**Create a function app**

Let's create a function app that we'll use throughout this entire module. A function app lets you group functions as a logical unit for easier management, deployment, and sharing of resources.

1. Sign into the [Azure portal](https://portal.azure.com/learn.docs.microsoft.com) using the same account you activated the sandbox with.
2. Select the **Create a resource** button found on the upper left-hand corner of the Azure portal, then select **Compute** > **Function App**.
3. Set the function app properties as follows:

| **Property** | **Suggested value** | **Description** |
| --- | --- | --- |
| **App name** | Globally unique name | Name that identifies your new function app. Valid characters are a-z, 0-9, and -. |
| **Subscription** | Your subscription | The subscription under which this new function app is created. |
| **Resource Group** | Select **Use existing** and choose *7cba6e7c-96f5-4e78-868d-ec01d077bb2e* | Name of the resource group in which to create your function app. |
| **OS** | Windows | The operating system that hosts the function app. |
| **Hosting Plan** | Consumption plan | Hosting plan that defines how resources are allocated to your function app. In the default **Consumption Plan**, resources are added dynamically as required by your functions. In this serverless hosting model, you only pay for the time your functions run. |
| **Location** | Select from the list | Choose the nearest one to you that is also one of the allowed *Sandbox regions* listed below. |
| **Runtime Stack** | JavaScript | The sample code in this module is written in JavaScript. |
| **Storage** | Globally unique name | Name of the new storage account used by your function app. Storage account names must be between 3 and 24 characters in length and may contain numbers and lowercase letters only. This dialog populates the field with a unique name that is derived from the name you gave the app. However, feel free to use a different name or even an existing account. |

**Sandbox regions**

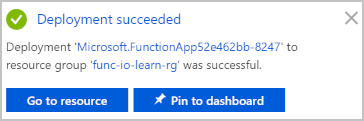
The free sandbox allows you to create resources in a subset of Azure's global regions. Select a region from the following list when creating any resources:

* + West US 2
  + South Central US
  + Central US
  + East US
  + West Europe
  + Southeast Asia
  + Japan East
  + Brazil South
  + Australia Southeast
  + Central India

1. Select **Create** to provision and deploy the function app.
2. Select the Notification icon in the upper-right corner of the portal and watch for a **Deployment in progress** message similar to the following message.



1. Deployment can take some time. So, stay in the notification hub and watch for a **Deployment succeeded** message similar to the following message.



1. Once the function app is deployed, go to **All resources** in the portal. The function app will be listed with type **App Service** and has the name you gave it. Select the function app from the list to open it.

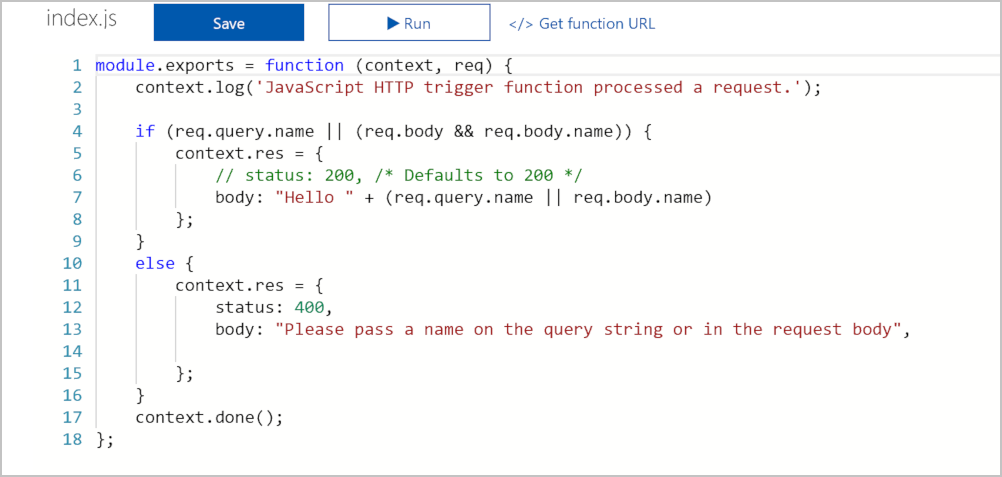
**Create a function**

Now that we have a function app, it's time to create a function. A function is activated through a trigger. In this module, we'll use an HTTP trigger.

1. Select the Add (**+**) button next to **Functions**. This action starts the function creation process.
2. On the **Azure Functions for JavaScript - getting started** page, select **In-portal** and then select **continue**.
3. In the **Create a function** step, select **More templates...** and then select **Finish and view templates**.
4. In the list of all templates available to this function app, select **HTTP Trigger** .
5. On the **New Function** screen, change the name if you want, leave the **Authorization level** as *Function*, and click **Create**.
6. In your new function, click the **</> Get function URL** link at the top right, select **default (Function key)**, and then select **Copy**.
7. Paste the function URL you copied into the address bar of a new tab in your browser.
8. Add the query string value &name=Azure to the end of this URL, and then press Enter on your keyboard to execute the request. You should see a response similar to the following response returned by the function displayed in your browser.

<string xmlns="http://schemas.microsoft.com/2003/10/Serialization/">Hello Azure</string>

As you can see from this exercise so far, you have to select a trigger type when you create a function. Every function has one and only one trigger. In this example, we're using an HTTP trigger, which means that our function starts when it receives an HTTP request. The default implementation, shown in the following screenshot in JavaScript, responds with the value of the parameter *name* it received in the query string or body of the request. If no string was provided, the function responds with a message that asks whomever is calling to supply a name value.



All of this code is in the **index.js** file in this function's folder. Let's look briefly at the function's other file, the **function.json**config file. This configuration data is shown in the following JSON listing.

{

"bindings": [

{

"authLevel": "function",

"type": "httpTrigger",

"direction": "in",

"name": "req",

"methods": [

"get",

"post"

]

},

{

"type": "http",

"direction": "out",

"name": "res"

}

],

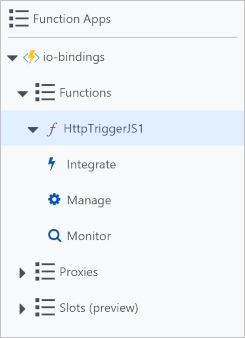
"disabled": false

}

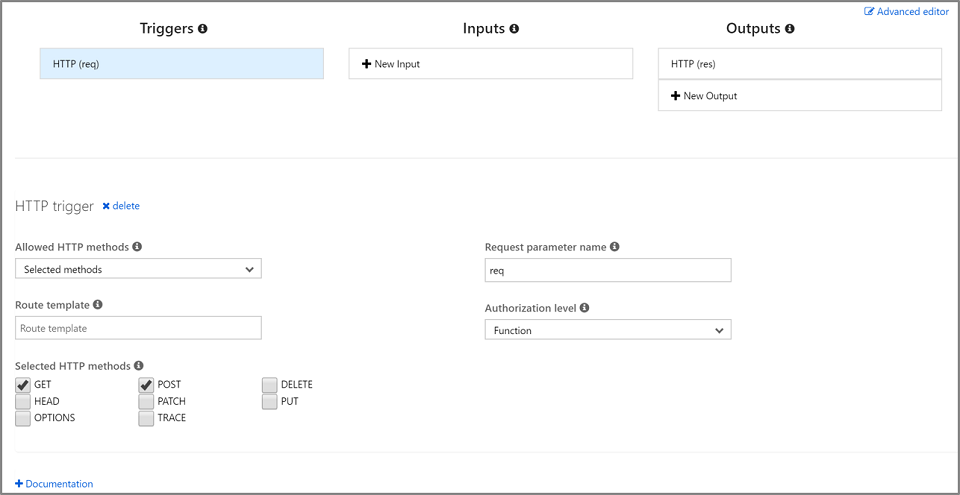
As you can see, this function has a trigger binding named **req** of type httpTrigger and an output binding named **res** of type HTTP. In the preceding code for our function, we saw how we accessed the payload of the incoming HTTP request through our **req** parameter. Similarly, we sent an HTTP response simply by setting our **res** parameter. Bindings really do take care of some of the heavy lifting for us.

### Explore binding types

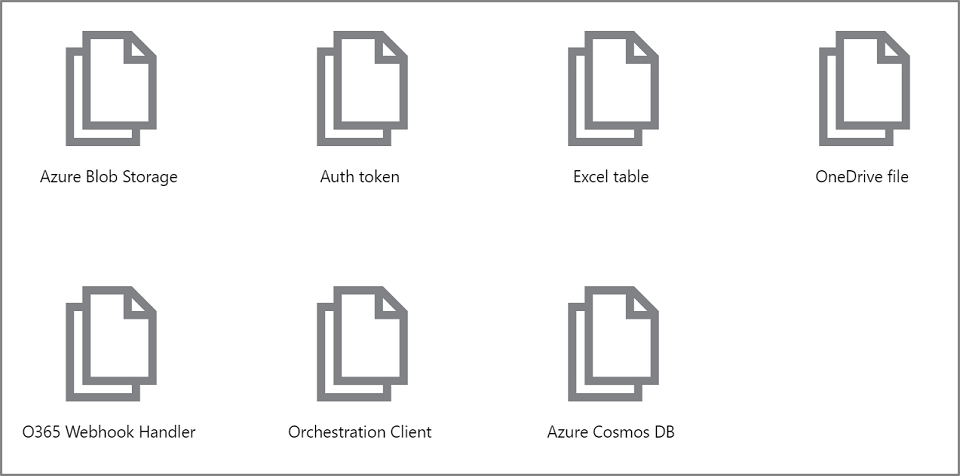
1. Notice under the function entry there is a set of menu items as shown in the following screenshot.



1. Select the **Integrate** menu item to open the integration tab for our function. If you've been following along with this unit, the integrate tab should look very similar to the following screenshot.

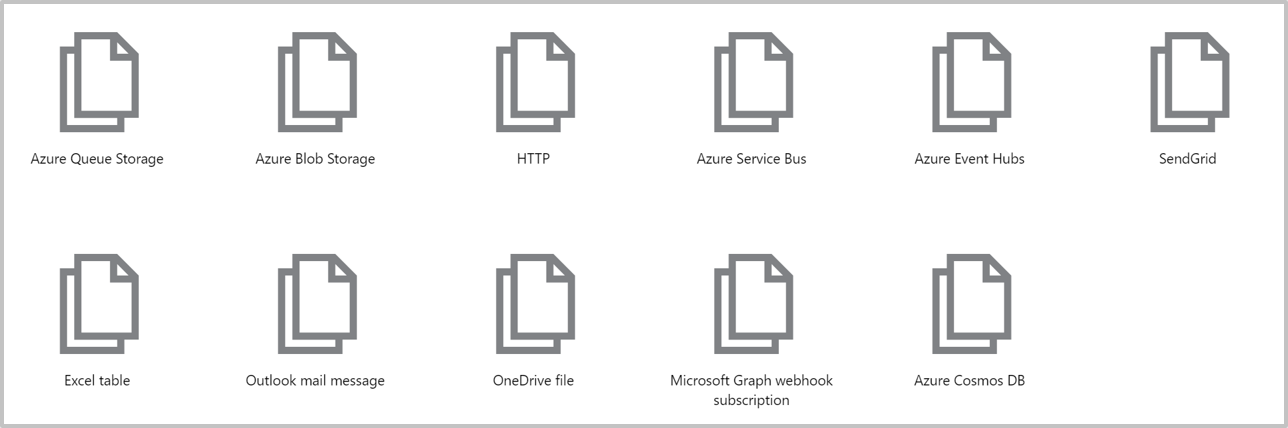


1. Select **+ New Input** under the **Inputs** column. A list of all possible input binding types is displayed as shown in the following screenshot.



Take a moment to consider each of these input bindings and how you might use them in a solution. There are a lot to choose from. This list might even have changed by the time you read this module, as we continue to support more data sources.

1. We'll get back to adding input bindings later in the module but, for now, select **Cancel** to dismiss this list.
2. Select **+ New Output** under the **Outputs** column. A list of all possible output binding types is displayed as shown in the following screenshot.\



As you can see, there are several output binding types at your disposal. We'll get back to adding output bindings later in the module but, for now, select **Cancel** to dismiss this list.

So far, we've learned how to create a function app and add a function to it. We've seen a simple function in action, one that runs when an HTTP request is made to it. We've also explored the Azure portal UI and types of input and output binding that are available to our functions. In the next unit, we'll use an input binding to read text from a database.

## Create an Azure Cosmos DB account

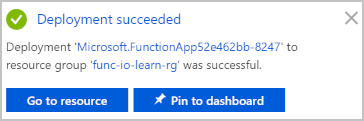
**Create a database account**

A database account is a container for managing one or more databases. Before we can create a database, we need to create a database account.

1. Make sure you are signed into the [Azure portal](https://portal.azure.com/learn.docs.microsoft.com) using the same account you activated the sandbox with.
2. Select the **Create a resource** button found on the upper left-hand corner of the Azure portal, then select **Databases**> **Azure Cosmos DB**.
3. In the **Create Azure Cosmos DB Account** page, enter the settings for the new Azure Cosmos DB account.

| **Setting** | **Value** | **Description** |
| --- | --- | --- |
| Subscription | Concierge subscription | The Azure subscription that you want to use for this Azure Cosmos DB account. |
| Resource Group | 7cba6e7c-96f5-4e78-868d-ec01d077bb2e | This field is pre-populated with the resource group from your sandbox. |
| Account Name | *Enter a unique name* | Enter a unique name to identify this Azure Cosmos DB account. Because documents.azure.com is appended to the name that you provide to create your URI, use a unique but identifiable name.  The account name can contain only lowercase letters, numbers, and the hyphen (-) character, and it must contain 3 to 50 characters. |
| API | SQL | The API determines the type of account to create. Azure Cosmos DB provides five APIs to suit the needs of your application: SQL (document database), Gremlin (graph database), MongoDB (document database), Azure Table, and Cassandra, each of which currently require a separate account.   Select **SQL**. At this time, the Azure Cosmos DB trigger, input bindings, and output bindings only work with SQL API and Graph API accounts. |
| Location | Select from the list | Choose the nearest one to you that is also one of the allowed *Sandbox regions* listed below. |

1. Leave all other fields in the **New account** blade at their default values.
2. **Sandbox regions**
3. The free sandbox allows you to create resources in a subset of Azure's global regions. Select a region from the following list when creating any resources:
   * West US 2
   * South Central US
   * Central US
   * East US
   * West Europe
   * Southeast Asia
   * Japan East
   * Brazil South
   * Australia Southeast
   * Central India
4. Select **Review + create** to review and validate the configuration.
5. On the next screen, select **Create** to provision and deploy the database account.
6. Deployment can take some time. So, wait for a **Deployment succeeded** message in the Notification Hub before proceeding.



1. Select **Go to resource** to navigate to the database account in the portal. We'll add a collection to the database next.

**Add a collection**

In Cosmos DB, a *container* holds arbitrary user-generated entities. For SQL and MongoDB API accounts, a container maps to a *collection*. Inside a collection, we store documents.

Let's use the Data Explorer tool in the Azure portal to create a database and collection.

1. Select **Data Explorer** > **New Collection**.
2. Under **Add collection**, enter the settings for the new collection.

| **Setting** | **Suggested value** | **Description** |
| --- | --- | --- |
| Database ID | **func-io-learn-db** | Database names must contain from 1 through 255 characters, and they cannot contain /, \, #, ?, or a trailing space.  You are free to enter whatever you want here, but we suggest **func-io-learn-db** as the name for the new database, and that's what we'll refer to in this unit. |
| Collection ID | **Bookmarks** | Enter **Bookmarks** as the name for our new collection. Collection IDs have the same character requirements as database names. |
| Partition key | **/id** | The partition key specifies how the documents in Cosmos DB collections are distributed across logical data partitions. We will use the id field as a convenience, as we are not concerned with database performance in this module. If you would like to learn more about Cosmos DB partition key strategies, please explore the Microsoft Learn Cosmos DB modules. |
| Throughput | 1000 RU | Change the throughput to 1000 request units per second (RU/s). If you want to reduce latency, you can scale up the performance later. |

1. Click **OK**. The Data Explorer displays the new database and collection. So, now we have a database. Inside the database, we've defined a collection. Next, we'll add some data, also known as documents.

**Add test data**

We've defined a collection in our database called **Bookmarks**. We want to store a URL and ID in each document, like a list of web page bookmarks.

You'll add data to your new collection using Data Explorer.

1. In Data Explorer, the new database appears in the Collections pane. Expand the **func-io-learn-db** database, expand the **Bookmarks** collection, select **Documents**, and then select **New Document**.
2. Replace the default content of the new document with the following JSON.

{

"id": "docs",

"URL": "https://docs.microsoft.com/azure"

}

1. After you've added the JSON to the **Documents** tab, select **Save**.

Notice that there are more properties than the ones we added. They all begin with an underline (\_rid, \_self, \_etag, \_attachments, \_ts). These are properties generated by the system to help manage the document.

| **Property** | **Description** |
| --- | --- |
| \_rid | The resource ID is a unique identifier that is also hierarchical per the resource stack on the resource model. It is used internally for placement and navigation of the document resource. |
| \_self | The unique addressable URI for the resource. |
| \_etag | Required for optimistic concurrency control. |
| \_attachments | The addressable path for the attachments resource. |
| \_ts | The time stamp of the last update of this resource. |

1. Let's add a few more documents into the collection. Create four more documents with the following content. Remember to save your work.

{

"id": "portal",

"URL": "https://portal.azure.com"

}

{

"id": "learn",

"URL": "https://docs.microsoft.com/learn"

}

{

"id": "marketplace",

"URL": "https://azuremarketplace.microsoft.com/marketplace/apps"

}

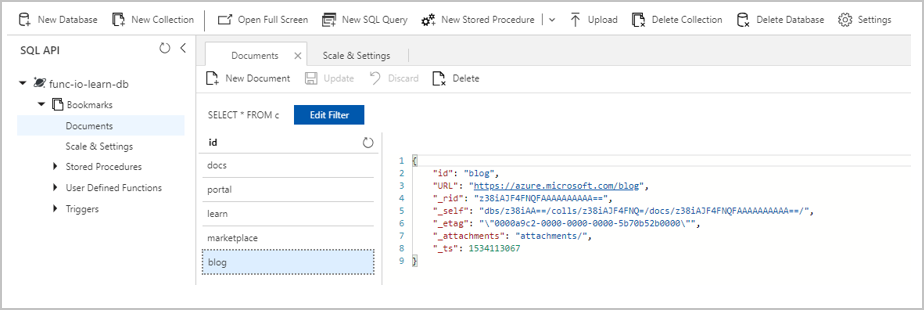
{

"id": "blog",

"URL": "https://azure.microsoft.com/blog"

}

1. When you've finished, your collection should look like the following:



You now have a few entries in your bookmark collection. Our scenario will work as follows. If a request arrives with, for example, "id=docs", you'll look up that ID in your bookmarks collection and return the URL https://docs.microsoft.com/azure. Let's make an Azure function that looks up values in this collection.

**Create your function**

1. Navigate to the function app that you created in the preceding unit.
2. Select the **Add** (**+**) button next to **Functions** to start the function creation process. The page displays the complete set of supported triggers.
3. Select **HTTP trigger**
4. Fill out the **New Function** dialog that appears to the right using the following values.

| **Field** | **Value** |
| --- | --- |
| Name | **find-bookmark** |
| Authorization level | **Function** |

1. Select **Create** to create your function. This action opens the *index.js* file in the code editor and displays a default implementation of the HTTP-triggered function.

**Verify the function**

You can verify what we have done so far by testing our new function as follows:

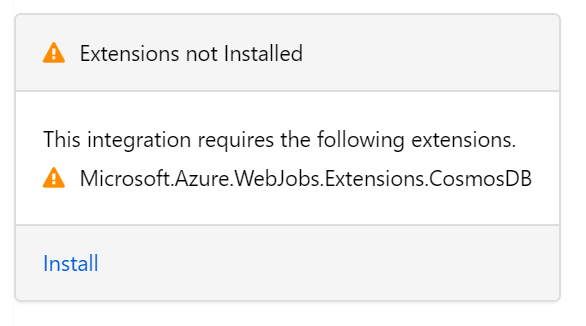
1. In your new function, click **Get function URL** at the top right, select **default (Function key)**, and then click **Copy**.
2. Paste the function URL you copied into your browser's address bar. Add the query string value &name=<yourname> to the end of the URL and press **Enter** to execute the request. You should get a response from the Azure Function right in the browser.

Now that we have our bare-bones function working, let's turn our attention to reading data from our Azure Cosmos DB, or in our scenario, our **Bookmarks** collection.

**Add an Azure Cosmos DB input binding**

To read data from the database, you need to define an input binding. As you'll see, you can configure a binding that can talk to your database in just a few steps.

1. Select **Integrate** in the left pane to open the integration tab. The template you used created an HTTP trigger and an HTTP output binding. Now add your new Azure Cosmos DB input binding.
2. Select **New Input** in the **Inputs** column. A list of all possible input binding types is displayed.
3. In the list, select **Azure Cosmos DB**, and then select **Select**. This action opens the Azure Cosmos DB input configuration page. Next, you'll set up a connection to your database.
4. If the following message appears in the **Azure Cosmos DB input** configuration UI telling you that you must install an extension, select **Install**.



1. Next to the **Azure Cosmos DB account connection** box, select **new**. This action opens the **Connection** window, which already has **Azure Cosmos DB account** and your Azure subscription selected. The only thing left to do is to select a database account ID.
2. In the "Create a database account" section, you had to supply an ID value. Find that value in the **Database Account**drop-down list, and then click **Select**.
3. A new connection to the database is configured and is shown in the **Azure Cosmos DB account connection** field. If you're curious about what is actually behind this abstract name, click *show value* to reveal the connection string.

You want to look up a bookmark with a specific ID, so let's tie the ID we receive to the binding.

1. In the **Document ID (optional)** field, enter {id}. This syntax is known as a *binding expression*. The function is triggered by an HTTP request that uses a query string to specify the ID to look up. Since IDs are unique in our collection, the binding will return either 0 (not found) or 1 (found) documents.
2. Carefully fill out the remaining fields on this page using the values in the following table. At any time, you can click on the information icon to the right of each field name to learn more about the purpose of each field.

| **Setting** | **Value** | **Description** |
| --- | --- | --- |
| Document parameter name | **bookmark** | The name used to identify this binding in your code. |
| Database name | **func-io-learn-db** | The database to work with. This value is the database name we set earlier in this lesson. |
| Collection Name | **Bookmarks** | The collection from which we'll read data. This setting was defined earlier in the lesson. |
| SQL Query (optional) | leave blank | We are only retrieving one document at a time based on the ID. So, filtering with the Document ID field is a better than using a SQL Query in this instance. We could craft a SQL Query to return one entry (SELECT \* from b where b.ID = {id}). That query would indeed return a document, but it would return it in a document collection. Our code would have to manipulate a collection unnecessarily. Use the SQL Query approach when you want to get multiple documents. |
| Partition key (optional) | **{id}** | Add the partition key that we defined when we created the **Bookmarks** Cosmos DB collection earlier. The key entered here (specified in input binding format {<key>}) must match the one in the collection. |

1. Select **Save** to save all changes to this binding configuration.

Now that you have your binding defined, it's time to use it in your function.

**Update function implementation**

1. Select your function, **find-bookmark**, to open *index.js* in the code editor. You've added an input binding to read from your database, so update the logic to use this binding.
2. Replace all code in *index.js* with the code from the following snippet and hit **Save**.

module.exports = function (context, req) {

var bookmark = context.bindings.bookmark

if(bookmark){

context.res = {

body: { "URL": bookmark.URL },

headers: {

'Content-Type': 'application/json'

}

};

}

else {

context.res = {

status: 404,

body : "No bookmarks found",

headers: {

'Content-Type': 'application/json'

}

};

}

context.done();

};

An incoming HTTP request triggers the function, and an id query parameter is passed to the Cosmos DB input binding. If the database finds a document that matches this ID, then the bookmark parameter will be set to the located document. In that case, we construct a response that contains the URL value found in the bookmarked document. If no document is found matching this key, we would respond with a payload and status code that tells the user the bad news.

## Try it out

1. Select **Get function URL** at the top right, select **default (Function key)**, and then select **Copy** to copy the function's URL.
2. Paste the function URL you copied into your browser's address bar. Add the query string value &id=docs to the end of this URL and press the Enter key on your keyboard to execute the request. You should see a response that includes a URL to that resource.
3. Replace &id=docs with &id=missing, and observe the response.