Lab 4: Global Connectivity

Learning outcome: Understand how Wireless technology can be used to connect to a Cloud-based IoT service, and how a full stack implementation might look.

# Introduction

## Lab overview

In this lab, we will program the embedded device to communicate with a service in the cloud, by transmitting sensor data through the Wi-Fi interface to a serverless function hosted in Google Cloud. This function will decode the sensor data, and store it in a database. We will build a small website for displaying the data.

## Hardware and Software Requirements

* The DISCO-L475VG-IOT01A board
* Mbed Studio or another suitable development environment
* A Google Cloud account
  + Available free by signing up here: <https://cloud.google.com>
  + Additional charges may apply if you exceed the daily usage limit.

# Getting Started

## The Development Board

On the embedded device, we are going to build our code from scratch, but we will need to add some additional libraries to enable HTTP communication.

Start by creating a blank Mbed OS 6 project, and add the following libraries:

* The Board Support files for the IoT Discovery board:  
  https://os.mbed.com/teams/ST/code/BSP\_B-L475E-IOT01/
* The ISM43362 Wi-Fi component:  
  https://github.com/ARMmbed/wifi-ism43362/
* An Mbed OS 6 HTTP client:  
  https://github.com/rasmus0201/mbed-http-client.git

Now, we need to fill in some configuration data for the Mbed project, so that we can connect to Wi-Fi, and ultimately communicate with Google Cloud. To do this, you need to create a new mbed\_app.json file, and insert the following configuration:

|  |
| --- |
| {  "target\_overrides": {  "\*": {  "nsapi.default-wifi-security": "WPA\_WPA2",  "nsapi.default-wifi-ssid": "\"**<YOUR-SSID>**\"",  "nsapi.default-wifi-password": "\"**<YOUR-PASSWORD>**\"",  "platform.stdio-baud-rate": 115200,  "rtos.main-thread-stack-size": 8192,  "target.printf\_lib": "std"  },  "DISCO\_L475VG\_IOT01A": {  "target.components\_add": ["ism43362"],  "ism43362.provide-default": true,  "target.network-default-interface-type": "WIFI"  }  }  } |

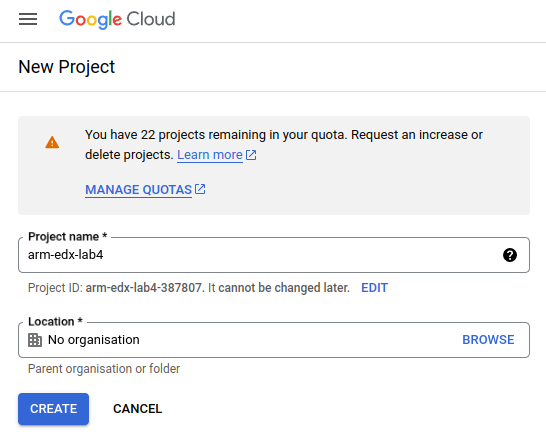
You will need to replace **<YOUR-SSID>** and **<YOUR-PASSWORD>** with your own Wi-Fi credentials.

## The Cloud

Now we move to the Cloud. To get started, create a project for this lab in the Google Cloud console. If you don’t already have a Google account, you’ll need to register for one, and once you’re all set-up, you can access the Google Cloud Developer Console:

<https://console.cloud.google.com>

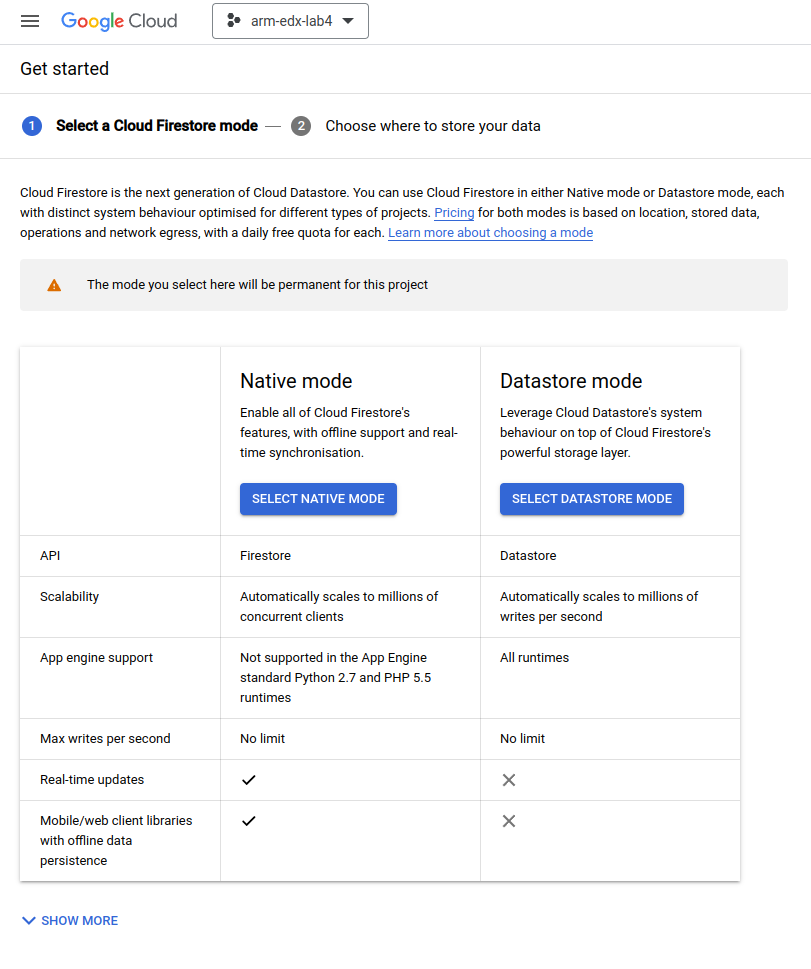
Then, create a project, giving it a name such as “arm-edx-lab4” in the console:



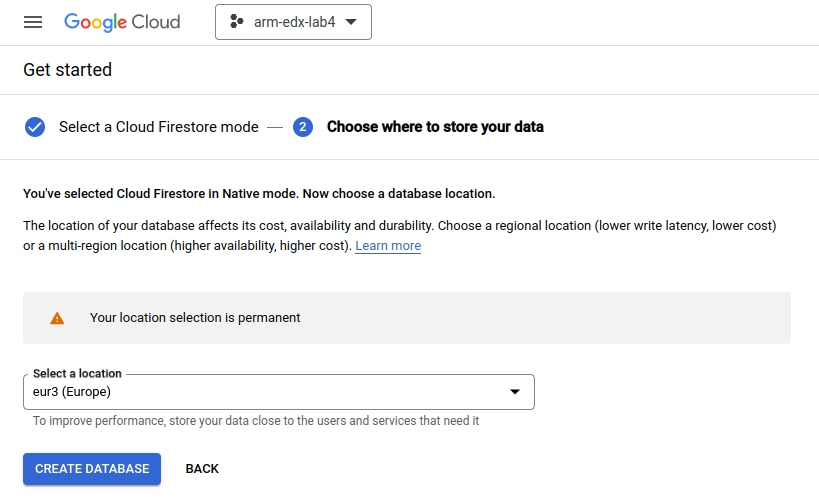
**Important: Take note of the Project ID, as you’ll need this later. You’ll always be able to find it again, but it would be convenient to note it down somewhere for quick reference.**

Make sure this is the active project, and in the list of services on the left pin the “Cloud Functions” service for quick access. You can find this in the list of services by scrolling down to “Serverless”, then hovering over “Cloud Functions”, and clicking the pin. This will pull it to the top of the list. Next, we need to enable the “Firestore” database service. Scroll down to “Databases”, and pin “Firestore” to the list, then click on it to begin the setup process.

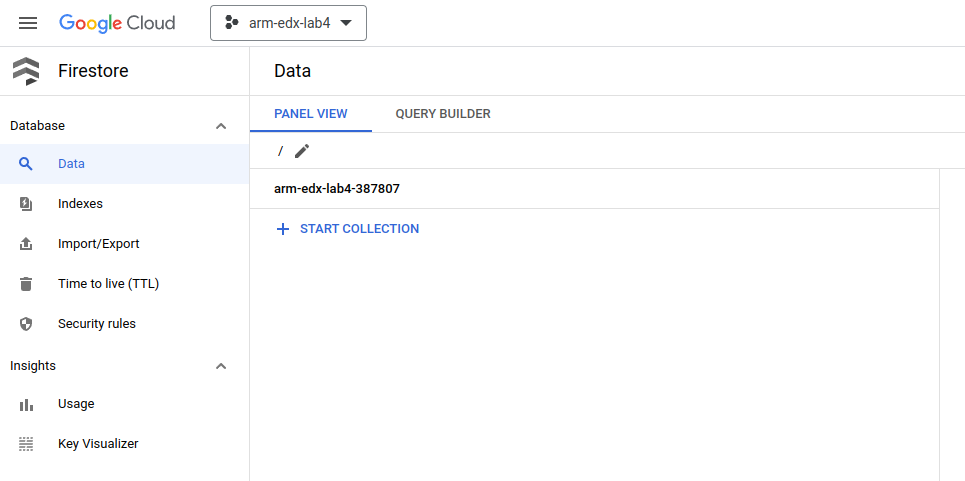
You should be presented with the following screen:



Choose to activate the database in “Native Mode”, and then on the following screen select a location that’s geographically close to you:

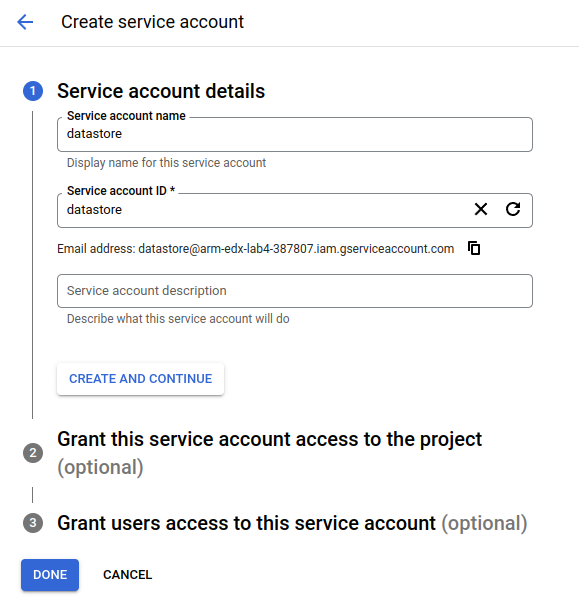


Click “Create Database”, and a couple of minutes later you should see the database management screen:

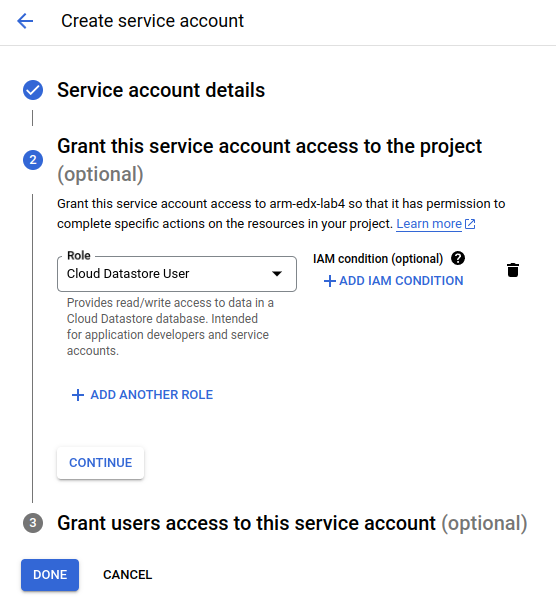


We also need to create a “service account” which will allow the Cloud Functions to interrogate and update the database. To do this, navigate to the “Credentials” section under “APIs and services” in the console. Click “Manage service accounts” from the “Service Accounts” list, and then choose “Create service account” from the top menu.

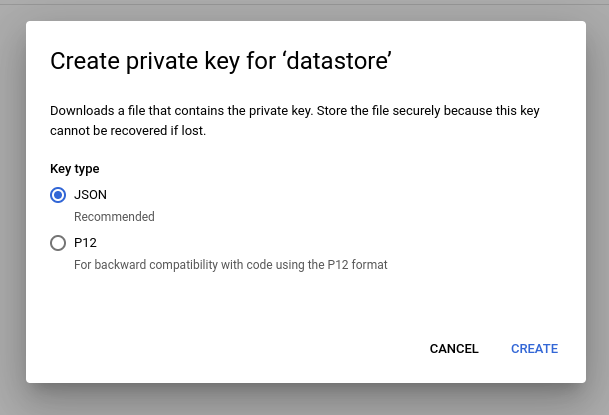
You should see a screen like the following:



Enter a name for the service account, such as “datastore”, and click “Create and Continue”. On the next screen (section 2), select “Cloud Datastore User” for a role (you may need to use the filter box to search for this role):



Then click “Done”, and the service account will be created. Following this, click on the newly created service account in the list, and navigate to the “Keys” tab. Click “Add Key”, and choose to create a JSON key:



Clicking “Create” will download the newly created key to your computer. Make sure you keep it safe -- and private!

Now that everything is created and configured, we can start to write our Cloud Functions.

# 

# Cloud Functions

Cloud Functions are Google’s Function-as-a-Service (FaaS) offering in their Cloud ecosystem. Sometimes, Function-as-a-Service is called “Serverless”, because you (as a developer) do not need to manage the underlying servers that the functions run on. They are literally individual functions that are invoked when triggered by some event, such as a HTTP request.

We are going to use Cloud Functions to support communication from the embedded device to the Cloud, by having the board send a web request to a Cloud Function, containing the current environmental sensor data.

We need to make two functions to support our project:

1. **StoreSensorData:** A function to receive sensor data, and store it in the database.
   * This will be used by the embedded device to upload data.
2. **GetRecentSensorData:** A function to retrieve stored sensor data from the database.
   * This will be used by the simple webpage to download data.

To get started, navigate to the “Cloud Functions” service in the Console.

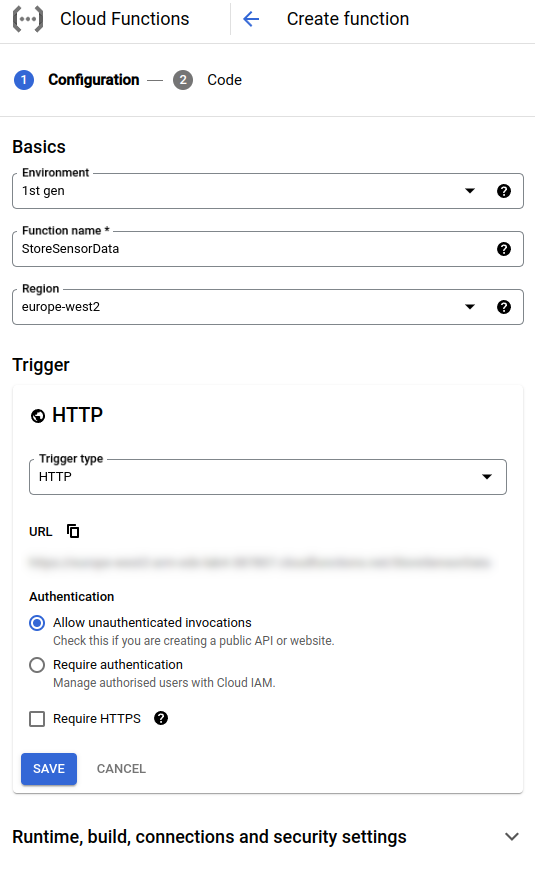
## The “StoreSensorData” Function

In the Cloud Functions screen, start by creating a new function. The first time you do this, you’ll be prompted to enable some additional APIs. Click “Enable” to action this, and wait for a minute for the APIs to be enabled.

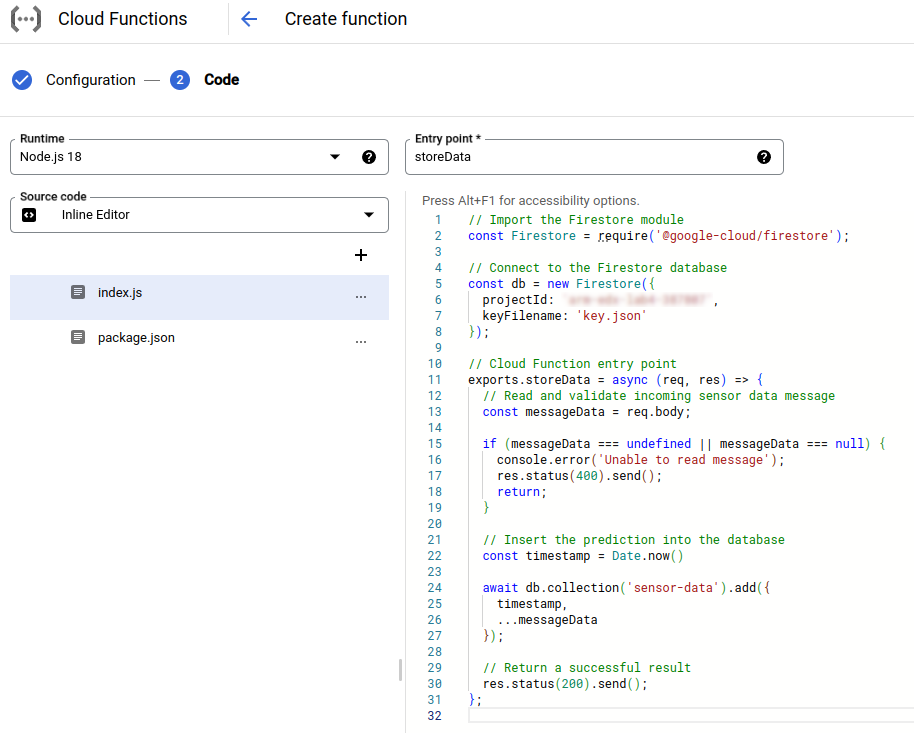
Now, you should be presented with a configuration screen. Enter a name for the function, e.g. “StoreSensorData”, choose the same region as you choose when initialising the Firestore database, then select an HTTP trigger. For the HTTP trigger settings, choose “Allow unauthenticated invocations”, and untick “Require HTTPS”.

Whilst this significantly reduces the security of invoking the Cloud Function, it significantly simplifies the implementation on the embedded device, and for example purposes is acceptable. If you were to be building a production system, you would carefully analyse the security requirements of your system, and almost certainly choose to implement authentication and encryption.

The configuration screen should look like the following:



Click Save, and you should be presented with a code editor:



Here, there is a list of files on the left, and an editor on the right. Change the “Entry point” to “storeData”, and then making sure you’re editing the index.js file, insert the following code - making sure you replace **<YOUR-PROJECT-ID>** with the ID (not the name) of your project:

|  |
| --- |
| // Import the Firestore module  **const** Firestore = require('@google-cloud/firestore');  // Connect to the Firestore database  **const** db = new Firestore({  projectId: '**<YOUR-PROJECT-ID>**',  keyFilename: 'key.json'  });  // Cloud Function entry point  exports.storeData = **async** (req, res) => {  // Read and validate incoming sensor data message  **const** messageData = req.body;  **if** (messageData === undefined || messageData === null) {  console.error('Unable to read message');  res.status(400).send();  **return**;  }  // Insert the data into the database  **const** timestamp = Date.now()  **await** db.collection('sensor-data').add({  timestamp,  ...messageData  });  // Return a successful result  res.status(200).send();  }; |

|  |
| --- |
| **What’s happening here?**  This code will run when a HTTP request is made to the function entrypoint, which is an URL. The code takes the incoming data, and inserts it into the database, along with a timestamp.  It does this by creating a reference to the “sensor-data” collection in the Firestore database, and then inserting a record that contains the current time, and the data contained within the web request.  Technically, this function allows us to upload arbitrary data, and really we should support some kind of validation - but for the purposes of this lab, we’ll just accept any incoming data. |

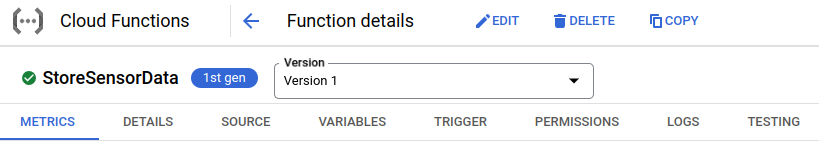
Now, select package.json from the file list, and update its contents to the following:

|  |
| --- |
| {  "name": "store-sensor-data",  "version": "0.0.1",  "dependencies": {  "@google-cloud/firestore": "^6.4.2"  }  } |

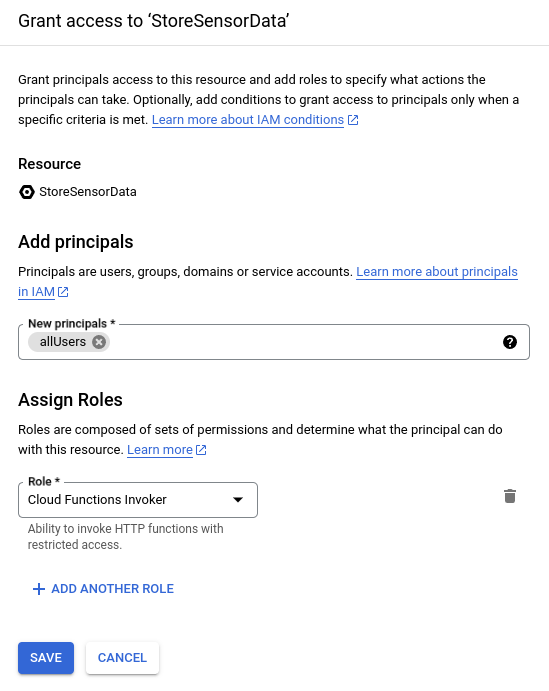
This indicates to the FaaS runtime that the function needs the “@google-cloud/firestore” library, so that the code can interact with the database.

Finally, we need to add the private key of the service account we created earlier. To do this, create a new file called key.json (click the plus button in the file list), and copy-and-paste the entire contents from the downloaded service account key file into the editor. This key allows the function to create and update data in the database.

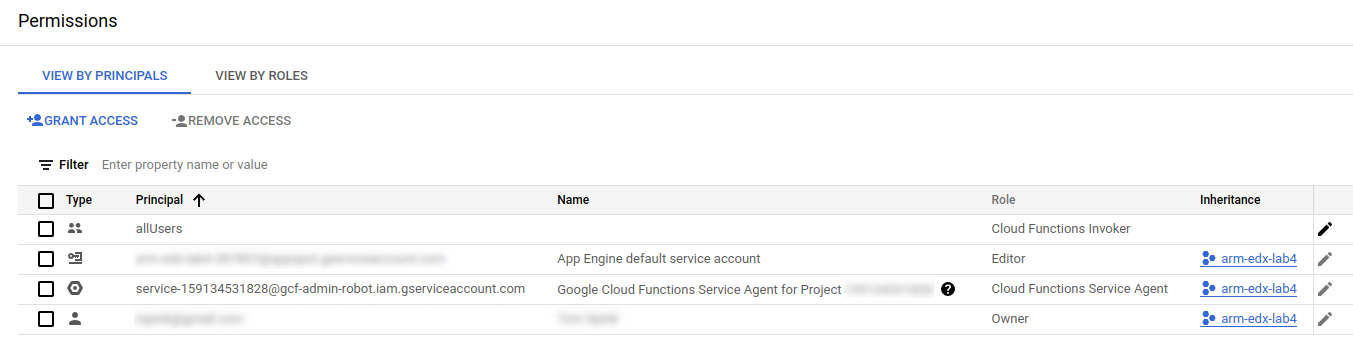
Now, click “Deploy”! The function will take a few minutes to deploy, but you should be presented with a new screen that will tell you when deployment is complete. If everything went well, a green tick will appear next to the function name:



We now need to add permissions to allow any user to invoke this function. To do this, navigate to the “Permissions” tab, and click “Grant Access”. In the popup window that appears, enter “allUsers” for the “Principal”, and choose “Cloud Functions Invoker” for the role:



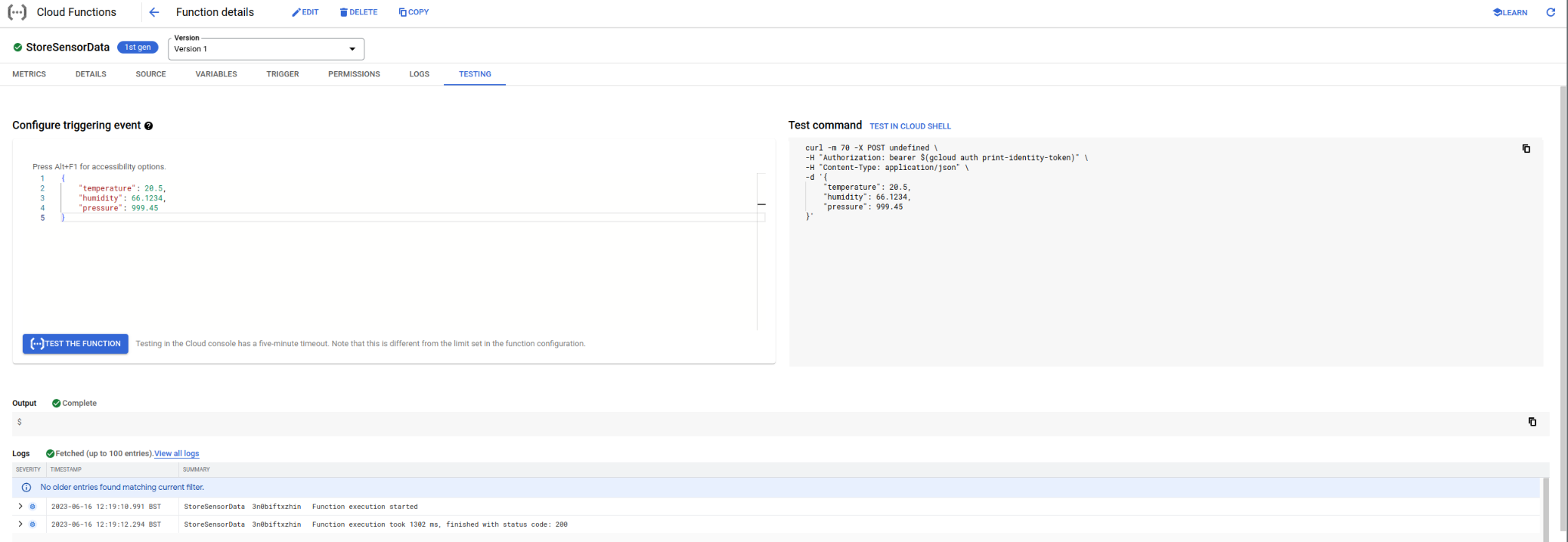
Click “Save”, and the permission should be added to the list:



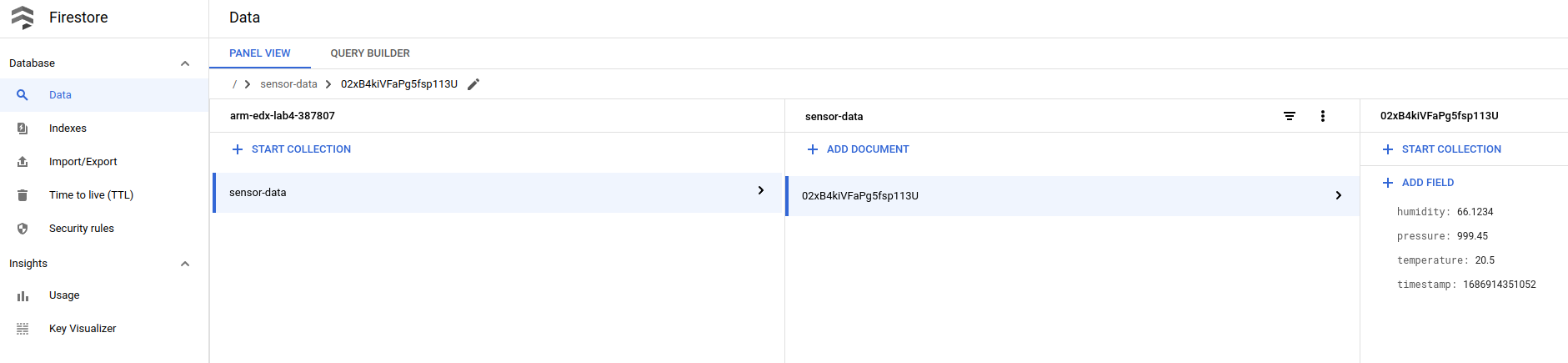
We should now test the function, to make sure it is inserting data into the database. To do this, navigate to the “Testing” tab, and in the “Configure triggering event” editor on the left, input the following JSON test data:

|  |
| --- |
| {  "temperature": 20.5,  "humidity": 66.1234,  "pressure": 999.45  } |

Click “Test the function”, and a few seconds later, you should see a successful result:



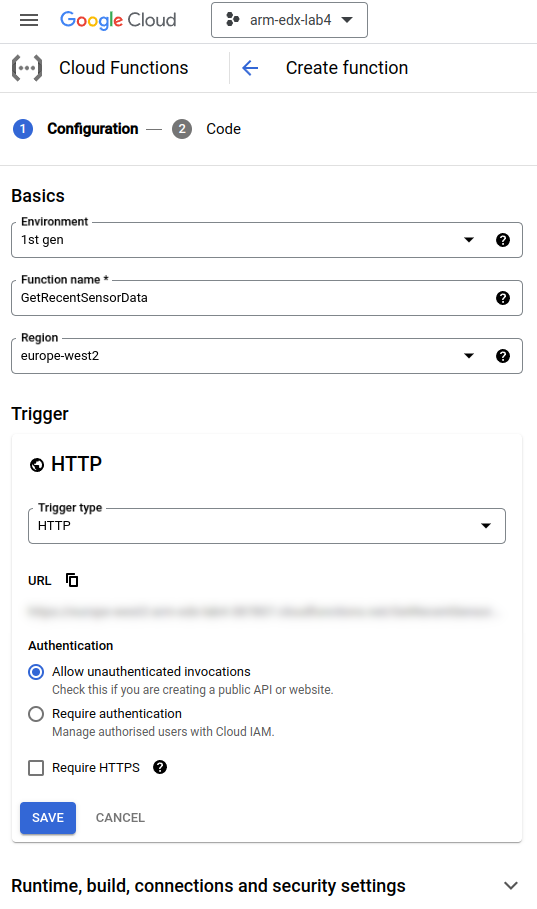
Finally, let’s check to make sure the data really did appear in the database. Navigate to “Firestore” in the console, and you should see that the “sensor-data” collection was created, and our test data appears:



## 

## Retrieving Sensor Data

The next function we need to create is one to access the most recent data from the database. To get started, navigate back to “Cloud Functions”, and create a new Cloud Function as before, but call it “GetRecentSensorData”:



And, in the code editor, change the entry point to “getData”, and insert the following code into index.js (remembering to replace **<YOUR-PROJECT-ID>** again):

|  |
| --- |
| // Import the Firestore module  const Firestore = require('@google-cloud/firestore');  // Connect to the Firestore database.  **const** db = **new** Firestore({  projectId: '**<YOUR-PROJECT-ID>**',  keyFilename: 'key.json'  });  exports.getData = **async** (req, res) => {  // Permit cross-site invocation.  res.set('Access-Control-Allow-Origin', '\*');  **if** (req.method === 'OPTIONS') {  // Settings for cross-site invocation.  res.set('Access-Control-Allow-Methods', 'GET');  res.set('Access-Control-Allow-Headers', 'Content-Type');  res.set('Access-Control-Max-Age', '3600');  res.status(204).send('');  } **else** {  // Retrieve sensor data from the DB.  **const** coll = db.collection('sensor-data');  **const** sensorDataSnapshot = **await** coll  .orderBy('timestamp')  .limit(10)  .get();  **const** sensorData = sensorDataSnapshot.docs.map(doc => doc.data());  // Return the activity data as a JSON object.  res.json(sensorData);  }  }; |

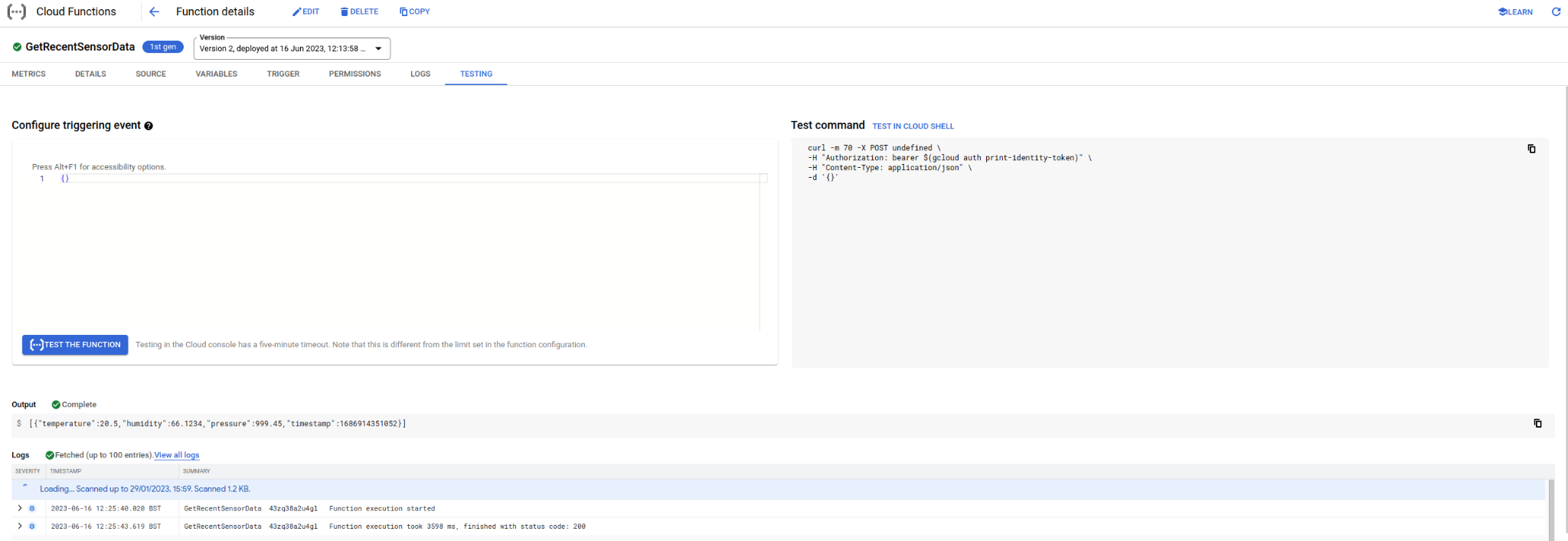
|  |
| --- |
| **What’s happening here?**  This function will be called from the web application, to retrieve the last 10 data items in the database. It works by simply querying the database for all sensor data, ordering them by timestamp, then limiting the result to 10 items. These records are then returned by the function.  There’s also a bit of magic to allow the function to be called from a web page - this is called CORS or *Cross-origin Resource Sharing*. This needs to be enabled, so that web pages on different domains can access the function. Here, we’re being lenient and allowing the function to be called from any web page. |

You also need to update package.json to include the firestore dependency as before:

|  |
| --- |
| {  "name": "get-sensor-data",  "version": "0.0.1",  "dependencies": {  "@google-cloud/firestore": "^6.4.2"  }  } |

And, again, create a key.json file containing the contents of the service account private key. Click “Deploy”, and wait for the function to be deployed. Navigate to the Permissions tab, and add the “allUsers” permission as before. When the green tick appears, we’re ready to test this function.

Navigate to the “Testing” tab, and this time, we don’t need to specify any input data. Instead, just click on “Test the function”, and after a few seconds you should see the following:



Here, in the “Output” section, you should see the data that was inserted when you tested the first Cloud Function.

# Sending Sensor Readings

Now that our Cloud Functions are working, we can start developing the embedded system, and sending real sensor data from the board to the cloud service. Enter your development environment, and as in previous labs, start by including the necessary headers at the top of the program:

|  |
| --- |
| **#include** "http\_request.h"  **#include** "mbed.h"  **#include** "stm32l475e\_iot01.h"  **#include** "stm32l475e\_iot01\_hsensor.h"  **#include** "stm32l475e\_iot01\_psensor.h"  **#include** "stm32l475e\_iot01\_tsensor.h"  static BufferedSerial serial\_port(USBTX, USBRX, 9600);  FileHandle \*mbed::mbed\_override\_console(int fd)  {  return &serial\_port;  } |

Then, we need a function to initialise the sensors:

|  |
| --- |
| **static** **void** initialiseSensors() {  BSP\_TSENSOR\_Init();  BSP\_HSENSOR\_Init();  BSP\_PSENSOR\_Init();  } |

Now, we need a function that will take sensor readings and send them to the StoreSensorData Cloud Function. Make sure you fill in **<YOUR-TRIGGER-URL>** with the URL for your own StoreSensorData function. You can find this, and copy it to the clipboard, by navigating to the “Trigger” tab in the Cloud Function editor, and looking at the “Trigger URL”.

**Important note:** You need to use the HTTP version of the URL, and not the HTTPS version, as we haven’t configured the embedded system to work with HTTPS (SSL) certificates. Make sure your trigger URL begins with http:// and NOT https://.

|  |
| --- |
| **static** **void** sendSensorData(NetworkInterface \*net) {  **const** **char** messageFormat[] =  "{ \"temperature\": %f, \"humidity\": %f, \"pressure\": %f }";  **char** messageData[256] = {0};  sprintf(messageData, messageFormat, BSP\_TSENSOR\_ReadTemp(),  BSP\_HSENSOR\_ReadHumidity(), BSP\_PSENSOR\_ReadPressure());  **auto** \*req = **new** HttpRequest(  net, HTTP\_POST,  "http://**<YOUR-TRIGGER-URL>**");  req->set\_header("Content-Type", "application/json");  printf("Sending message: %s\n", messageData);  HttpResponse \*res = req->send(messageData, strlen(messageData));  **if** (!res) {  printf("Http request failed (error code %d)\n", req->get\_error());  }  **delete** req;  } |

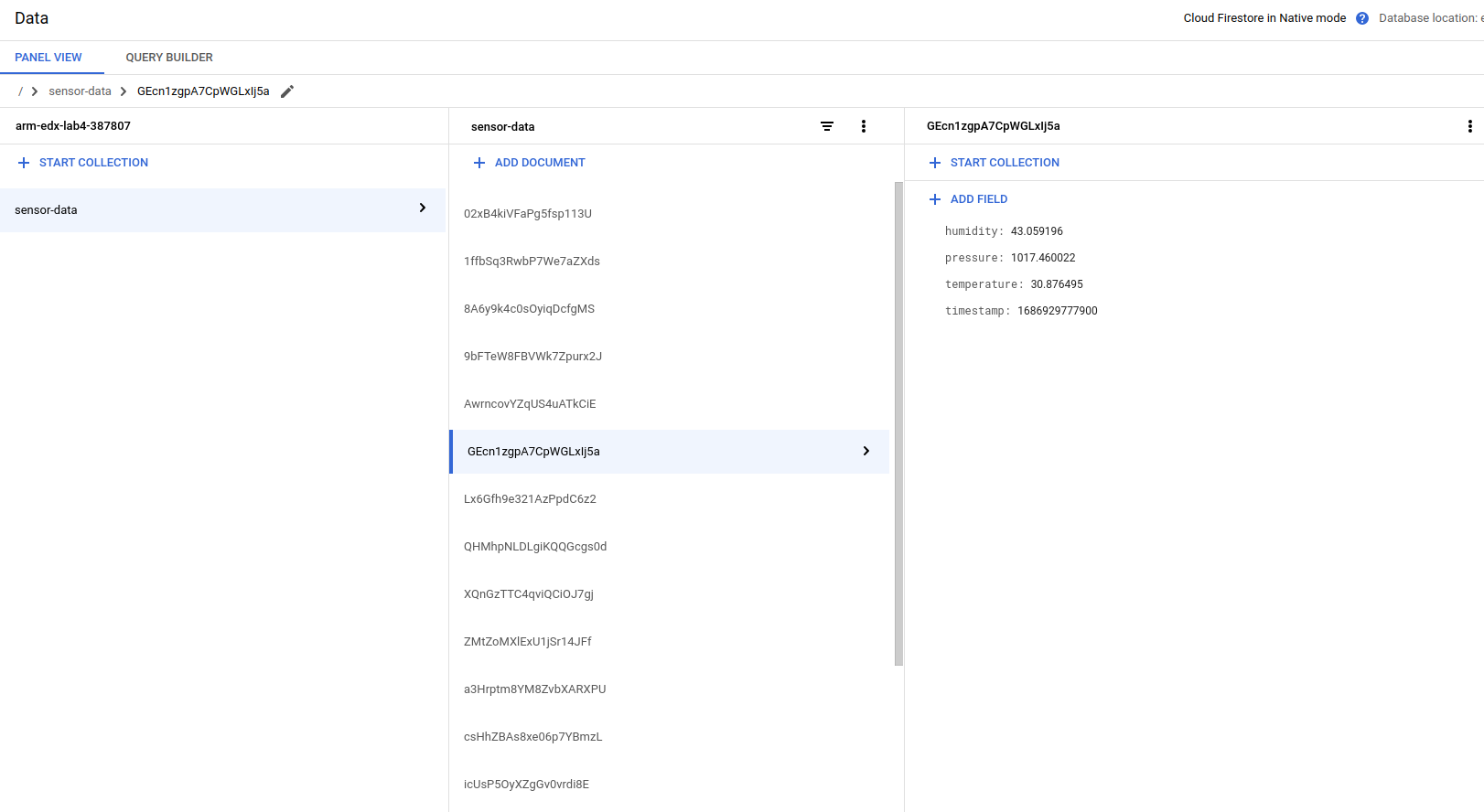
|  |
| --- |
| **What’s happening here?**  In this block of code, a JSON message (a cross-platform data format for describing structured data) is generated that contains the sensor readings.  It works by creating a template of the JSON message in a string, then using the sprintf function to fill in the placeholders. In the string, the %f directive tells sprintf to replace this with a floating point value.  The values come from the calls to the sensor reading functions (BSP\_TSENSOR\_ReadTemp(), etc), that are the last three parameters.  Once the message has been generated, it is sent to the Cloud by invoking the Cloud Function through a HTTP web request. |

Finally, we need to fill in the main function, to connect to the network, and trigger the send function every second:

|  |
| --- |
| **int** main() {  printf("Initialising sensors...\n");  initialiseSensors();  **auto** net = NetworkInterface::get\_default\_instance();  printf("Connecting to the network...\r\n");  // Connect to the network  nsapi\_size\_or\_error\_t result = net->connect();  **if** (result != 0) {  printf("Error! net->connect() returned: %d\r\n", result);  return -1;  }  SocketAddress ipaddr;  net->get\_ip\_address(&ipaddr);  printf("Connected with IP address: %s\r\n",  ipaddr.get\_ip\_address() ? ipaddr.get\_ip\_address() : "(none)");  // Transmit sensor readings every second  **while** (**true**) {  sendSensorData(net);  ThisThread::sleep\_for(1s);  }  } |

|  |
| --- |
| **What’s happening here?**  This block of code is the main function, and so is run when the board starts up. It begins by initialising the sensors we’re going to use, and then attempts to connect to the network (using the Wi-Fi credentials specified in the mbed\_app.json configuration file).  The IP address is displayed (for debugging purposes), and then an infinite loop is entered, with the sendSensorData function being called, followed by a delay of one second. |

Making sure your board is plugged in, build and run your application, and shortly after the board connects to the Wi-Fi, you should start seeing data appear in the Firestore database:



Try not to leave your board connected for long periods of time, as this will generate a significant amount of data, and it may exceed your free usage allowance.

# 

# The Web Application

We have already created the “GetRecentSensorData” function that allows us to retrieve the most recent 10 sensor readings, so now we can create a web page to display this data. This interface will be very basic, and simply read the data returned by the function we’ve created, and put it in a table.

To do this, we’ll create an HTML file, and upload it to cloud storage. Start by creating a new file in an editor of your choice, calling it e.g. “view.html” and put in the following code:

|  |
| --- |
| <!DOCTYPE html>  <html lang="en">  <head>  <title>Sensor Data Viewer</title>  <script src="https://unpkg.com/axios/dist/axios.min.js"></script>  </head>  <body>  <h1>Sensor Data</h1>  <table id="data-table" border="1">  <thead>  <tr>  <th>Time</th>  <th>Temperature</th>  <th>Humidity</th>  <th>Pressure</th>  </tr>  </thead>  <tbody></tbody>  </table>  <script>  const dataTable = document.getElementById('data-table');  const dataTableBody = dataTable.getElementsByTagName('tbody')[0];  function loadData() {  dataTableBody.innerHTML = '';  axios.get('**URL-TO-YOUR-CLOUD-FUNCTION**').then(function (data) {  for (const reading of data.data) {  const row = document.createElement('tr');  const timeCell = document.createElement('td');  const ts = new Date(reading.timestamp);  timeCell.innerText = ts.toLocaleString('en-GB');  row.appendChild(timeCell);  const tempCell = document.createElement('td');  tempCell.innerHTML = reading.temperature.toFixed(1) + ' &deg;C';  row.appendChild(tempCell);  const humCell = document.createElement('td');  humCell.innerText = reading.humidity.toFixed(0) + '%';  row.appendChild(humCell);  const presCell = document.createElement('td');  presCell.innerText = reading.pressure.toFixed(1);  row.appendChild(presCell);  dataTableBody.appendChild(row);  }  });  }  setInterval(loadData, 5000);  loadData();  </script>  </body>  </html> |

It looks like there’s a lot going on here, so let’s break it down.

The beginning of the file (in the HEAD section) gives the web page a title, and links to a library that we’re going to use to retrieve data (Axios). Next comes the body of our web page. This comprises a heading (H1), and a TABLE that’s initially empty, except for some headings. We give the table an identifier, so that we can reference it in code.

Now, the interesting bit is the SCRIPT at the bottom of the web page. This is where we retrieve the data from our function, and load it into the table.

We start by grabbing a reference to the body of the HTML table we’re going to populate. getElementById lets us find the table element, using the ID we added. Then, the body is retrieved by looking at the child elements, and finding the only one that is the TBODY type.

Next, in our loadData() function, we clear the contents of the table body, so that we overwrite whatever we’ve done previously.

We then make a call to our cloud function using the Axios web request library (remember to fill in the public URL to the GetRecentSensorData function, which you can get from the Google Cloud Console), and when we get the data back, we iterate over it one by one -- that’s the for loop.

Inside the for loop, for each data item, we create a new table row (TR), and add the four cells (TD) to it:

* The cell containing the timestamp, we use toLocaleString to nicely format the date.
* The cell containing the temperature, we use toFixed to reduce the number of decimal places to one, and add “°C” to the end.
* The cell containing the humidity, where we reduce the number of decimal places again and add “%” to the end.
* Finally, the cell containing the pressure, which we use toFixed on again.

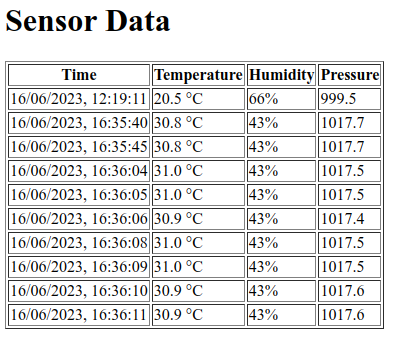
Once this loop has run, the table will be populated with the data. We then tell the browser to execute the loadData() function every five seconds (setInterval), and kick it off with an initial invocation.

You’re encouraged to experiment with styling and modifications you could make, to enhance the user experience - for example, can you also display the temperature in Fahrenheit?

Now, once you’ve created this file, you can upload it to the cloud, by going to the “Cloud Storage” section in the Google Cloud Console.

Here, you can create a new “bucket”, calling it e.g. “lab4-sensor-data-viewer” (note, bucket names need to be globally unique - the Cloud will tell you if you need to choose a different name). Once you’ve created the bucket, you can upload your HTML file, and then see it in action by finding the “Authenticated URL” property (click on the newly uploaded file), and visiting it in a web browser. If you want to make the web page public (not recommended), you can change the permissions.

If you’ve done all of this, you should see something like the following:



Congratulations! You’ve built a full-stack IoT system!

As with all of the labs, feel free to go the extra mile and change things up - if you already have web development experience, you could try using a UI framework, such as Bootstrap or Bulma to style your pages.