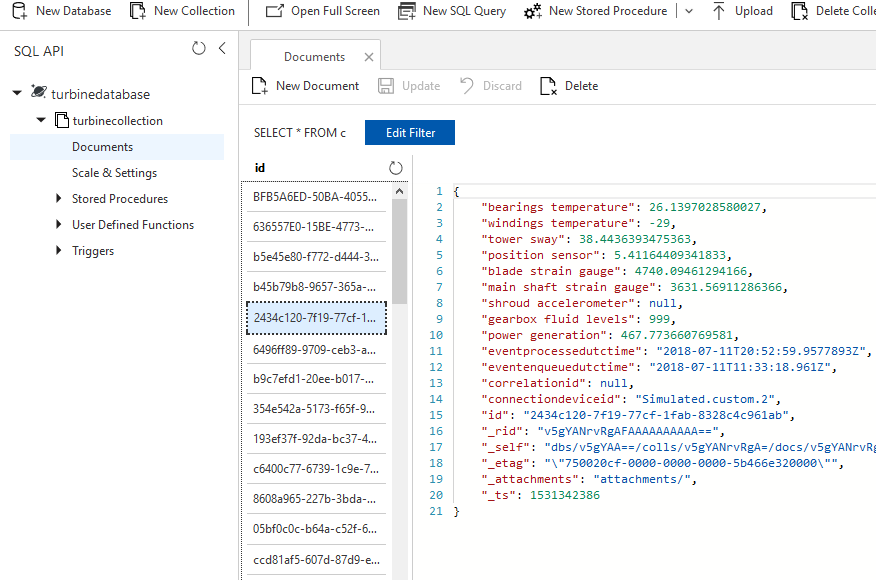
A couple of things to note: The analytics job can sometimes throw errors, because it is sending data faster than the provisioned throughput of Cosmos DB (or the streaming units for the stream analytics job). This would be an issue in production and would require careful sizing analysis and execution, but if your query works, it should not have an effect on the test scenario.

Also: the new database records may be slow to show up in the **Documents** window. When the new json documents are being inserted at a high velocity, the documents listing may be difficult to refresh or show odd results. Give it some time to allow the records to flow through.

1. Click one of the newly inserted records under **Documents** once things have settled down and you see new records.

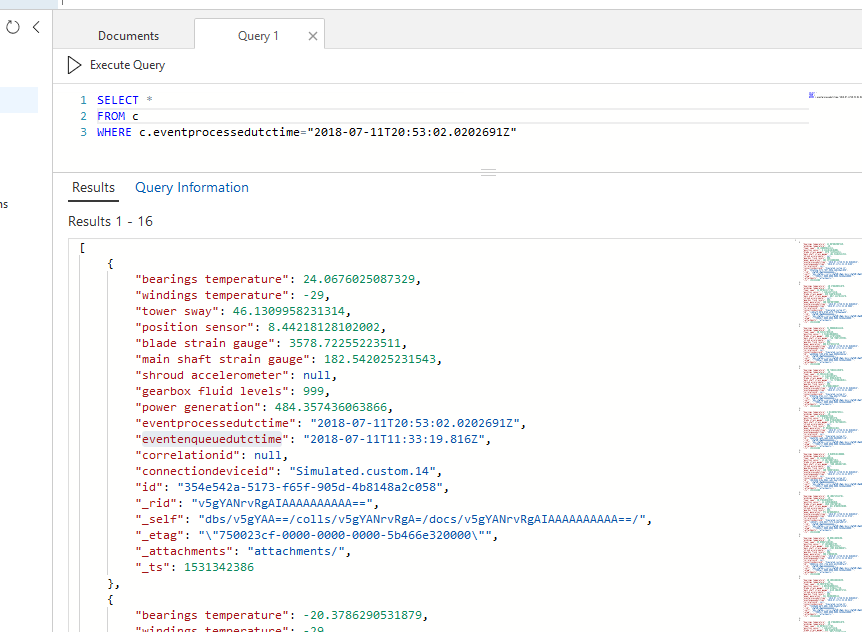
Notice that the data shape corresponds to your query. Notice also the properties that Cosmos DB adds for its own operation: \_rid, \_self, \_etag, etc.

In particular, notice that Cosmos DB has generated unique **id** properties for each document. As noted, if you do not supply a unique **id** any time you insert into Cosmos DB, Cosmos will create one.



1. Try out some queries. An easy way to do so is to copy a value from a document you find on the **Documents** pane.
2. Click **New SQL Query** at the top of the page.
3. Then use your copied value in a query. For instance:
4. SELECT \*
5. FROM c
6. WHERE c.eventprocessedutctime="2018-07-11T20:53:02.0202691Z"

(You will need to substitute a valid value from your data) Cosmos will index every property by default, so feel free to use a property of your liking.

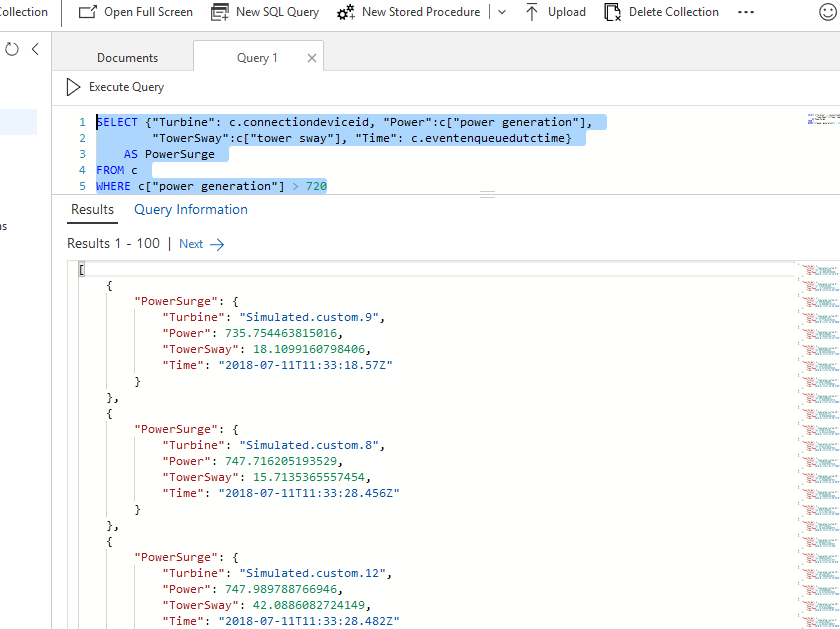


Next try a slightly more realistic query that you might use for warm data. Say that you want to know when a wind turbine has a power surge, so you could set up a query for unusually high power readings and have an application send a notification. (We will do something similar in a later lesson).

1. Write a query such as this:
2. SELECT {"Turbine": c.connectiondeviceid, "Power":c["power generation"],
3. "TowerSway":c["tower sway"], "Time": c.eventenqueuedutctime}
4. AS PowerSurge
5. FROM c
6. WHERE c["power generation"] > 200

You may need to adjust the power generation value down, if you get no results or adjust the value up, if you get too many results. Feel free to experiment.

Also, notice the shape of the results. In this scenario, we just want to find the correlation between power surges and tower sway, so we just include those values, along with the turbine identifier and the event’s time stamp.



**Summary**

In this lesson, you once again generated wind turbine IoT data. This time, you set up an analytics job to stream the data into Cosmos DB warm storage. You examined some of the particular characteristics of warm storage, including more sophisticated data queries and operational-oriented data sets. You also got some practice with Cosmos DB ad-hoc queries.

**Cosmos DB and Business Systems Integration**

**Cosmos DB and Business Systems Integration**

In this module, you will learn about

* Business system integration in an IoT architecture
* The unique characteristics of line of business applications
* How warm storage can interact with existing business systems

In this module, you are going to work with Azure Logic Apps, a representative technology for business integration. Cosmos DB has rich integration with Logic apps and can take actions based on Cosmos queries. You will create an app to send an e-mail when a telemetry reading goes above a certain threshold.

**Add a stored procedure to Cosmos DB**

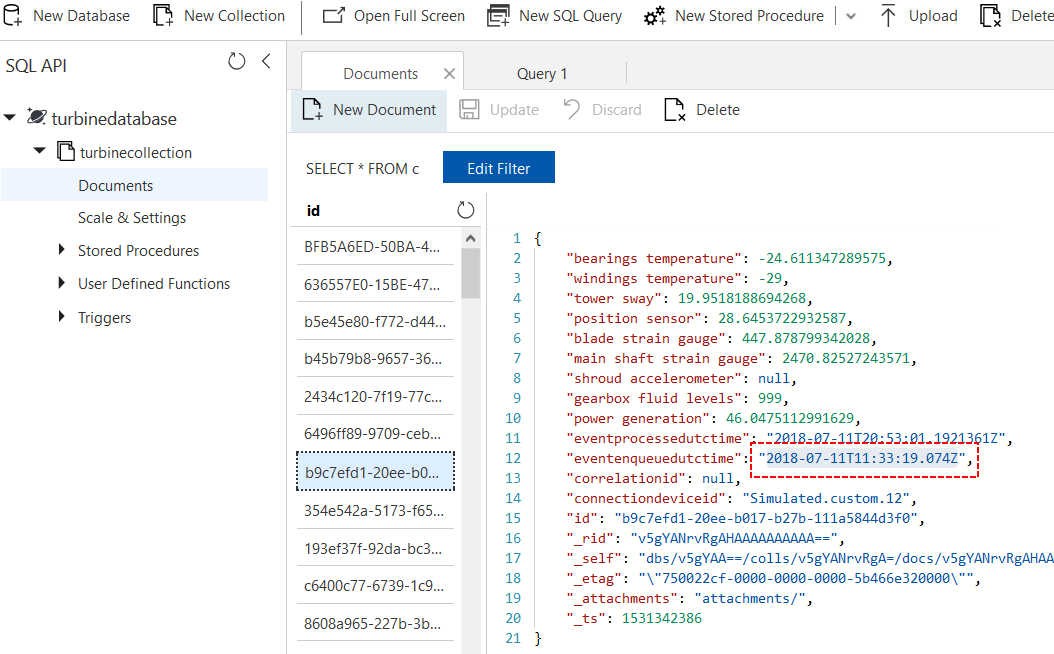
In this lesson, you will integrate the Cosmos DB collection that you’ve built with an Azure Logic App. This lesson is intended to demonstrate two main points: warm storage (in general) can be used for business system integration, and Cosmos DB (in particular) has deep integration with a number of technologies.

The overall goal of the lesson is to use an Azure Logic App to send a notification when a wind turbine’s main shaft is under a dangerous amount of strain. The first step in doing so is setting up a stored procedure for Cosmos DB that queries strain values of the wind turbine data that you have inserted into the database.

Stored procedures in Cosmos DB have a structure that is different than stored procedures in a relational database. They are built with JavaScript and are hosted within a single database partition. However, their structure allows functionality that is not common among other non-relational databases, such as transactional consistency. Let’s take a look.

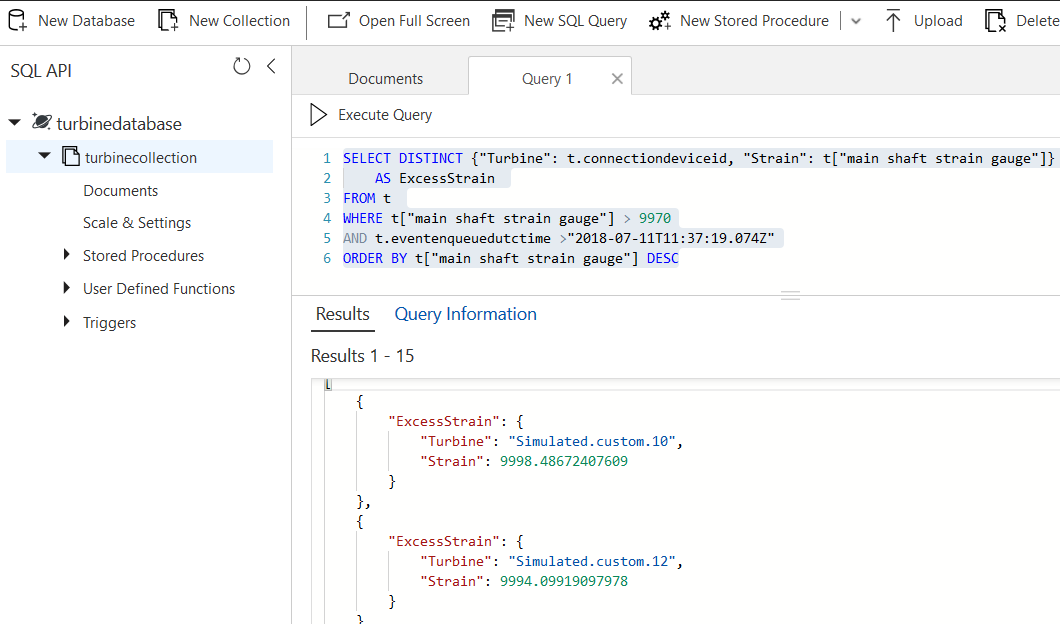
1. Navigate to the **Overview** pane for the Cosmos DB resource that you built in the last lesson.
2. Navigate to the **Data Explorer**
3. Expand **turbinecollection** in the SQL API treeview, then select **Documents**
4. Select one of the existing documents, to help get some representative values from your records.

Specifically, find some representative values for **EventEnqueuedUtcTime**. You are driving toward using a date comparison against wind turbine values. Find one of the relevant values, such as **“2018-07-11T11:37:19.074Z”**, and copy it.



1. Click **New SQL Query** at the top of the window. Try out the following query (substitute your date value):
2. SELECT DISTINCT {"Turbine": t.connectiondeviceid,
3. "Shaft Strain": t["main shaft strain gauge"],
4. "Blade Strain": t["blade strain gauge"]}
5. AS ExcessStrain
6. FROM t
7. WHERE t["main shaft strain gauge"] > 9970
8. AND t.eventenqueuedutctime > "2018-07-11T11:37:19.074Z"
9. ORDER BY t["main shaft strain gauge"] DESC

The data in your Cosmos DB is the product of simulated data, with random data falling within specified ranges. You will probably have to adjust the date value and the strain value (**9970**), to arrive at a small data set. It does not need to be exact, but try to find values that result in a data set of less than 25 or so. Again, we are trying to simulate a scenario where action would happen as wind turbines produced anomalous readings.



Familiarize yourself with the result set. It contains the turbine identifier, the shaft strain reading and the blade strain reading. A query like this might be of interest to a field technician who is keeping an eye on turbines with high strain values.

[

{

"ExcessStrain": {

"Turbine": "Simulated.custom.10",

"Shaft Strain": 9998.48672407609,

"Blade Strain": 2415.42200236135

}

}

]

After you have tested your query, you will create your Cosmos DB stored procedure.

1. Click **New Stored Procedure** at the top of the *Data Explorer* window.

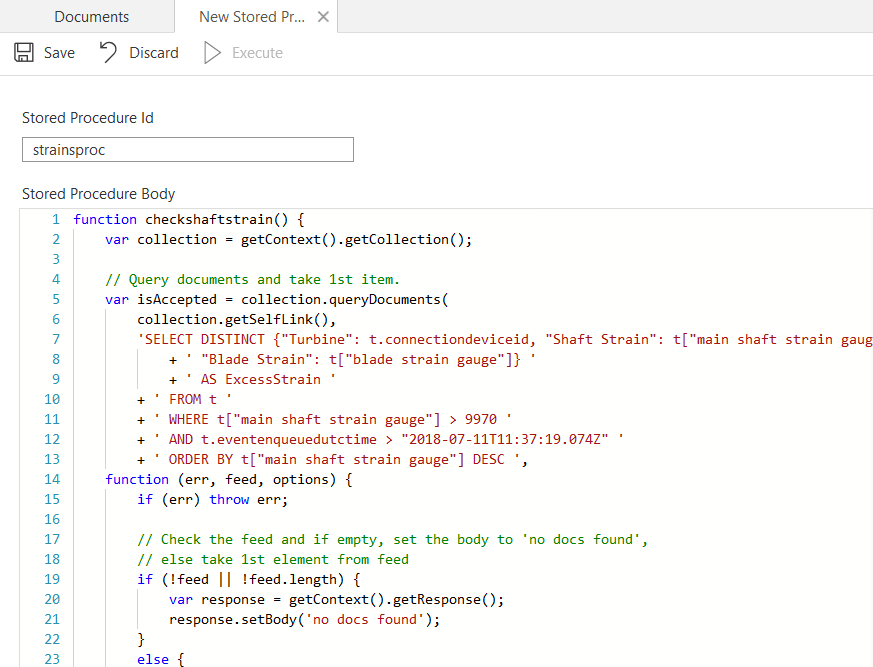
Take note of the place holder JavaScript that Cosmos fills in for the **Stored Procedure Body** box. We will use a slight variation on the default script, utilizing the query that you created in the previous task.

1. In the **Stored Procedure Id** box, name the procedure **strainsproc**
2. Replace the default JavaScript with this (make certain to change the query parameters to your test values):
3. function checkshaftstrain() {
4. var collection = getContext().getCollection();
5. // Query documents and take 1st item.
6. var isAccepted = collection.queryDocuments(
7. collection.getSelfLink(),
8. 'SELECT DISTINCT {"Turbine": t.connectiondeviceid, "Shaft Strain": t["main shaft strain gauge"], '
9. + ' "Blade Strain": t["blade strain gauge"]} '
10. + ' AS ExcessStrain '
11. + ' FROM t '
12. + ' WHERE t["main shaft strain gauge"] > 9970 '
13. + ' AND t.eventenqueuedutctime > "2018-07-11T11:37:19.074Z" '
14. + ' ORDER BY t["main shaft strain gauge"] DESC ',
15. function (err, feed, options) {
16. if (err) throw err;
17. // Check the feed and if empty, set the body to 'no docs found',
18. // else take 1st element from feed
19. if (!feed || !feed.length) {
20. var response = getContext().getResponse();
21. response.setBody('no docs found');
22. }
23. else {
24. var response = getContext().getResponse();
25. var body = { moststrain: feed[0] };
26. response.setBody(JSON.stringify(body));
27. }
28. });
29. if (!isAccepted) throw new Error('The query was not accepted by the server.');
30. }
31. Look at some of the components of the JavaScript function:

We named the function something descriptive – **checkshaftstrain**

The variable *collection* is initialized with the context of the Cosmos DB collection that is “hosting” it.

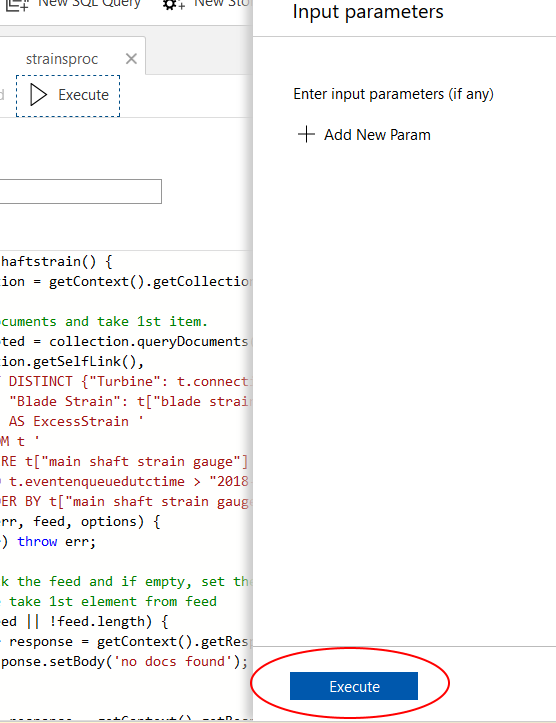
The main work for querying data is performed by the Cosmos DB JavaScript function *queryDocuments*. It takes as parameters the string representation of the SQL query, and it sets up a function that will return a string representation of the first document result of the query, or return *no docs found*, if there are no results. It also throws an error if something goes amiss



1. Click **Save** This will save the **strainsproc** stored procedure. The **Execute** button should now be active.
2. Click **Execute**

You will get a popup asking you for input parameters. You don’t need to add any.

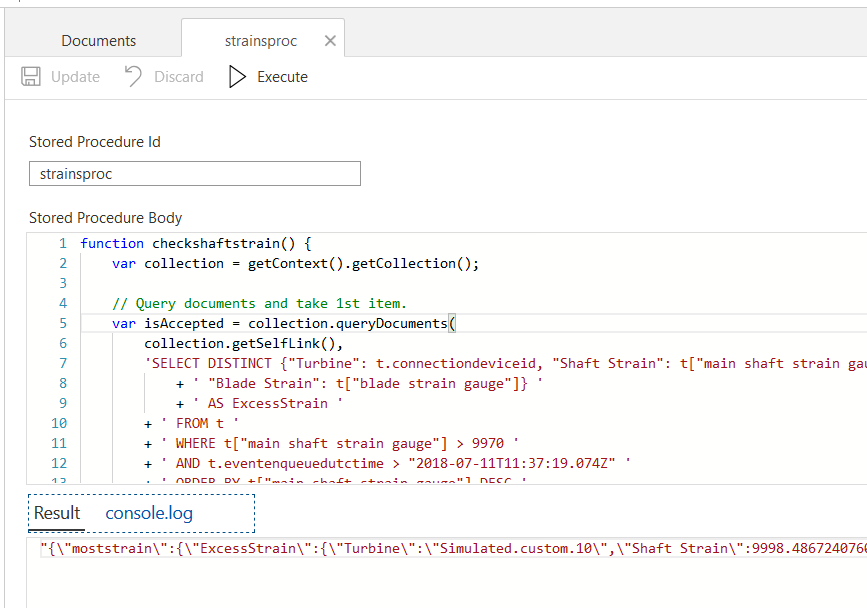
1. Click the second **Execute** button.



1. Look at the results of the stored procedure in the *Result* window.

There will be an escaped string version of one json document.

"{\"moststrain\":{\"ExcessStrain\":{\"Turbine\":\"Simulated.custom.10\",\"Shaft Strain\":9998.48672407609,\"Blade Strain\":2415.42200236135}}}"



The unescaped version looks like this. It will be the record with the highest value for Shaft Strain, since the SQL query uses Shaft Strain as the sort value (descending) and the stored procedure takes the first record.

{

"moststrain": {

"ExcessStrain": {

"Turbine": "Simulated.custom.10",

"Shaft Strain": 9998.48672407609,

"Blade Strain": 2415.42200236135

}

}

}

Your stored procedure is now ready for integration with an Azure Logic App.

**Summary**

In this lesson, you built up a SQL query that is representative of operational data you would extract from warm storage. You then used the SQL query to create a Cosmos DB stored procedure that can be integrated with a Logic App.

**Create a logic app to integrate with Cosmos DB warm storage**

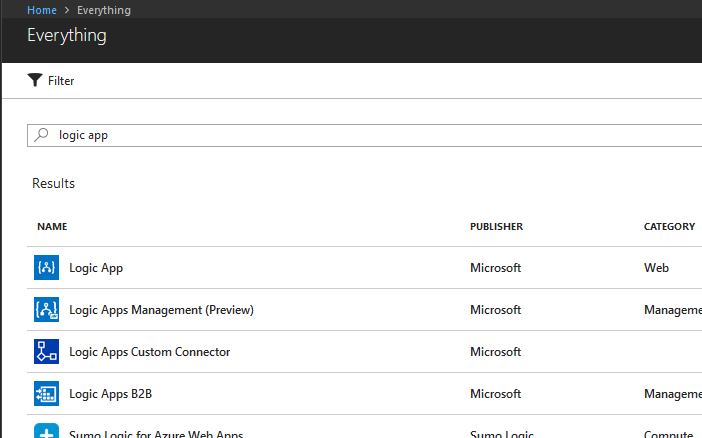
In this lesson, you will build an Azure Logic App that leverages the stored procedure you created in the last task. If the logic app gets a positive result from its query, it will send an e-mail message.

Azure Logic Apps are one of several serverless offerings that Azure provides. With Logic Apps, there is no need to provision infrastructure. Instead, you build workflow-driven applications. The technology is lightweight, but massively scalable. It can be used for application integration, data integration, system integration, enterprise application integration (EAI), and business-to-business (B2B) communication, whether in the cloud, on premises, or both.

For the purposes of this lesson, the logic app will perform an operational task — sending an e-mail alert based on telemetry data from the warm storage database. Logic apps are one of Microsoft’s recommended technologies for Business Process integration. Keep in mind though, when you use Cosmos DB as warm storage, you have a staggering number of choices you can use for your client applications. Cosmos DB has native clients that you can use with .NET, Python, Node.js, JavaScript, and Java. Many of the Azure technologies integrate with Cosmos DB natively: Azure Functions, Flow, Batch, and more.

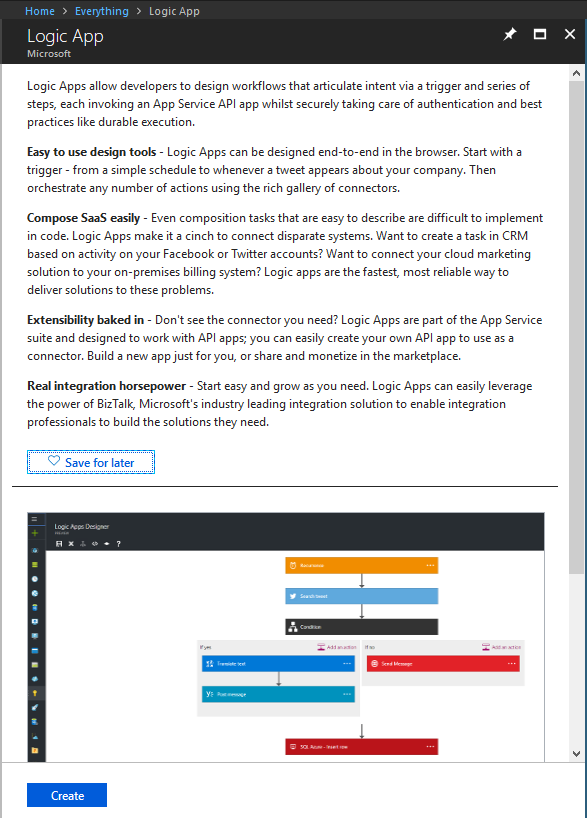
An IoT architecture built on Azure can give you a lot of flexibility for integrating with other technologies. If a certain application technology is used in your company, chances are you can integrate it.

1. On your Azure portal, click **Create a resource**
2. Search for **Logic App**
3. Select the Microsoft resource

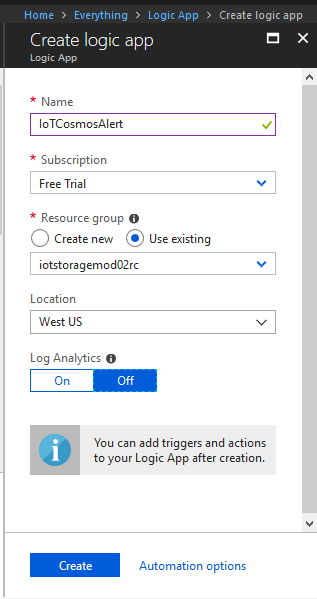


Logic apps have tooling that you can use with Visual Studio.NET or Visual Studio Code. The Azure Portal also hosts a visual designer for creating Logic App workflows. In a production environment, you would most likely author Logic Apps with offline tools and publish them to Azure. That allows deeper integration with existing source control and development processes. For the purpose of this exercise, you will create your Logic App on the Azure portal. Its visual designer can do everything that the offline tools do; in fact, new features are generally released on the Azure portal before they have support in offline tooling.

1. Click **Create** on the Logic App page.

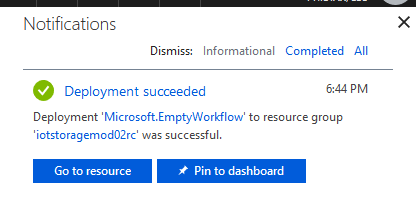


1. Under **Name**, enter **IoTCosmosAlert**
2. Select the appropriate Azure **Subscription** from the menu.
3. Use the same **Resource group** that you have been using for previous exercises.
4. Select a region near you for **Location**
5. Click **Create**



It may take a moment to provision your logic app. Pay attention to your notifications.

1. Navigate to the *Logic App's* resource page after it finishes.



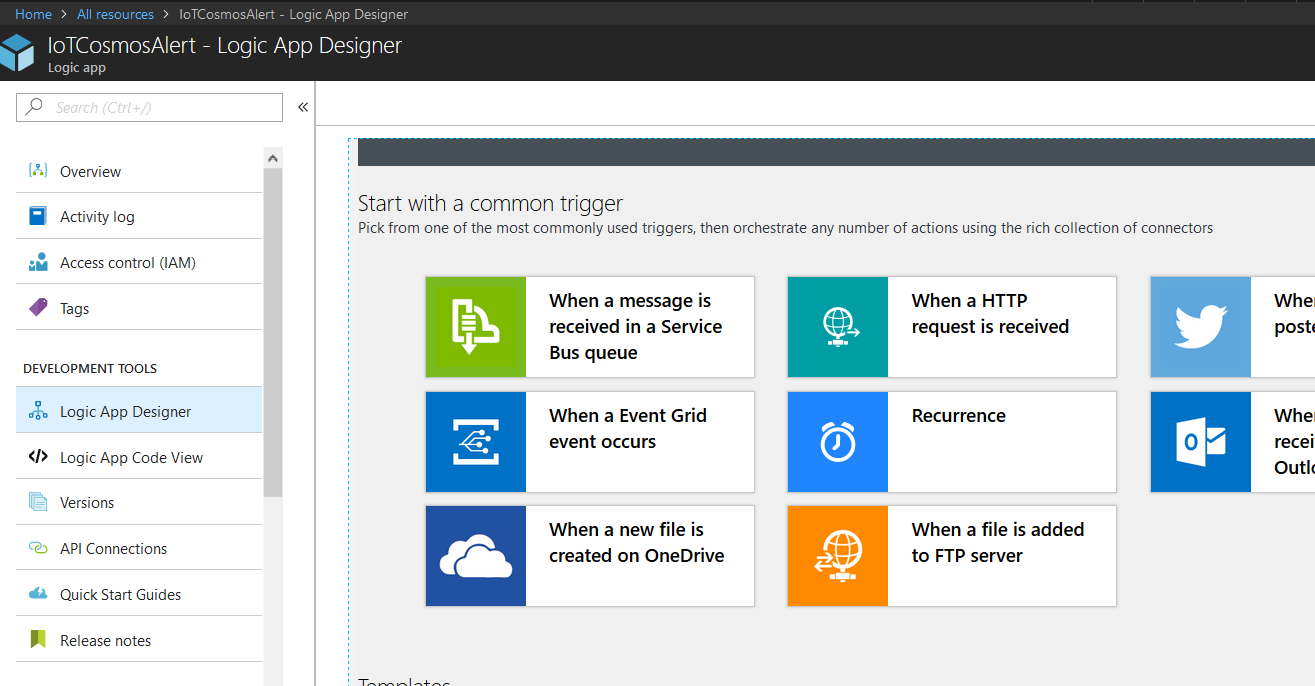
The first step for creating a Logic App is defining a trigger. Every logic app workflow starts with a trigger, which fires when a specific event happens, or when new available data meets specific criteria. Many triggers include basic scheduling capabilities so that you can specify how regularly your workloads run.

Each time that the trigger fires, the *Logic App* engine creates an instance that runs the workflow's actions. These actions can also include data conversions and flow controls, such as conditional statements, switch statements, loops, and branching.

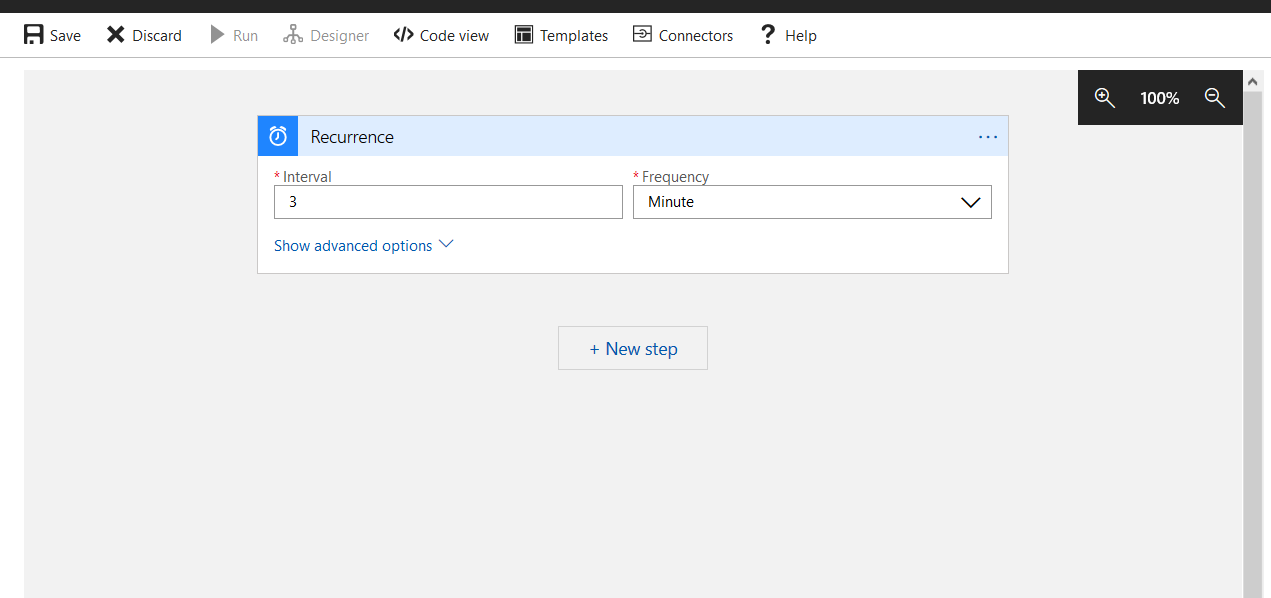
On the **Logic App Designer** page, you will see several choices for different types of triggers.

1. Choose **Recurrence**

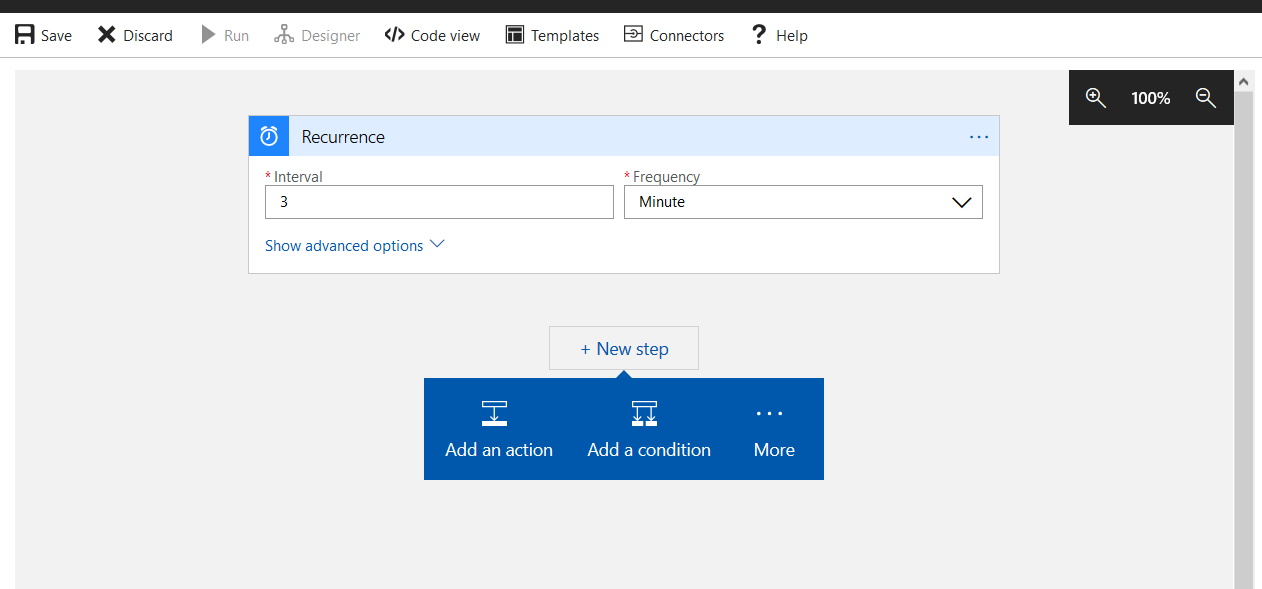
This will create a trigger that fires on set time intervals.



You will automatically navigate to the main workflow designer page, and it will contain a workflow shape for a **Recurrence** trigger. The default frequency is 3 minutes. You can leave the default as-is for this lesson.



1. Click **+ New step** on the designer surface.
2. Choose **Add an Action** from the resulting pop-up menu.



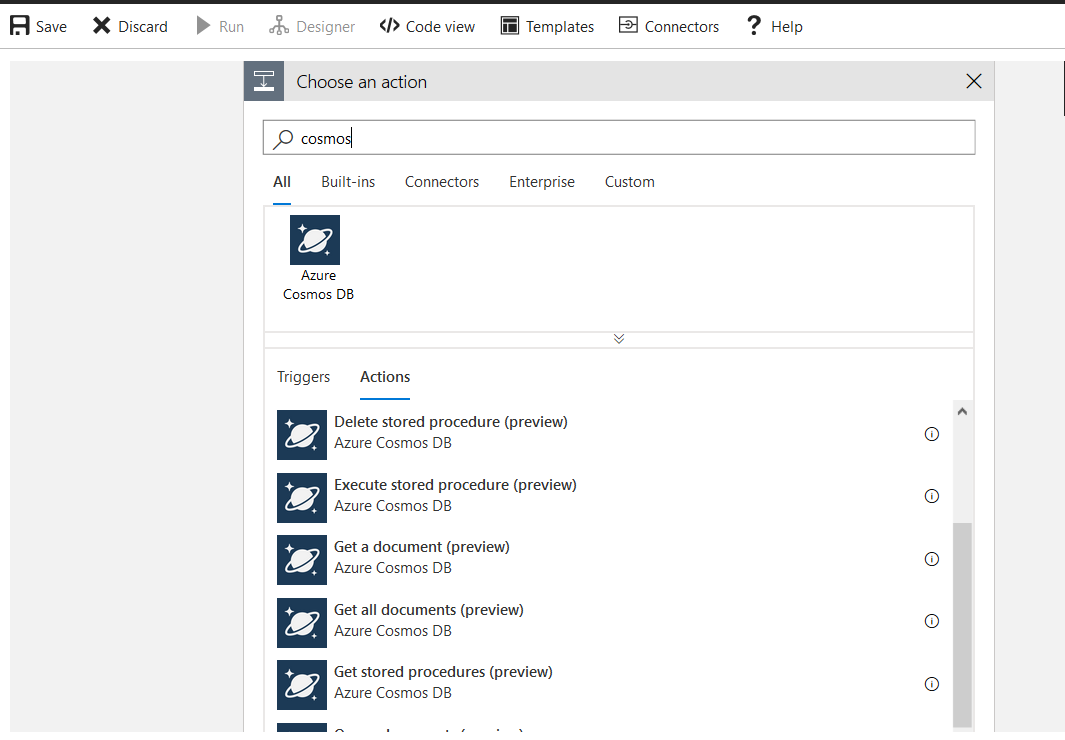
A generic **Choose an action** workflow shape will appear. Notice that there are hundreds of choices, representing all manner of actions and integrations with other services and applications.

1. Under **Choose an action**, enter **Cosmos** into the search box at the top of the workflow shape.

Several Cosmos DB-related actions will appear.

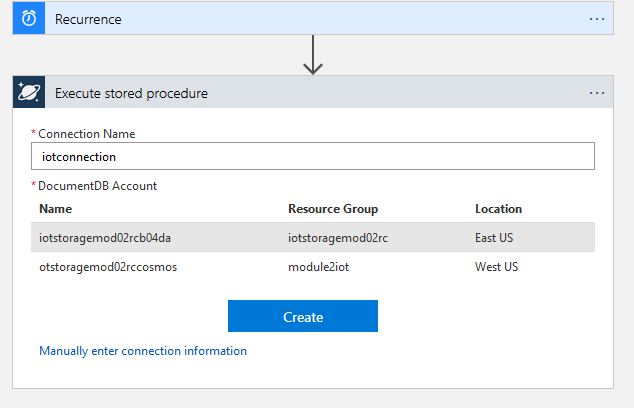
1. Select **Execute Stored Procedure**.

You will be presented with a list of parameters to fill in.



1. If prompted, under **Connection Name** enter **iotconnection**.
2. Select your Cosmos DB account.

**Note**: You may not need to do this step, if you only have one Cosmos DB account.

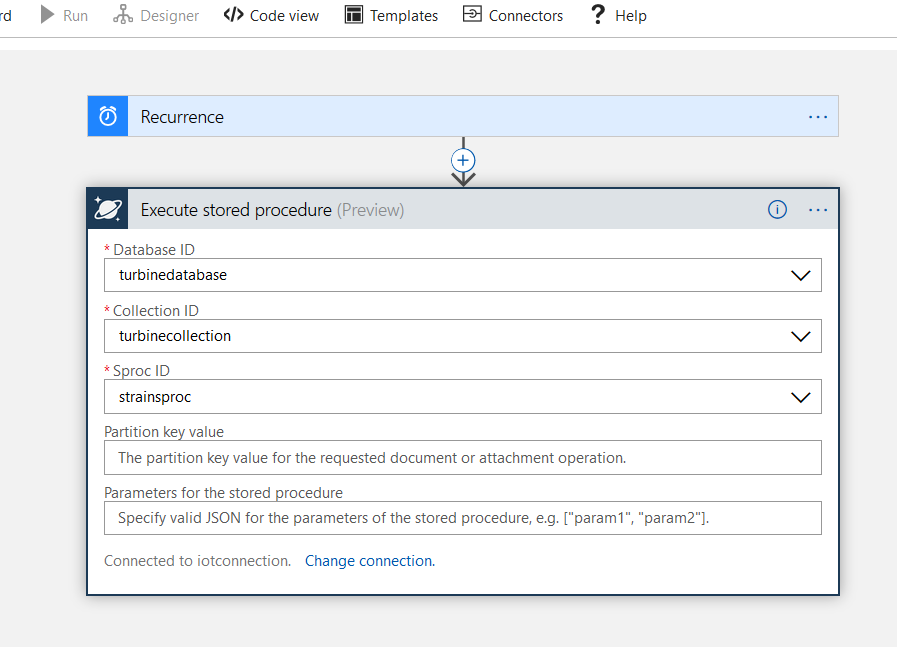


1. Under **Database ID**, select **turbinedatabase**
2. Under **Collection ID**, select **turbinecollection**
3. Under **Sproc ID**, select **strainsproc**.
4. Leave the **Partition key value** box blank.

As mentioned before, the scope of a Cosmos DB stored procedure is a partition key. Since we created a fixed size Cosmos DB instance without a specific partition key, the stored procedure will work against all the documents in the **turbinecollection**.

1. Also leave the **parameters** box blank.

Most production-grade stored procedures would have input parameters of some kind, but we created **strainsproc** without parameters.



1. Select **Save** at the top of the window.

This will check to see if everything is working thus far.

1. Select **Run**

You will see some status messages as the logic app prepares to execute. Then you should see green check marks next to each workflow step, when the logic app completes.

1. Select the **Execute stored procedure** workflow shape.

This will display its inputs and outputs. Pay special attention to the **Outputs** section. Notice that in the **OUTPUTS** section of this workflow shape, it has a property named *Body* that captures the results of the **strainsproc** stored procedure.

{

"moststrain": {

"ExcessStrain": {

"Turbine": "Simulated.custom.10",

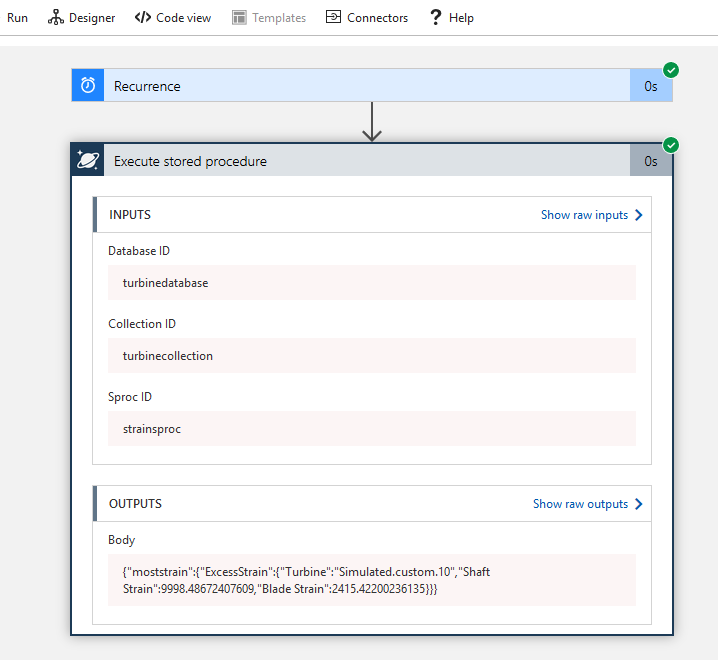
"Shaft Strain": 9998.48672407609,

"Blade Strain": 2415.42200236135

}

}

}



1. Navigate back to the **Logic App Designer** pane.
2. Select **+ New Step** again.

This time choose **Add a condition**. You are going to add a workflow step that will execute logic based on the results of the **strainsproc** stored procedure.

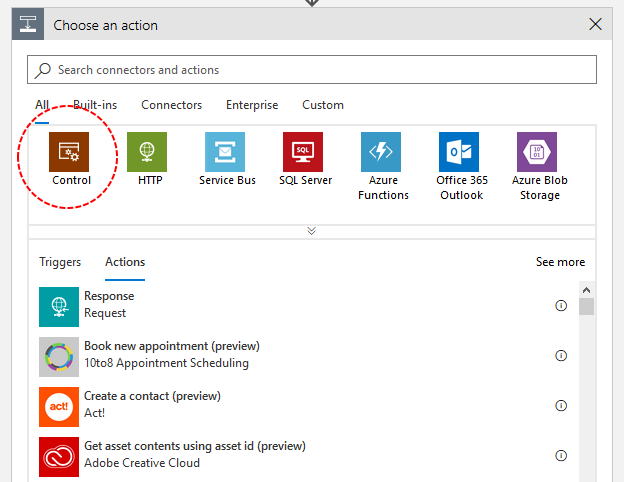
1. Select the **if-then** workflow shape.

This shape will choose one of two paths, based on the condition that you set.

**ALTERNATIVELY**: If you do not get the option to select **Add a condition** on your workflow designer.

1. Under **Choose an action** click **Control** from the top row of **connectors and actions**.

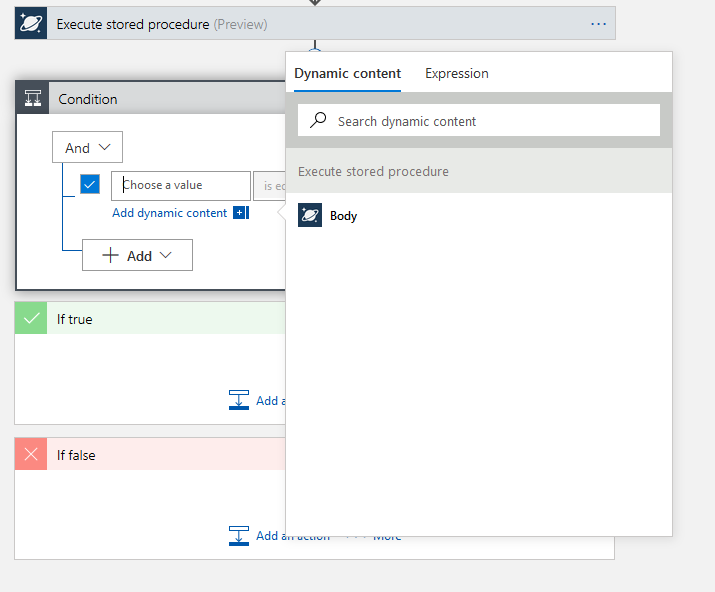
You will then be able to select a **Condition** control. (Disregard this step, if #27 and #28 work for you).



1. Under the **Condition** shape, select **Choose a value**.
2. Select **Add dynamic content**.

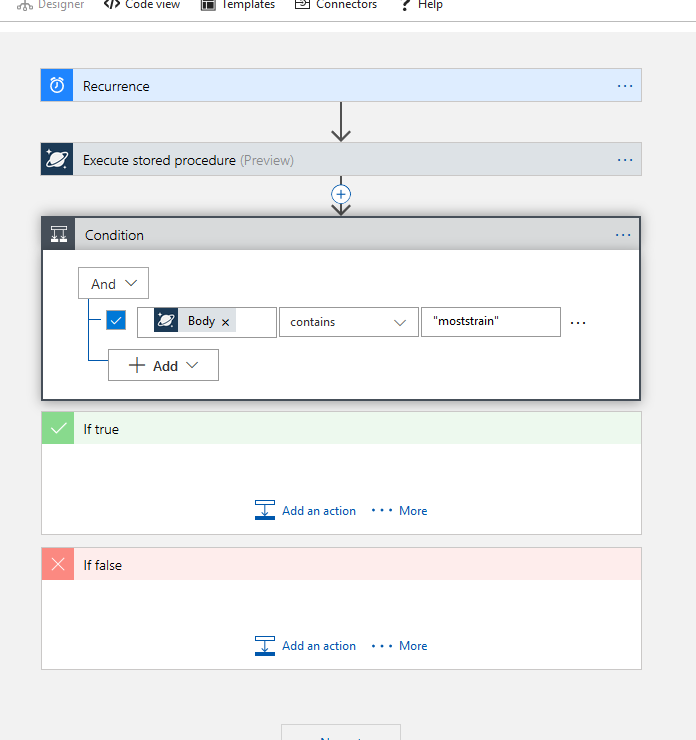
Notice the pop-up screen that has an icon named *Body*. This represents the results of the stored procedure.

1. Select the icon to populate the **value** box.



1. Under the list next to the value box, select **contains** .
2. In the final box on the right, enter **“moststrain”** .

This will set up a condition so that if a record comes back from the stored procedure, the *If true* condition will fire. Recall that if the Cosmos DB query returns no records, the stored procedure will return a string value *“no docs found”*. In which case, the body will not contain the string “moststrain” and the *If false* condition will fire.

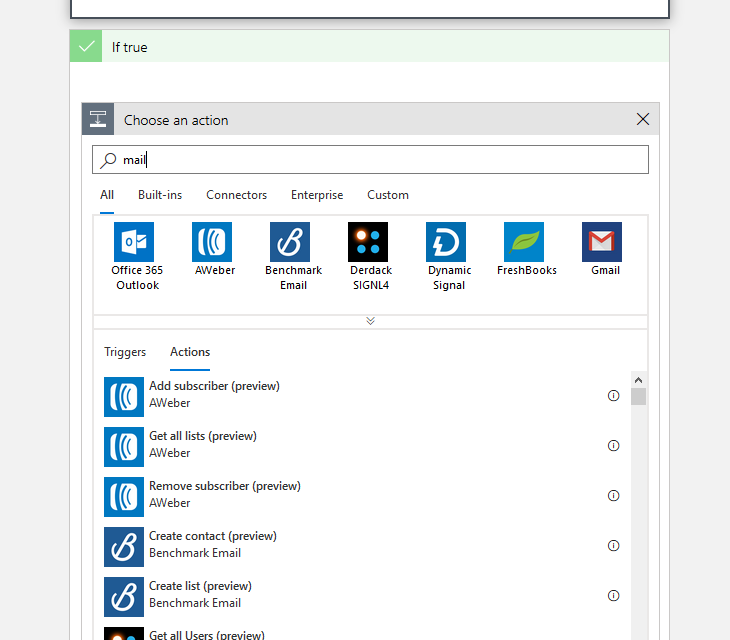


1. In the *Condition* workflow shape, go to the *If true* sub-shape and select **Add an action**
2. Under the generic *Choose an action* workflow shape, search for **mail**.

There will be several choices, and some of them depend on the type of account you are using. The graphic below shows what you might see if you are using Azure in conjunction with an Office 365 account. You may see a different link that suggests Outlook mail. Or you may need to sign in to a generic mail provider.

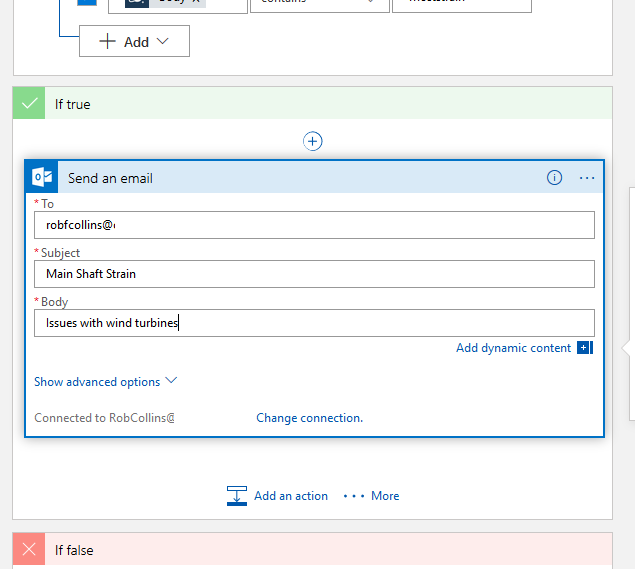
Regardless, your goal is to find an action named **Send an e-mail**, click on it, and go through the process of e-mail login.

(It may not ask, if your account already has e-mail access.



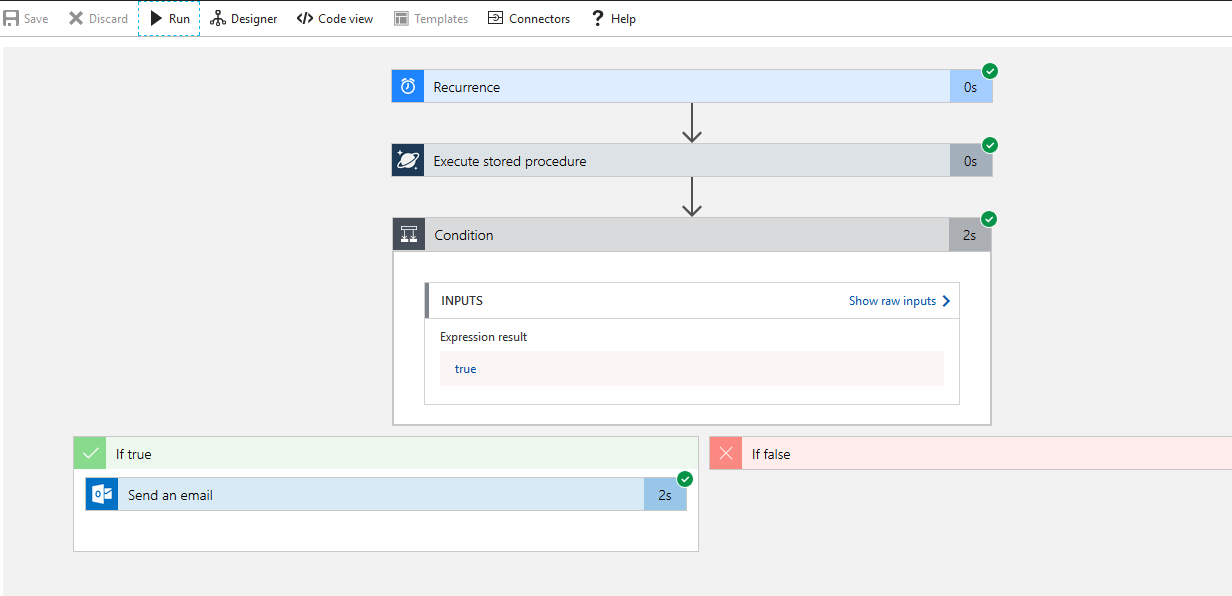
1. Once validated, fill out the contents of the **Send an email** shape.
2. In the **To** box, write an e-mail that you control as the e-mail recipient.
3. Add a **Subject** such as **Main Shaft Strain**
4. Add an e-mail **Body** such as **Issues with wind turbines**.

As you can tell from the **Add dynamic content** link, you can populate the body with something like the results of stored procedure. We’ll just keep it simple for this exercise.



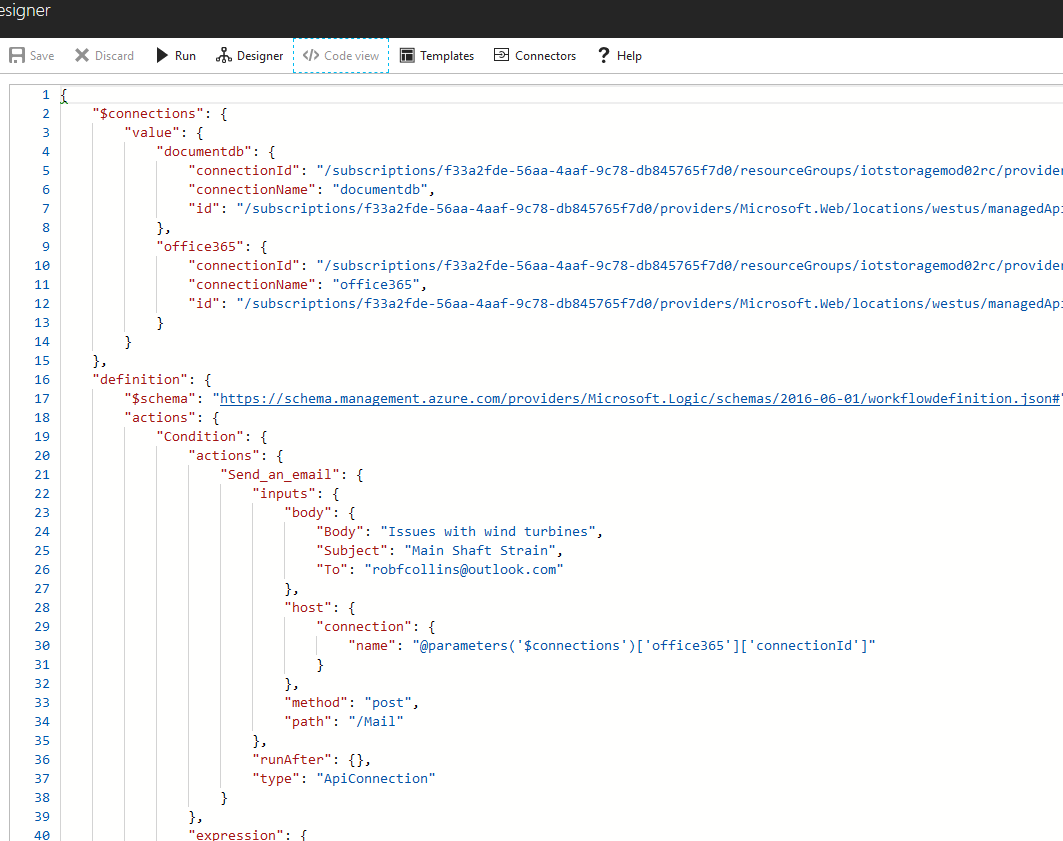
1. Select **Save**.
2. Select **Run** at the top of the window.

You should shortly see green check marks next to each step in the workflow, indicating that the message has been sent.

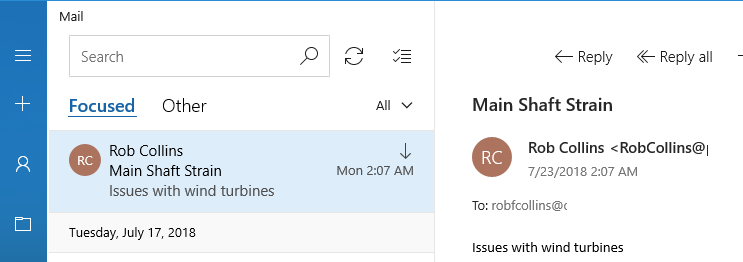


1. At the top of the workflow designer window, select **Code view**.

Treat this action as one final thing before you find the sent e-mail. Notice that all the actions, conditions and parameters that you set with the designer show up in this view as JSON properties and values. This code represents the building blocks that Azure uses for Logic Apps.



Provided your e-mail addresses are validated and correct, you should receive the mail you set up.



By setting up and saving this logic app, you automatically enabled it. When you are done with this lesson, be certain to disable it, otherwise you will continue to get the same e-mail every three minutes until you do.

1. Navigate to the Logic App's Overview blade
2. Select **Disable**

**Summary**

In this lesson, you created an Azure Logic App. You assigned a trigger to make it run on a set schedule, then you added a workflow activity to make it execute a Cosmos DB stored procedure and capture the results. You added a conditional workflow step that compared the results of the stored procedure results and sent an e-mail, if results came through. These steps illustrated an example of using warm storage IoT data for business integration.