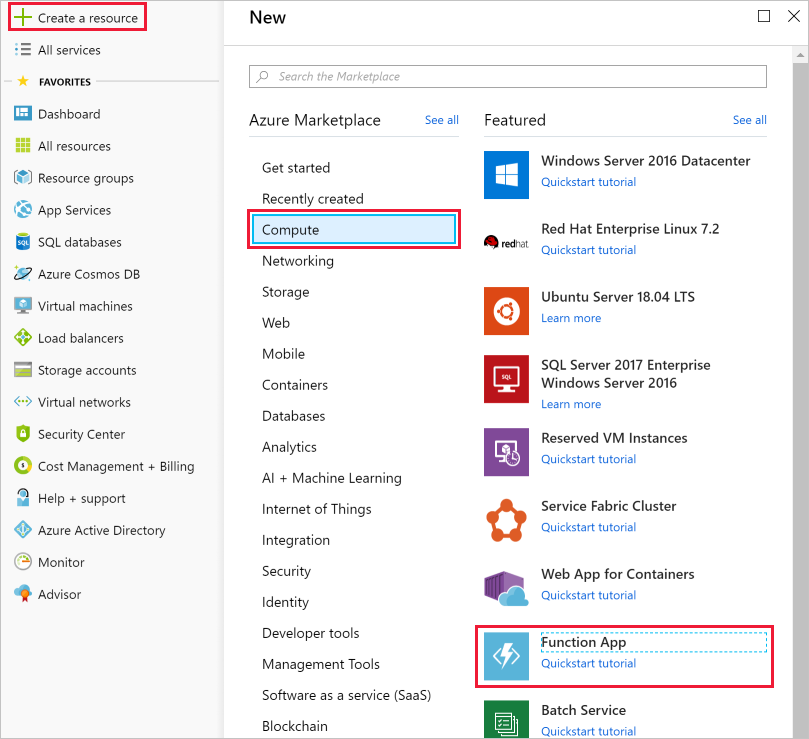
**Create a function app**

Let's create a function app in the Azure portal.

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, and then select **Get started > Serverless Function App** to open the Function App *Create* blade. Alternatively, you can use the **Compute > Function App** option, which will open the same blade.



1. Choose a globally unique app name. This will serve as the base URL of your service. For example, you can name it **escalator-functions-xxxxxxx**, where the x's can be replaced with your initials and your birth year. If this isn't globally unique, you can try any other combination. Valid characters are a-z, 0-9 and -.
2. Select the Azure subscription where you would like the function app hosted.
3. Select the existing resource group or create a new one.
4. Select **Windows** for OS.
5. For **Hosting Plan**, select **Consumption Plan**, which is the serverless hosting option.
6. Select the geographical location closest to you from the list below. In a production system, you would want to select a location near your customers or consumers of the function.

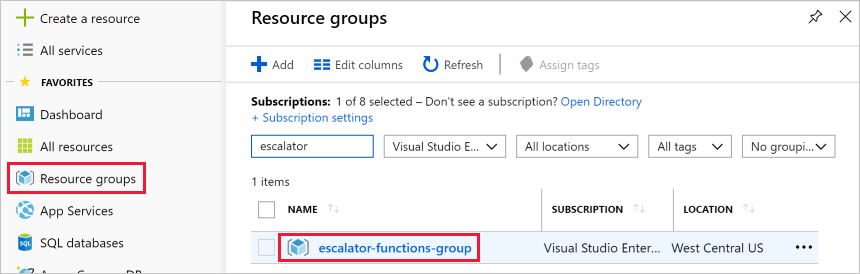
The free sandbox allows you to create resources in a subset of Azure's global regions. Select a region from the following list when creating any resources:

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

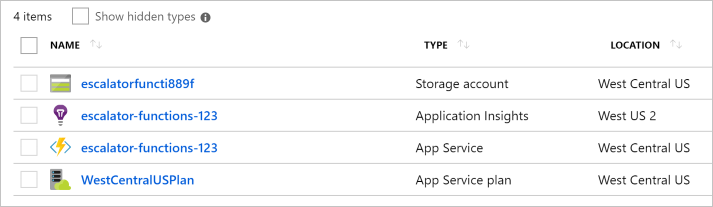
1. For **Runtime Stack**, select JavaScript from the dropdown, which is the language in which we implement the function examples in this exercise.
2. Create a new storage account. Azure will give it a name based on the app name. You can change it if you like, but it must also be unique.
3. Make sure that Azure Application Insights is **On** and select the region closest to you (or your customers).
4. Select **Create**; deployment will take a few minutes. You'll receive a notification once it's complete.

**Verify your Azure function app**

1. From the Azure portal left-hand menu, select **Resource groups**. You should then see a resource group you have selected to create the function app in the list of available groups.



1. Select the resource group. You should then see a resource list like the following list.



The item with the lightning bolt Function icon, listed as an App Service, is your new function app. You can click on it to open the details about the new function - it has a public URL assigned to it, if you open that in a browser, you should get a default web page that indicates your Function App is running.

### Quickstart templates

When adding your first function, you are presented with the Quickstart screen. This screen allows you to choose a trigger type (HTTP, Timer, or Data) and programming language (C#, JavaScript, F# or Java). Then, based on your selections, Azure will generate the function code and configuration for you with some sample code provided to display out the input data received in the log.

### Custom function templates

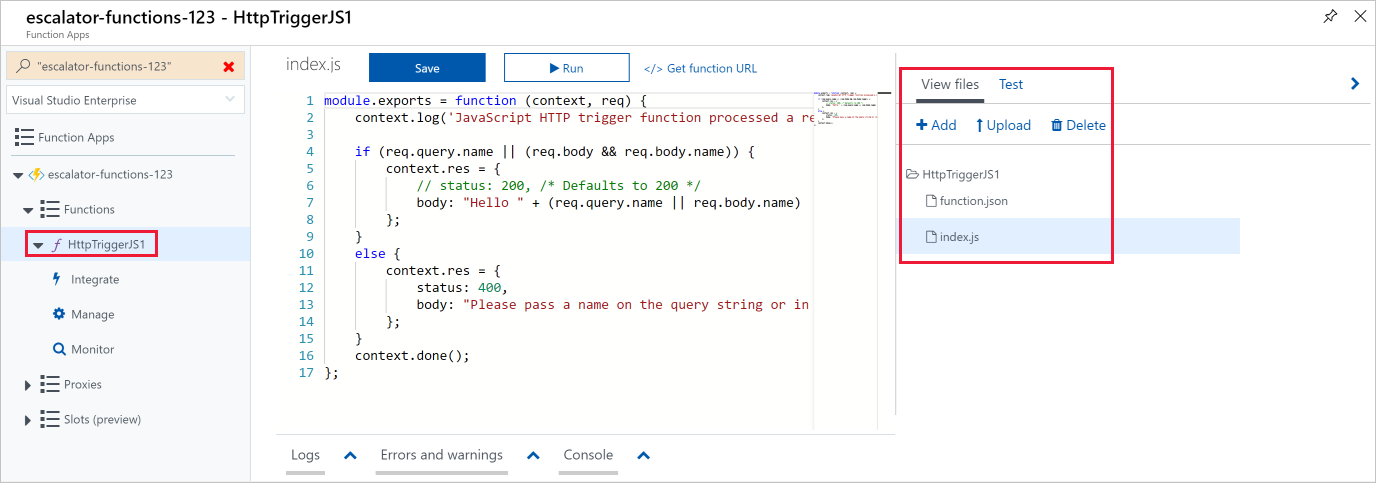
The selection of Quickstart templates provides easy access to the most common scenarios. However, Azure provides over 30 additional templates you can start with. These can be selected from the template list screen when creating subsequent functions or be selected by using the **Custom function** option on the Quickstart screen.

* HTTP trigger w/ C#, F#, or JavaScript
* Timer trigger w/ C#, F#, or JavaScript
* Queue trigger w/ C#, F#, or JavaScript
* Service Bus Queue trigger w/ C#, F#, or JavaScript
* Cosmos DB trigger w/ C# or JavaScript
* IoT Hub (Event Hub) w/ C#, F#, or JavaScript
* ... and many more

## Navigating to your function and files

When you create a function from a template, several files are created. For example, if you opted to use the Webhook + API Quickstart using JavaScript, the files generated would be a configuration file, **function.json**, and a source code file, **index.js**. The functions you create in a function app appear under the **Functions** menu item in the function app portal.

When you select a function in your function app, a code editor opens and displays the code for your function, as illustrated in the following screenshot.



As you can see in the preceding screenshot, there's a flyout menu on the right that includes a tab to **View files**. Selecting this tab shows the file structure that makes up your function.

## Testing your Azure function

Once you've created a function, you'll want to test it. There are a couple of approaches: manual execution and testing from within the Azure portal itself.

### **Manual execution**

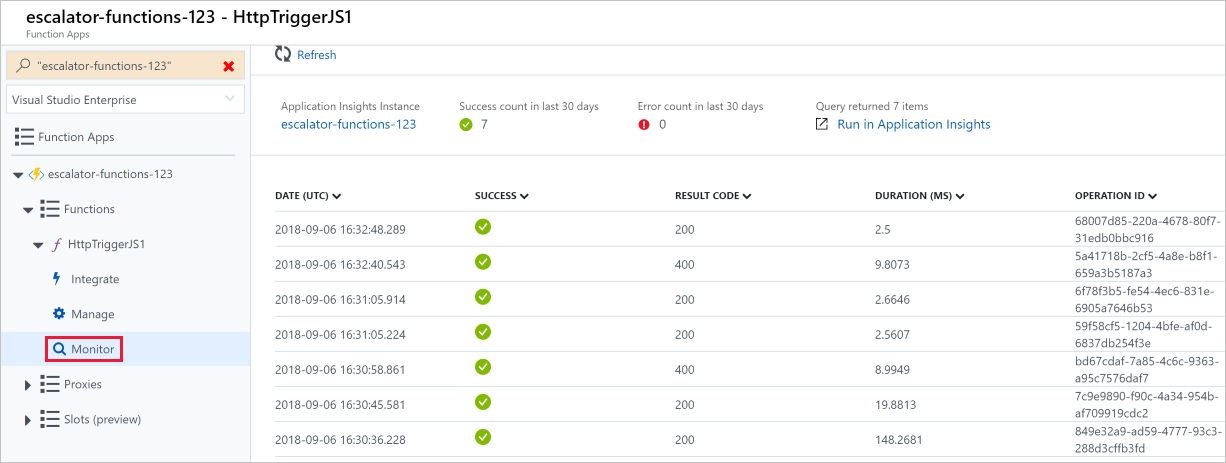
You can start a function by manually triggering the configured trigger. For instance, if you are using an HTTP trigger - you can use a tool such as Postman or cURL to initiate an HTTP request to your function endpoint URL, which is available from the HTTP trigger definition (**Get function URL**).

### **Testing in the Azure portal**

The portal also provides a convenient way to test your functions. On the right side of the code window, there is a flyout tabbed navigation menu. This menu contains a **Test** item. Expanding the menu and selecting this tab gives you another way to execute your function and view the result. When you click **Run** in this test window, the results are displayed in the output window, along with a status code.

## Monitoring dashboard

The ability to monitor your functions is critical during development and in production. The Azure portal provides a monitoring dashboard available if you turn on the Application Insights integration. In the function app navigation menu, once you expand the function node you'll see a **Monitor** menu item. This monitor dashboard provides a quick way to view the history of function executions and displays the timestamp, result code, duration, and operation ID populated by Application Insights.



## Streaming log window

You're also able to add logging statements to your function for debugging in the Azure portal. The called methods for each language are passed a "logging" object, which may be used to log information to the log window located in a tabbed flyout menu located at the bottom of the code window.

The following JavaScript code snippet shows how to log a message using the context.log method (the context object is passed to the handler).

JavaScriptCopy

context.log('Enter your logging statement here');

We could do the same thing in C# using the log.Info method. In this case, the log object is passed to the C# method processing the function.

C#Copy

log.Info("Enter your logging statement here");

### Errors and warnings window

You can locate the errors and warnings window tab in the same flyout menu as the log window. This window will show compilation errors and warnings within your code.

Let's continue with our gear drive example and add the logic for the temperature service. Specifically, we're going to receive data from an HTTP request.

**Function requirements**

First, we need to define some requirements for our logic:

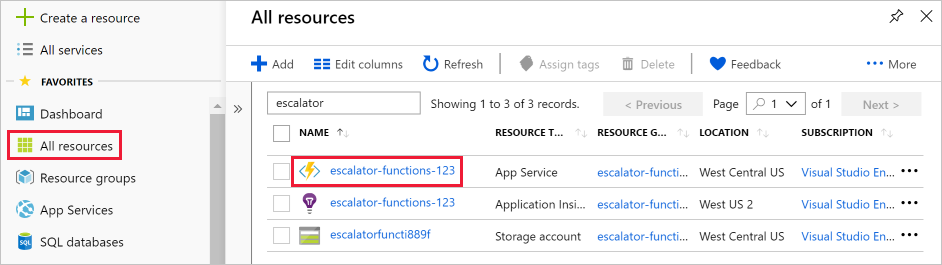
* Temperatures between 0-25 should be flagged as **OK**.
* Temperatures between 26-50 should be flagged as **CAUTION**.
* Temperatures above 50 should be flagged as **DANGER**.

**Add a function to our function app**

As we discussed in the preceding unit, Azure provides templates that help you get started building functions. In this unit, we'll use the HttpTrigger template to implement the temperature service.

As we discussed in the preceding unit, Azure provides templates that help you get started building functions. In this unit, we'll use the HttpTrigger template to implement the temperature service.

1. Sign in to the [Azure portal](https://portal.azure.com/learn.docs.microsoft.com).
2. Select the resource group from the first exercise by choosing **All resources** in the left-hand menu, and then selecting your resource group.
3. The resources for the group will then be displayed. Click the name of the function app that you created in the previous exercise by selecting the **escalator-functions-xxxxxxx** item (also indicated by the lightning bolt Function icon).



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. Enter **DriveGearTemperatureService** in the name field of the **New Function** dialog that appears. Leave the Authorization level as "Function" and press the **Create** button to create the function.
6. When your function creation completes, the code editor opens with the contents of the index.js code file. The default code that the template generated for us is listed in the following snippet.

module.exports = function (context, req) {

context.log('JavaScript HTTP trigger function processed a request.');

if (req.query.name || (req.body && req.body.name)) {

context.res = {

// status: 200, /\* Defaults to 200 \*/

body: "Hello " + (req.query.name || req.body.name)

};

}

else {

context.res = {

status: 400,

body: "Please pass a name on the query string or in the request body"

};

}

context.done();

};

Our function expects a name to be passed in either through the HTTP request query string or as part of the request body. The function responds by returning the message **Hello, {name}**, echoing back the name that was sent in the request.

On the right-hand side of the source view, you'll find two tabs. The **View files** tab lists the code and config file for your function. Select **function.json** to view the configuration of the function, which should look like the following:

{

"disabled": false,

"bindings": [

{

"authLevel": "function",

"type": "httpTrigger",

"direction": "in",

"name": "req"

},

{

"type": "http",

"direction": "out",

"name": "res"

}

]

}

This configuration declares that the function runs when it receives an HTTP request. The output binding declares that the response will be sent as an HTTP response.

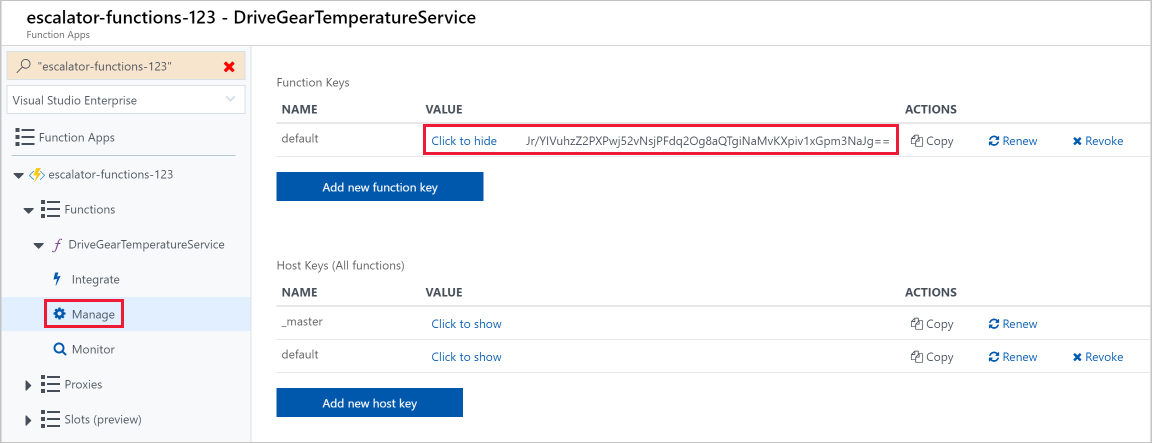
### **Securing HTTP triggers**

HTTP triggers let you use API keys to block unknown callers by requiring the key to be present on each request. When you create a function, you select the authorization level. By default, it's set to "Function", which requires a function-specific API key, but it can also be set to "Admin" to use a global "master" key, or "Anonymous" to indicate that no key is required. You can also change the authorization level through the function properties after creation.

Since we specified "Function" when we created this function, we will need to supply the key when we send the HTTP request. You can send it as a query string parameter named code, or as an HTTP header (preferred) named x-functions-key.

The function and master keys are found in the **Manage** section when the function is expanded. By default, they are hidden, and you need to display them.

1. Expand your function and select the **Manage** section, show the default Function Key, and copy it to the clipboard.



1. Next, from the command line where you installed the **cURL** tool, format a cURL command with the URL for your function, and the Function key.

* Use a POST request.
* Add a Content-Type header value of type application/json.
* Make sure to replace the URL below with your own.
* Pass the Function Key as the header value x-functions-key.

curl --header "Content-Type: application/json" --header "x-functions-key: <your-function-key>" --request POST --data "{\"name\": \"Azure Function\"}" https://<your-url-here>/api/DriveGearTemperatureService

## Add business logic to the function

Next, let's add the logic to the function that checks temperature readings that it receives and sets a status for each.

Our function is expecting an array of temperature readings. The following JSON snippet is an example of the request body that we'll send to our function. Each reading entry has an ID, timestamp, and temperature.

{

"readings": [

{

"driveGearId": 1,

"timestamp": 1534263995,

"temperature": 23

},

{

"driveGearId": 3,

"timestamp": 1534264048,

"temperature": 45

},

{

"driveGearId": 18,

"timestamp": 1534264050,

"temperature": 55

}

]

}

Next, we'll replace the default code in our function with the following code that implements our business logic.

1. Open the **index.js** file and replace it with the following code.

module.exports = function (context, req) {

context.log('Drive Gear Temperature Service triggered');

if (req.body && req.body.readings) {

req.body.readings.forEach(function(reading) {

if(reading.temperature<=25) {

reading.status = 'OK';

} else if (reading.temperature<=50) {

reading.status = 'CAUTION';

} else {

reading.status = 'DANGER'

}

context.log('Reading is ' + reading.status);

});

context.res = {

// status: 200, /\* Defaults to 200 \*/

body: {

"readings": req.body.readings

}

};

}

else {

context.res = {

status: 400,

body: "Please send an array of readings in the request body"

};

}

context.done();

};

The logic we added is straightforward. We iterate over the array of readings and check the temperature field. Depending on the value of that field, we set a status of **OK**, **CAUTION**, or **DANGER**. We then send back the array of readings with a status field added to each entry.

Notice the log statements. When the function runs, these statements will add messages in the log window.

## Test our business logic

In this case, we're going to use the **Test** pane in the portal to test our function.

1. Open the **Test** window from the right-hand side flyout menu.
2. Paste the sample request into the request body text box.

{

"readings": [

{

"driveGearId": 1,

"timestamp": 1534263995,

"temperature": 23

},

{

"driveGearId": 3,

"timestamp": 1534264048,

"temperature": 45

},

{

"driveGearId": 18,

"timestamp": 1534264050,

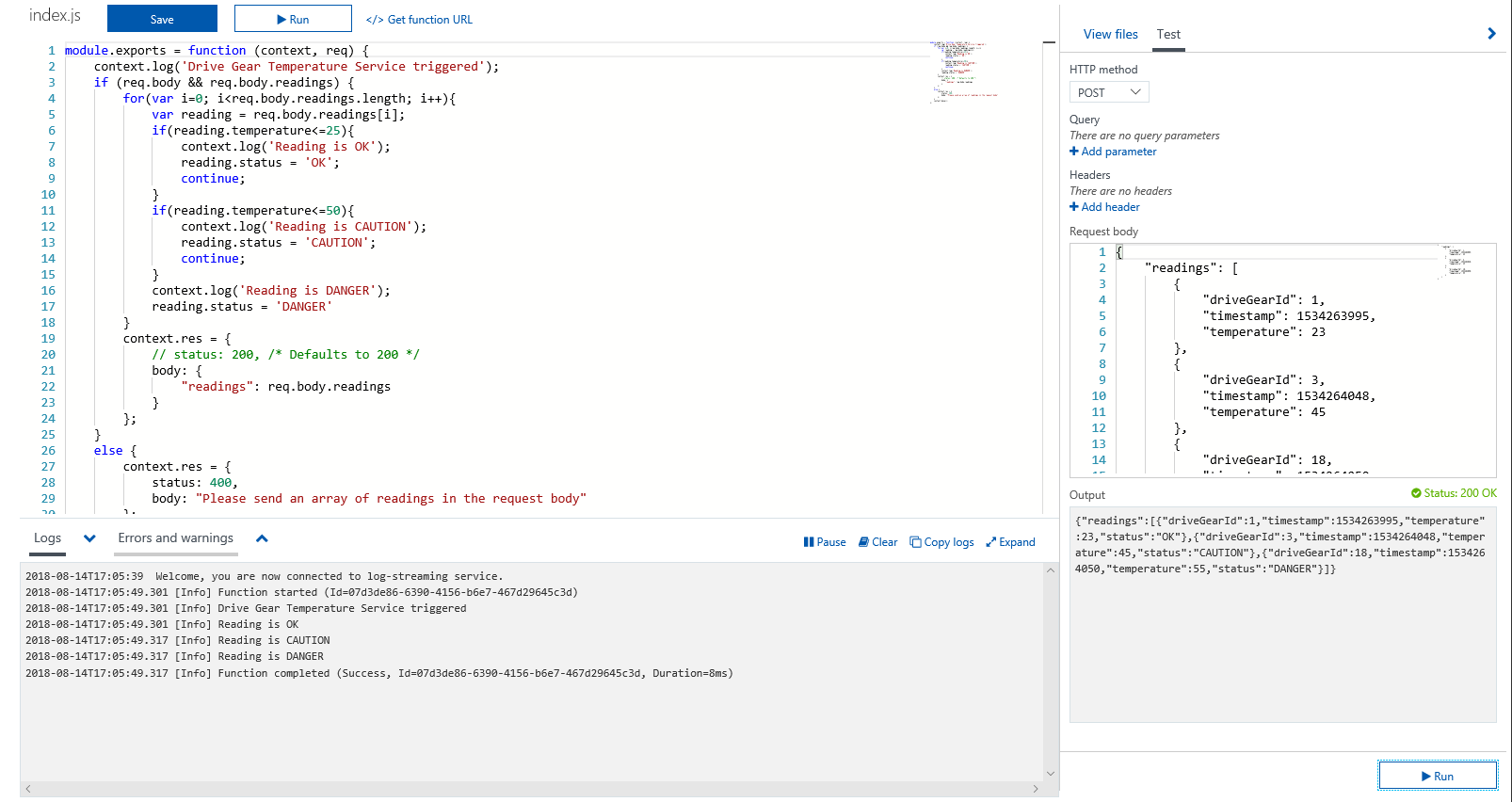
"temperature": 55

}

]

}

1. Select **Run** and view the response in the output pane. To see log messages, open the **Logs** tab in the bottom flyout of the page. The following screenshot shows an example response in the output pane and messages in the **Logs**pane.



1. You can see in the output pane that our status field has been correctly added to each of the readings.
2. If you navigate to the **Monitor** dashboard, you'll see that the request has been logged to Application Insights.

