Lab 5: The Cloud

Learning outcome: Understand how a full-stack IoT system is developed

# Introduction

In this lab, you will combine your experiences from the previous labs to build a full-stack Internet-of-Things system, which will detect and classify movement, and present a visualisation on a mobile app.

Similar to Lab 4, this will involve sending sensor data from your embedded device to the cloud, but this time analysing it, and making the results available (in close to real-time) on an Android App.

## Hardware and Software Requirements

* The DISCO-L475VG-IOT01A board
* Mbed Studio or another suitable development environment
* A Google Cloud account
  + Continue to use the one from Lab 4
* An Android mobile device in developer mode
  + Instructions for entering developer mode for your specific phone model are generally found online.
* The latest version of Android Studio
  + Available here: <https://developer.android.com/studio>

## Overview

This lab comprises three main components:

* The embedded device
  + With Accelerometer and Gyroscope sensors
* Cloud connectivity
  + For receiving and processing sensor data
* Mobile application
  + For displaying movement classification results

To achieve this, the embedded device will periodically send sensor data to the cloud, using Cloud Functions as introduced in Lab 4. The device will record five seconds worth of sensor data, and transmit it in batches.

The Cloud Function will then classify the batched sensor data, and the result will be stored in a database, to maintain a classification history. The mobile application will periodically query the classification history, and display the periods that the user was either walking or running.

# Getting Started

Building on the Google Cloud infrastructure from Lab 4, we will be extending our work to use the Gyroscope and Accelerometer sensors. There’s no need to create a separate Google Cloud project for this - we can continue to use the one setup in Lab 4, and just create some new Cloud Functions. However, If you’d like to create a new project, you’ll need to follow all the set-up instructions from Lab 4, and remember to update your Project ID and the service account private key in all the relevant places–along with activating Cloud services such as Firestore.

# The Cloud

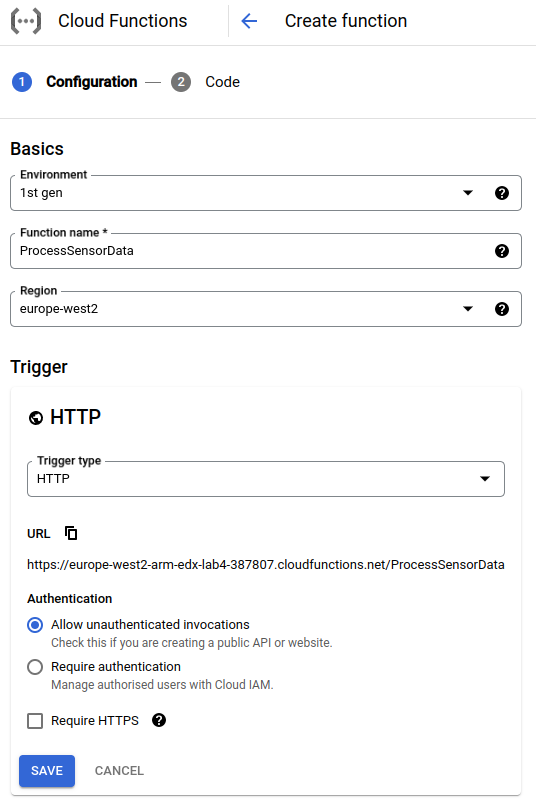
We’re going to create two new Cloud Functions, which are very similar to the ones in Lab 4:

1. **ProcessSensorData:** A function to receive batched sensor data from the embedded device, analyse the readings, and classify it into either: (a) standing still, (b) walking, or (c) running. The classification will be stored in the database.
   * This will be used by the embedded device to upload its sensor readings.
2. **GetActivityData:** A function to retrieve the stored activity data.
   * This will be used by the Mobile App to retrieve the activity data, ultimately to display it in the user interface.

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

## The “ProcessSensorData” Function

Start by creating a new Cloud Function, calling it “ProcessSensorData”:



Make sure you’ve selected “Allow unauthenticated invocations”, and you’ve unticked “Require HTTPS”. Click “Save”, and set the “Entry Point” to “process”. In the code editor, insert the following code for index.js, remembering to replace **YOUR-PROJECT-ID** with the ID 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'  });  *// Function to compute arithmetic mean over values in arr*  **function** computeMean(arr) {  **return** arr.reduce((a, b) => a + b) / arr.length;  }  *// Function to compute standard deviation over values in arr*  **function** computeStdDev(arr) {  **const** mean = computeMean(arr);  **const** variance = arr  .map(x => Math.pow(x - mean, 2))  .reduce((a, b) => a + b) / arr.length;  **return** Math.sqrt(variance);  }  *// Function to make an activity prediction, based on*  *// aggregated sensor data*  **function** predict(data) {  **if** (data.acceleration\_x\_std < 10 &&  data.acceleration\_y\_std < 10 &&  data.acceleration\_z\_std < 10) {  **return** {  label: 'still'  };  } **else** **if** (data.acceleration\_x\_std < 100 &&  data.acceleration\_y\_std < 100 &&  data.acceleration\_z\_std < 100) {  **return** {  label: 'walking'  };  } **else** {  **return** {  label: 'running'  };  }  }  *// Cloud Function entry point*  exports.process = **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**;  }  *// Extract individual readings for accelerometer*  *// and gyro axes into arrays.*  **const** accelX = messageData.readings.map(r => r.accel.x);  **const** accelY = messageData.readings.map(r => r.accel.y);  **const** accelZ = messageData.readings.map(r => r.accel.z);  **const** gyroX = messageData.readings.map(r => r.gyro.x);  **const** gyroY = messageData.readings.map(r => r.gyro.y);  **const** gyroZ = messageData.readings.map(r => r.gyro.z);  *// Compute mean and standard deviation over readings*  **const** inputData = {  acceleration\_x\_mean: computeMean(accelX).toFixed(6),  acceleration\_x\_std: computeStdDev(accelX).toFixed(6),  acceleration\_y\_mean: computeMean(accelY).toFixed(6),  acceleration\_y\_std: computeStdDev(accelY).toFixed(6),  acceleration\_z\_mean: computeMean(accelZ).toFixed(6),  acceleration\_z\_std: computeStdDev(accelZ).toFixed(6),  gyro\_x\_mean: computeMean(gyroX).toFixed(6),  gyro\_x\_std: computeStdDev(gyroX).toFixed(6),  gyro\_y\_mean: computeMean(gyroY).toFixed(6),  gyro\_y\_std: computeStdDev(gyroY).toFixed(6),  gyro\_z\_mean: computeMean(gyroZ).toFixed(6),  gyro\_z\_std: computeStdDev(gyroZ).toFixed(6)  };  *// Make the activity prediction*  **const** prediction = predict(inputData);  console.log('Activity data processed: ', prediction);  *// Insert the prediction into the database*  **const** timestamp = Date.now();  **await** db.collection('activity-data').add({  timestamp,  activity: prediction.label  });  *// Return a successful result*  res.status(200).send();  }; |
| --- |

There appears to be a lot going on here, so let’s break it down. As in the previous lab, we start with the configuration of the database - remember to input your own Project ID!

| *// 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'  }); |
| --- |

Next, we define some helper functions that are used to compute the mean and standard deviation of a dataset:

| *// Function to compute arithmetic mean over values in arr*  **function** computeMean(arr) {  **return** arr.reduce((a, b) => a + b) / arr.length;  }  *// Function to compute standard deviation over values in arr*  **function** computeStdDev(arr) {  **const** mean = computeMean(arr);  **const** variance = arr  .map(x => Math.pow(x - mean, 2))  .reduce((a, b) => a + b) / arr.length;  **return** Math.sqrt(variance);  } |
| --- |

computeMean uses the Array reduce function to sum together all the elements in the array, then divides that result by the length of the array - thus computing the arithmetic mean.

computeStdDev first computes the mean, using the computeMean helper function, then uses Array utility functions to compute the standard deviation. Math.sqrt computes the square root of a number, and Math.pow computes the number raised to a given power. If you take a look at the formula for computing the standard deviation, (and follow the code above) you should see how this implementation works.

After these helper methods, we have our predictor. If we were doing this with a machine learning engine, this function would pass the input data into the trained model, which would return a classification. But, since we’re simulating it, we’ll just make a guess based on the amount of standard deviation in the accelerometer values.

| *// Function to make an activity prediction, based on*  *// aggregated sensor data*  **function** predict(data) {  **if** (data.acceleration\_x\_std < 10 &&  data.acceleration\_y\_std < 10 &&  data.acceleration\_z\_std < 10) {  **return** {  label: 'still'  };  } **else** **if** (data.acceleration\_x\_std < 100 &&  data.acceleration\_y\_std < 100 &&  data.acceleration\_z\_std < 100) {  **return** {  label: 'walking'  };  } **else** {  **return** {  label: 'running'  };  }  } |
| --- |

If the standard deviations are less than 10, we’re probably still. If they are less than 100, we may be walking - otherwise we’re running. This approximation is by no means perfect, but it should illustrate the kind of processing you might need to do on data. Try fiddling around with the thresholds and seeing how they affect the classification.

If you have an interest in machine learning, feel free to extend this exercise to use a real machine learning model. There is a training data set available here, which could get you started:

<https://www.kaggle.com/vmalyi/run-or-walk>

Google Cloud offers various AI services (e.g. Vertex AI), but they generally require payment.

The next part of the Cloud Function is the entry point, which is run when the function is invoked. We’ll skip over the boiler plate, as we’ve already seen that in Lab 4.

This block of code extracts all of the components from the sensor readings into separate arrays. The embedded device will transmit a batch of five sensor readings, we need to split out the accelerometer and gyroscope X, Y, and Z components into separate arrays:

| *// Extract individual readings for accelerometer*  *// and gyro axes into arrays.*  **const** accelX = messageData.readings.map(r => r.accel.x);  **const** accelY = messageData.readings.map(r => r.accel.y);  **const** accelZ = messageData.readings.map(r => r.accel.z);  **const** gyroX = messageData.readings.map(r => r.gyro.x);  **const** gyroY = messageData.readings.map(r => r.gyro.y);  **const** gyroZ = messageData.readings.map(r => r.gyro.z); |
| --- |

This is so that we can then compute the mean and standard deviation for each:

| *// Compute mean and standard deviation over readings*  **const** inputData = {  acceleration\_x\_mean: computeMean(accelX).toFixed(6),  acceleration\_x\_std: computeStdDev(accelX).toFixed(6),  acceleration\_y\_mean: computeMean(accelY).toFixed(6),  acceleration\_y\_std: computeStdDev(accelY).toFixed(6),  acceleration\_z\_mean: computeMean(accelZ).toFixed(6),  acceleration\_z\_std: computeStdDev(accelZ).toFixed(6),  gyro\_x\_mean: computeMean(gyroX).toFixed(6),  gyro\_x\_std: computeStdDev(gyroX).toFixed(6),  gyro\_y\_mean: computeMean(gyroY).toFixed(6),  gyro\_y\_std: computeStdDev(gyroY).toFixed(6),  gyro\_z\_mean: computeMean(gyroZ).toFixed(6),  gyro\_z\_std: computeStdDev(gyroZ).toFixed(6)  }; |
| --- |

Here, we’re just using our helper functions to perform the computation, and storing the results in the object that we pass into the simulated predict function. At this point, we now have separate arrays for each of the sensor readings. Finally, the prediction is made, and the result stored in the database:

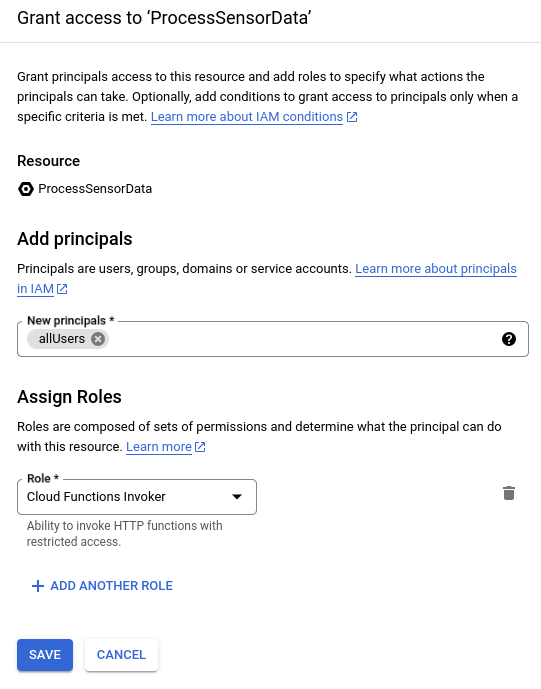
| *// Make the activity prediction*  **const** prediction = predict(inputData);  console.log('Activity data processed: ', prediction);  *// Insert the prediction into the database*  **const** timestamp = Date.now();  **await** db.collection('activity-data').add({  timestamp,  activity: prediction.label  }); |
| --- |

We now need to modify package.json to include a reference to the firestore package:

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

And, finally, upload your service account private key by creating a key.json file and inserting the contents from your downloaded private key into it.

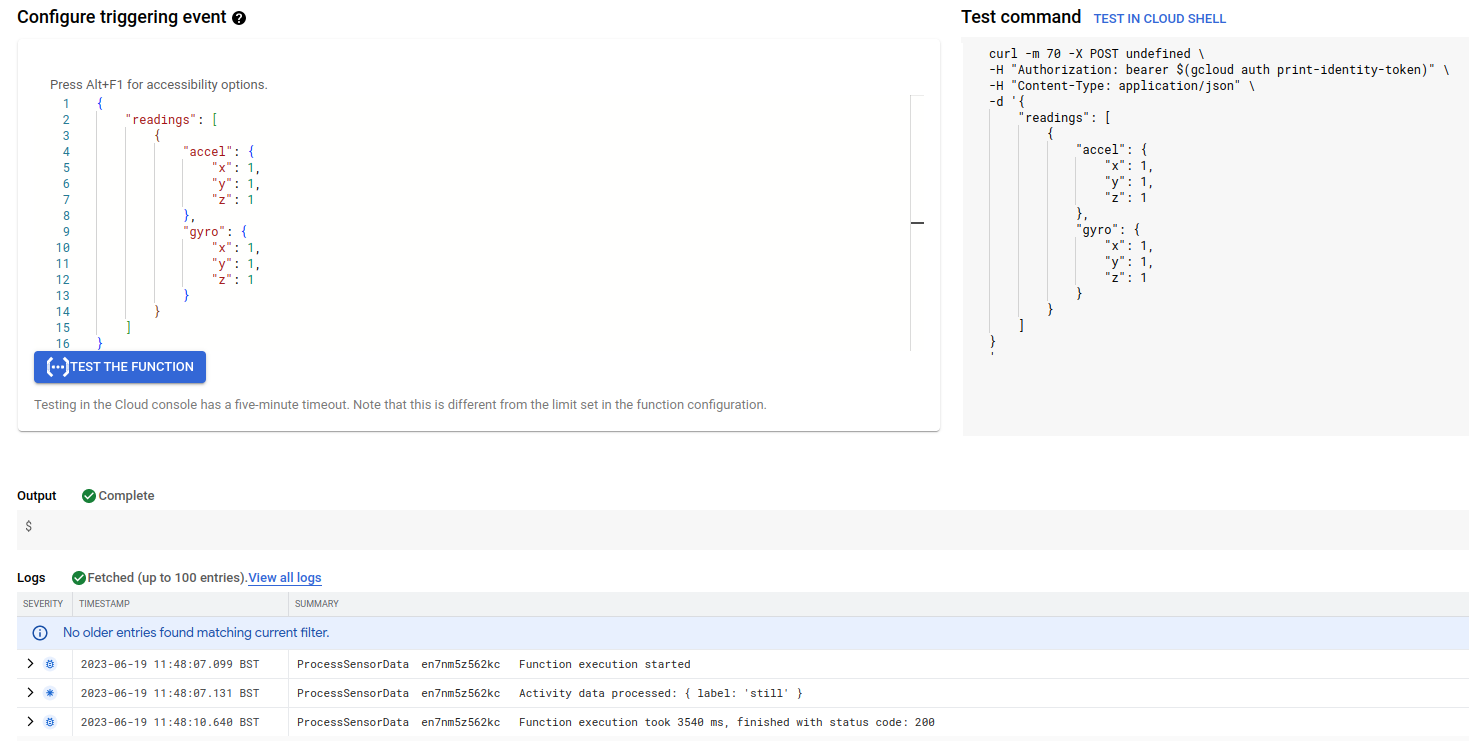
Once you’ve done all this, click “Deploy”, and wait for the green tick. Then, navigate to the “Permissions” tab, and add the “Cloud Functions Invoker” role for “allUsers”:



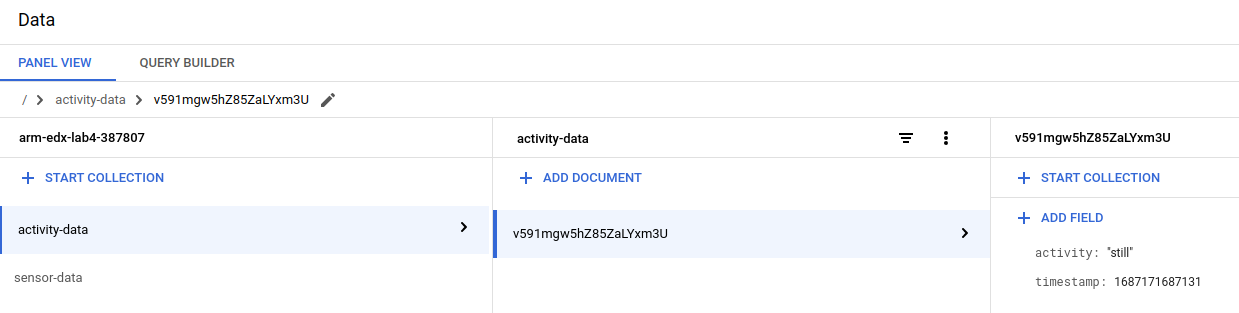
Now that permissions are set up, we can test this function. Navigate to the “Testing” tab, and put the following data into the test input section:

| {  "readings": [  {  "accel": {  "x": 1,  "y": 1,  "z": 1  },  "gyro": {  "x": 1,  "y": 1,  "z": 1  }  }  ]  } |
| --- |

Click “Test the function”, and after a few seconds you should see:

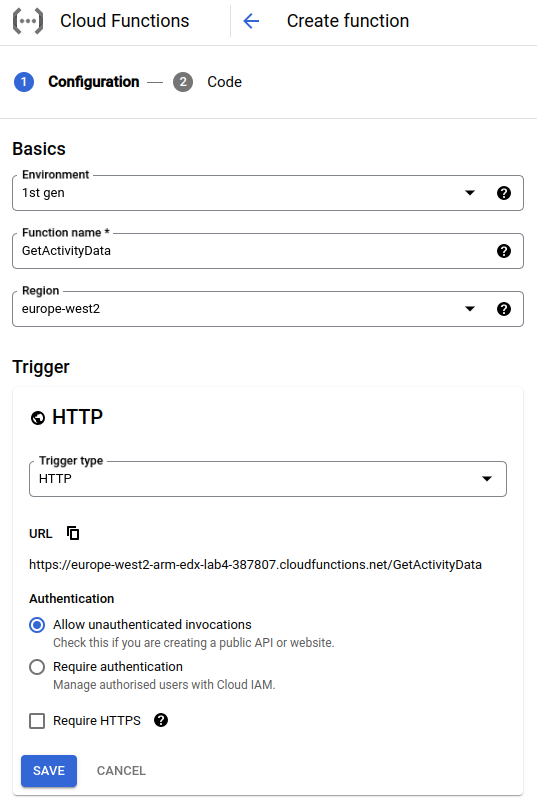


Here, in the log you should see “Activity data processed: { label: ‘still’ }”, which indicates that the classifier ran on the test data, and decided it represented “standing still”! Which makes sense, because we only sent one sensor reading. Importantly, however, navigate to the Firestore service, and check to make sure the classification was stored in the database:



## The “GetActivityData” Function

The next Cloud Function we’re going to build is the GetActivityData function, which will allow our mobile App to download the stored activity data. Navigate to the “Cloud Functions” service, and create a new function called “GetActivityData”:



Click “Save”, and in the code editor, specify “getData” as the entry point, and place the following code into index.js:

| *// 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 activity data from the DB.*  **const** coll = db.collection('activity-data');  **const** activityDataSnapshot = **await** coll.orderBy('timestamp').get();  **const** activityData = activityDataSnapshot.docs.map(doc => doc.data());  *// Return the activity data as a JSON object.*  res.json(activityData);  }  }; |
| --- |

| **What’s going on here?**  This function is quite straightforward - it is simply used as a mechanism for retrieving activity data by the mobile app. It works by accessing the database, ordering the entries by timestamp, and then sending the entries back as a JSON array.  The CORS logic is also present here, to make sure the function can be used from a different domain, i.e. from the Android App. |
| --- |

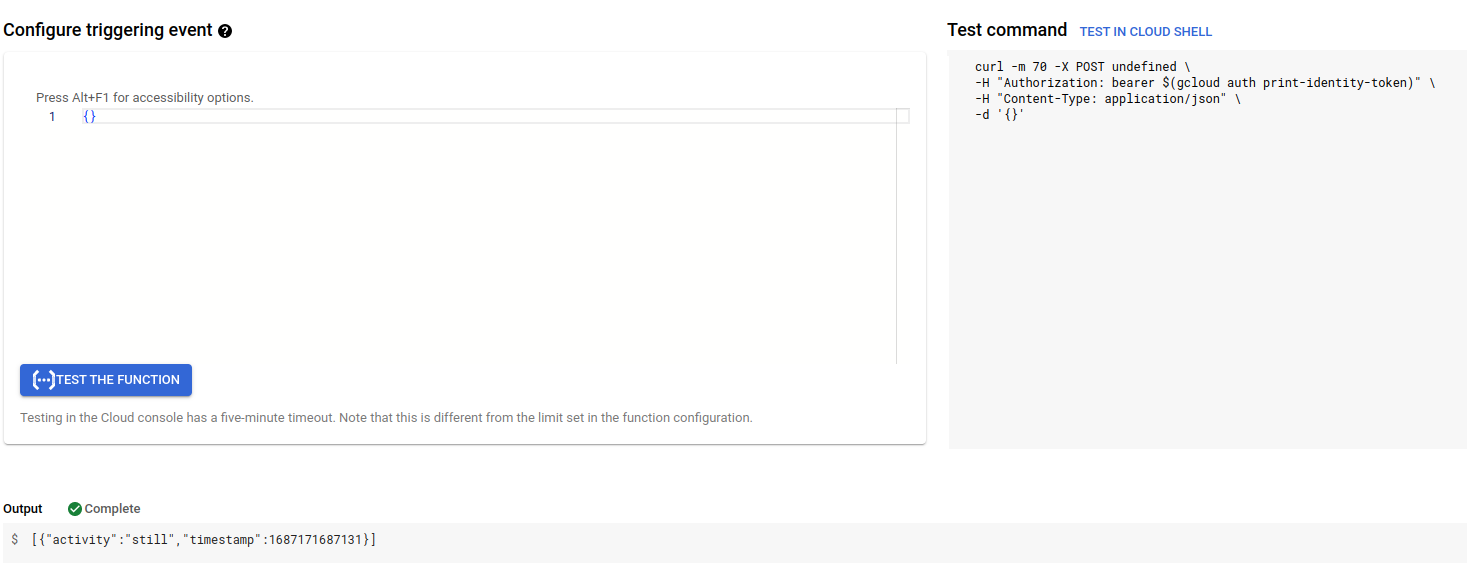
Now, update package.json to include a reference to the firestore package:

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

And, as before, upload your service account private key by creating a key.json file and inserting the contents from your downloaded key into it.

Once you’ve done all this, click “Deploy”, and wait for the green tick. Navigate to the “Permissions” tab, and add the “Cloud Functions Invoker” role for “allUsers” as in the previous function.

Navigate to the “Testing” tab, and click “Test the function”. This function does not require any input data, as it is purely a retrieval function:



In the “Output” section, you should see the test classification that you created when testing the ProcessSensorData function.

# Sending Gyro and Accelerometer data

Now we move on to the embedded device implementation! In Lab 4 we sent the sensor readings once every second, but in this lab we’re going to read gyro and accelerometer data, batch it up into blocks of five, and send the batches off to the Cloud Function. If you recall, the ProcessSensorData function classifies activity based on a batch of sensor readings.

We’ll be basing our device implementation on the Lab 4 work, so start by making a copy of your project, calling it e.g. “Lab 5”. Make sure the configuration in mbed\_app.json is up-to-date. You’ll also need to add the external library references:

* 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

Instead of the temperature, pressure, and humidity sensors, initialise the accelerometer and gyroscope sensors in the initialiseSensors function:

| **static** **void** initialiseSensors() {  BSP\_ACCELERO\_Init();  BSP\_GYRO\_Init();  } |
| --- |

Now, we’re going to introduce some helper routines:

| **struct** SensorData {  **int16\_t** accel[3];  **float** gyro[3];  };  **static** std::vector<SensorData> batchedSensorData;  **static** SensorData readSensors() {  printf("Reading sensors...\n");    SensorData d;  BSP\_ACCELERO\_AccGetXYZ(d.accel);  BSP\_GYRO\_GetXYZ(d.gyro);  **return** d;  }  **static** **void** sendReadings(NetworkInterface \*net) {  std::string messageData = "{ \"readings\": [";  **bool** first = true;  **for** (**const** **auto** &sd : batchedSensorData) {  **if** (first) {  first = false;  } **else** {  messageData += ", ";  }  messageData += "{ \"accel\": { ";  messageData += "\"x\": " + std::to\_string(sd.accel[0]) + ", ";  messageData += "\"y\": " + std::to\_string(sd.accel[1]) + ", ";  messageData += "\"z\": " + std::to\_string(sd.accel[2]) + " }, ";  messageData += "\"gyro\": { ";  messageData += "\"x\": " + std::to\_string(sd.gyro[0]) + ", ";  messageData += "\"y\": " + std::to\_string(sd.gyro[1]) + ", ";  messageData += "\"z\": " + std::to\_string(sd.gyro[2]) + " } }";  }  messageData += "] }";  **auto** \*req = **new** HttpRequest(  net, HTTP\_POST,  "**<YOUR-TRIGGER-URL>**");  req->set\_header("Content-Type", "application/json");  printf("Sending message: %s\n", messageData.c\_str());  HttpResponse \*res = req->send(messageData.c\_str(), messageData.length());  **if** (!res) {  printf("Http request failed (error code %d)\n", req->get\_error());  }  **delete** req;  } |
| --- |

You need to replace **<YOUR-TRIGGER-URL>** with the trigger URL of your ProcessSensorData Cloud Function, which you can get from the “Trigger” tab of the Function in the Google Cloud Console.

There’s quite a few things happening here, so let’s break it down. Here, we’re creating a custom data structure called “SensorData”, which we’re going to use to represent the readings from the sensors at a given point in time. Then, we create a “vector” (or a list) of these items, which will form our batch of five readings:

| **struct** SensorData {  **int16\_t** accel[3];  **float** gyro[3];  };  **static** std::vector<SensorData> batchedSensorData; |
| --- |

Next, we create a helper function called “readSensors”, which will take a reading from the accelerometer and gyro, and return it in the format of our custom data structure. Both the gyroscope and accelerometer produce three-axis values, so both of the fields are 3-element arrays. The gyroscope produces floating-point values, and the accelerometer produces integers, which is why we have two different data-types in the SensorData structure:

| **static** SensorData readSensors() {  printf("Reading sensors...\n");    SensorData d;  BSP\_ACCELERO\_AccGetXYZ(d.accel);  BSP\_GYRO\_GetXYZ(d.gyro);  **return** d;  } |
| --- |

Finally, the most complex-looking function is sendReadings. It might look daunting, but it’s quite straightforward. Its goal is to create and send the JSON message to the Cloud Function

| **static** **void** sendReadings(NetworkInterface \*net) {  std::string messageData = "{ \"readings\": [";  **bool** first = true;  **for** (**const** **auto** &sd : batchedSensorData) {  **if** (first) {  first = false;  } **else** {  messageData += ", ";  }  messageData += "{ \"accel\": { ";  messageData += "\"x\": " + std::to\_string(sd.accel[0]) + ", ";  messageData += "\"y\": " + std::to\_string(sd.accel[1]) + ", ";  messageData += "\"z\": " + std::to\_string(sd.accel[2]) + " }, ";  messageData += "\"gyro\": { ";  messageData += "\"x\": " + std::to\_string(sd.gyro[0]) + ", ";  messageData += "\"y\": " + std::to\_string(sd.gyro[1]) + ", ";  messageData += "\"z\": " + std::to\_string(sd.gyro[2]) + " } }";  }  messageData += "] }";  **auto** \*req = **new** HttpRequest(  net, HTTP\_POST,  "**<YOUR-TRIGGER-URL>**");  req->set\_header("Content-Type", "application/json");  printf("Sending message: %s\n", messageData.c\_str());  HttpResponse \*res = req->send(messageData.c\_str(), messageData.length());  **if** (!res) {  printf("Http request failed (error code %d)\n", req->get\_error());  }  **delete** req;  } |
| --- |

It does this through some string manipulation, by iterating over each sensor reading in the batchedSensorData list - that’s the “**for**” loop. For each item, it converts the sensor data into the appropriate JSON format, and adds it to the message. Finally, the function finishes up the message (adds the closing brackets!), and sends the JSON request off to the Cloud.

Now that we have these helpers in place, we can rewrite the sendSensorData function, to batch up data:

| **static** **void** sendSensorData(NetworkInterface \*net) {  **auto** sensorData = readSensors();  batchedSensorData.push\_back(sensorData);  **if** (batchedSensorData.size() >= 5) {  sendReadings(net);  batchedSensorData.clear();  }  } |
| --- |

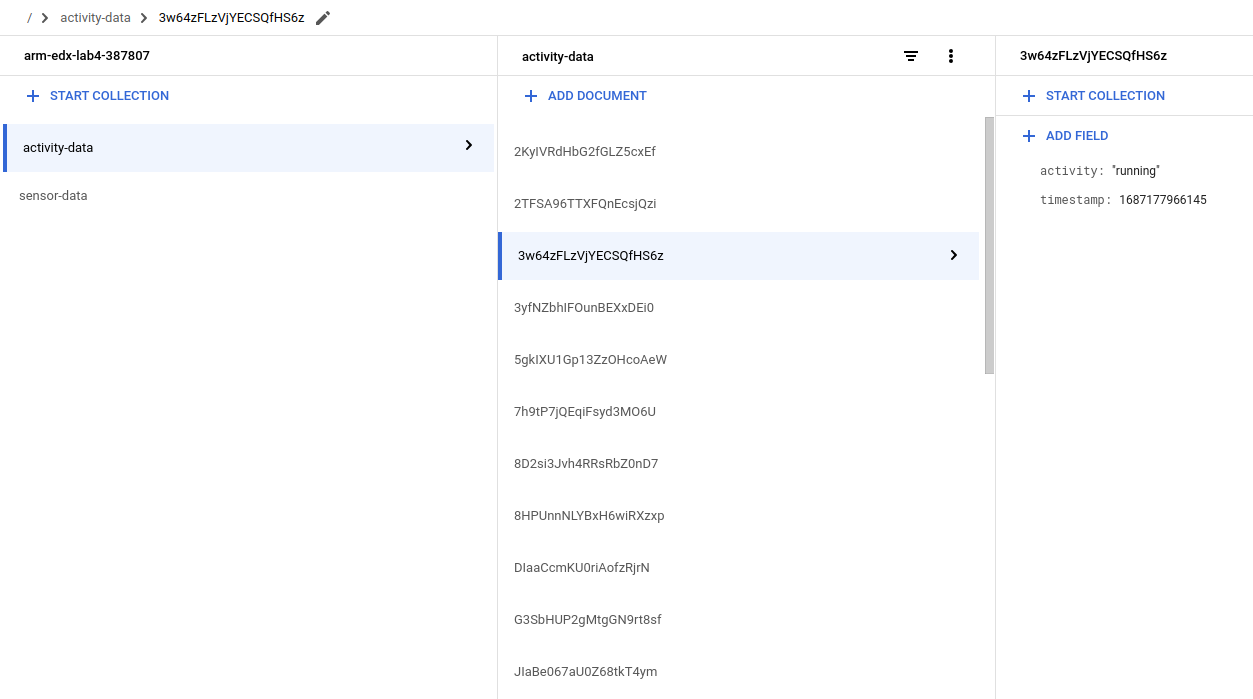
This function is quite simple now - we call “readSensors” to take a sensor reading, then add it to the batchedSensorData list. If the number of elements in the list is at least 5, call “sendReadings” and clear the batchedSensorData list, i.e. remove all readings we’ve just sent to the Cloud.

The main function can remain unchanged from Lab 4:

| **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);  }  } |
| --- |

So, to summarise, every second a sensor reading is taken. Then, after five consecutive readings (i.e. five seconds), the most recent sensor readings (since the last transmission) will be sent to the cloud service. This batching of sensor values helps to not overload the cloud systems -- there’s no need to continually send out data -- it’s perfectly fine to batch it together and send it in one go.

At this point, you should be able to build and run the code on the embedded device, and see the database being populated with activity classifications!



# 

# Viewing Activity History

Now that our embedded device is sending sensor readings to the cloud, and they are being processed and stored in the database, we can work on the final part of this lab - the user interface, through the Android app.

# Building the App

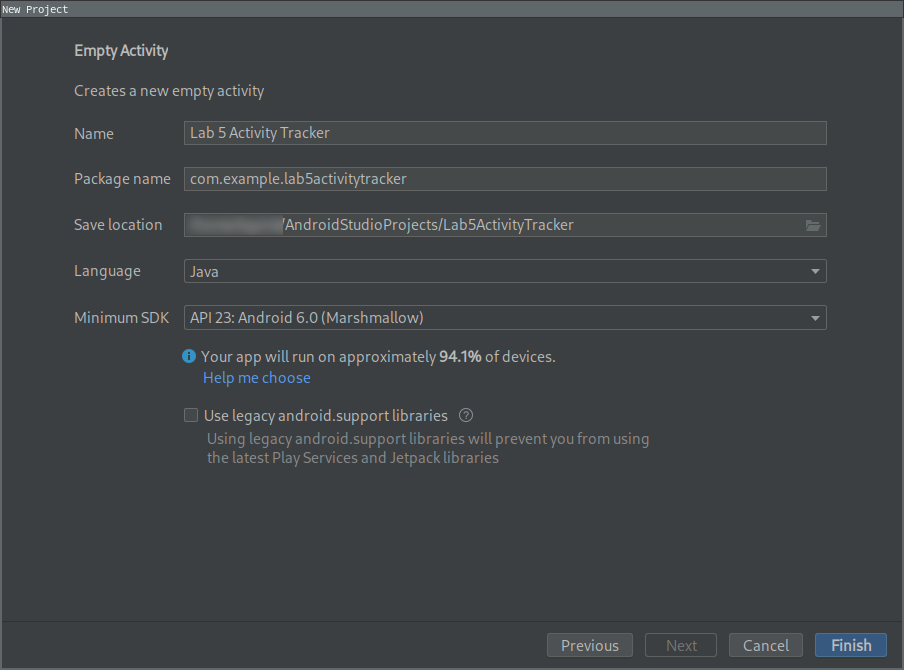
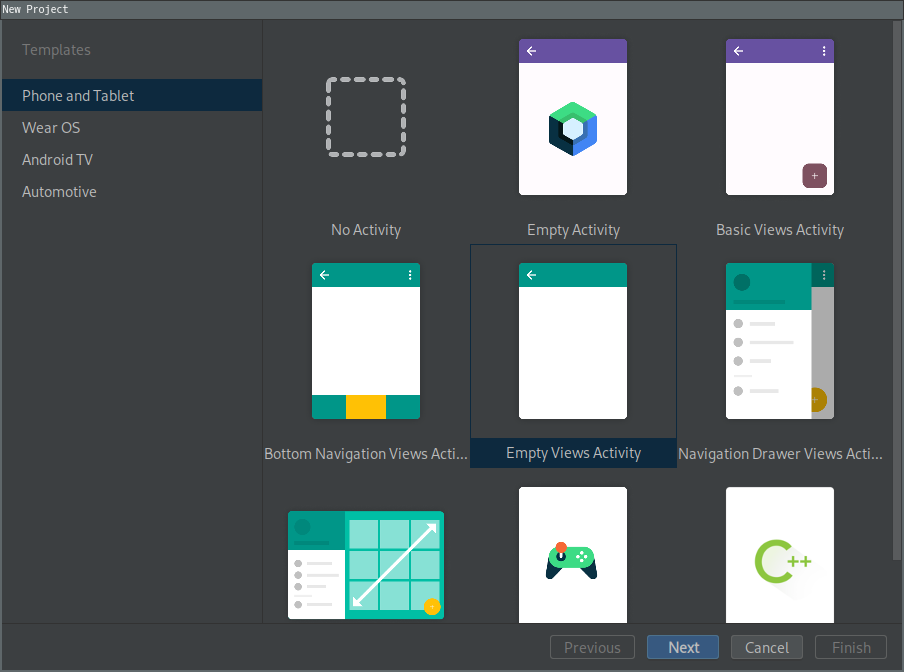
## Getting started

Building an entire mobile application can be a complex, daunting, and challenging task, so we’re only going to touch on the basics of Android development. If you feel comfortable - or adventurous - however, feel free to explore how you can turn this basic App into something better!

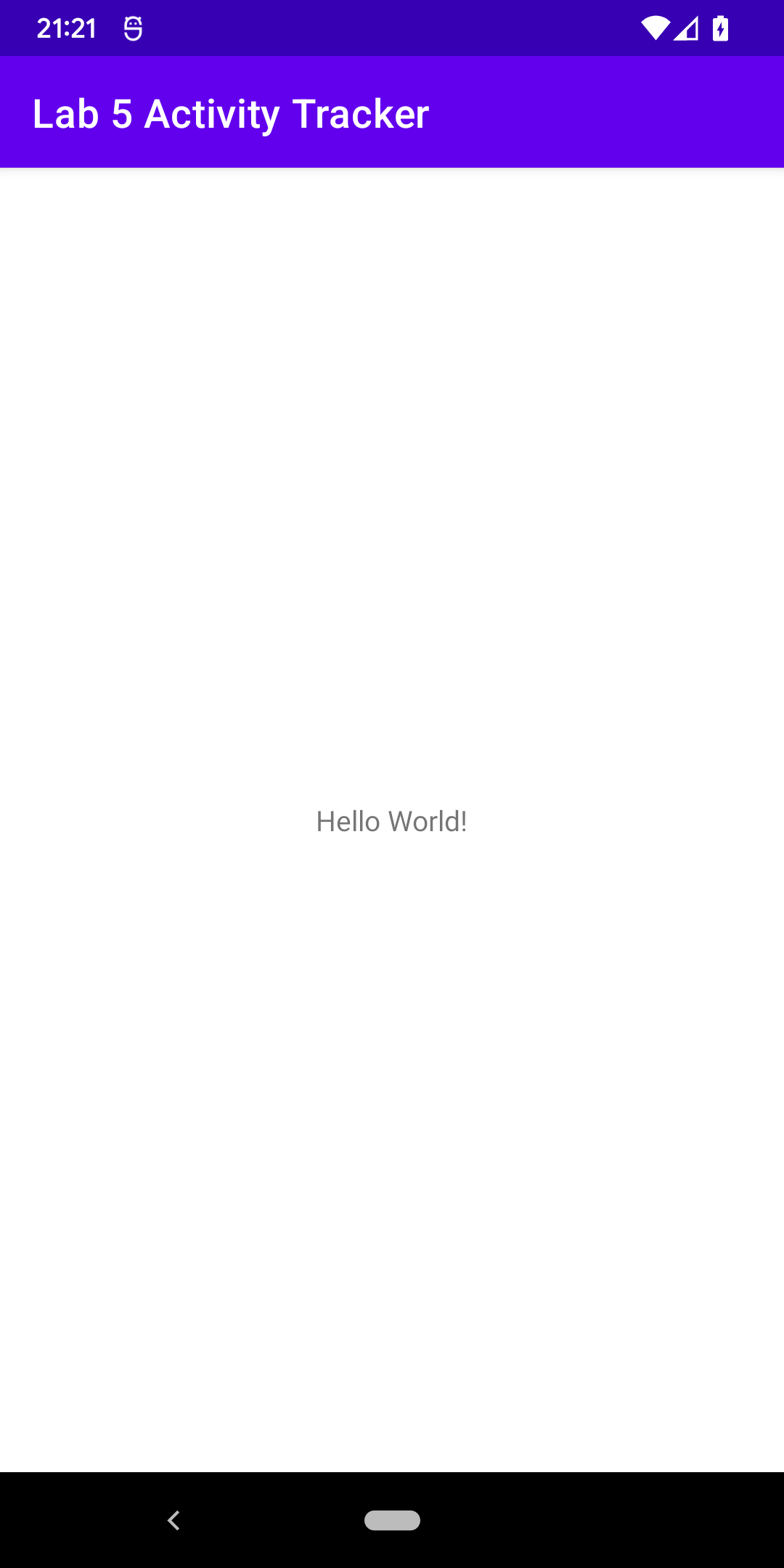
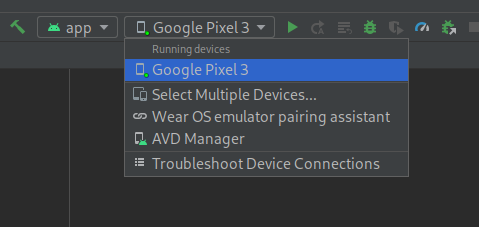
To get started with Android development, you will need an Android phone that’s in developer mode (details available online on how to do this for your specific model), and an installation of the latest version of Android Studio. Android Studio is available here:

<https://developer.android.com/studio>

Once you have installed Android Studio, open it up, and create a new project - choosing an “Empty Views Activity” (don’t confuse this with “Empty Activity”, which uses the Kotlin language) and giving the project a suitable name. Make sure you choose “Java” for the Language:



Once the project is created, plug in your development mobile device and check that you can successfully test and debug the application. Your device should appear in the list here (if not, follow the troubleshooting guide), and when selected, click the Play button, and after a few moments the empty App will appear on your mobile device:



When this is working, we’re ready to start developing.

## Overview

This App will provide a switch, which when turned on will interrogate the Cloud function that we previously built every five seconds, and request the data. It will then build a list of *“activity periods”* which is the duration of a particular activity, e.g. walking or running.

These *“activity periods”* will be shown in a table on the main screen.

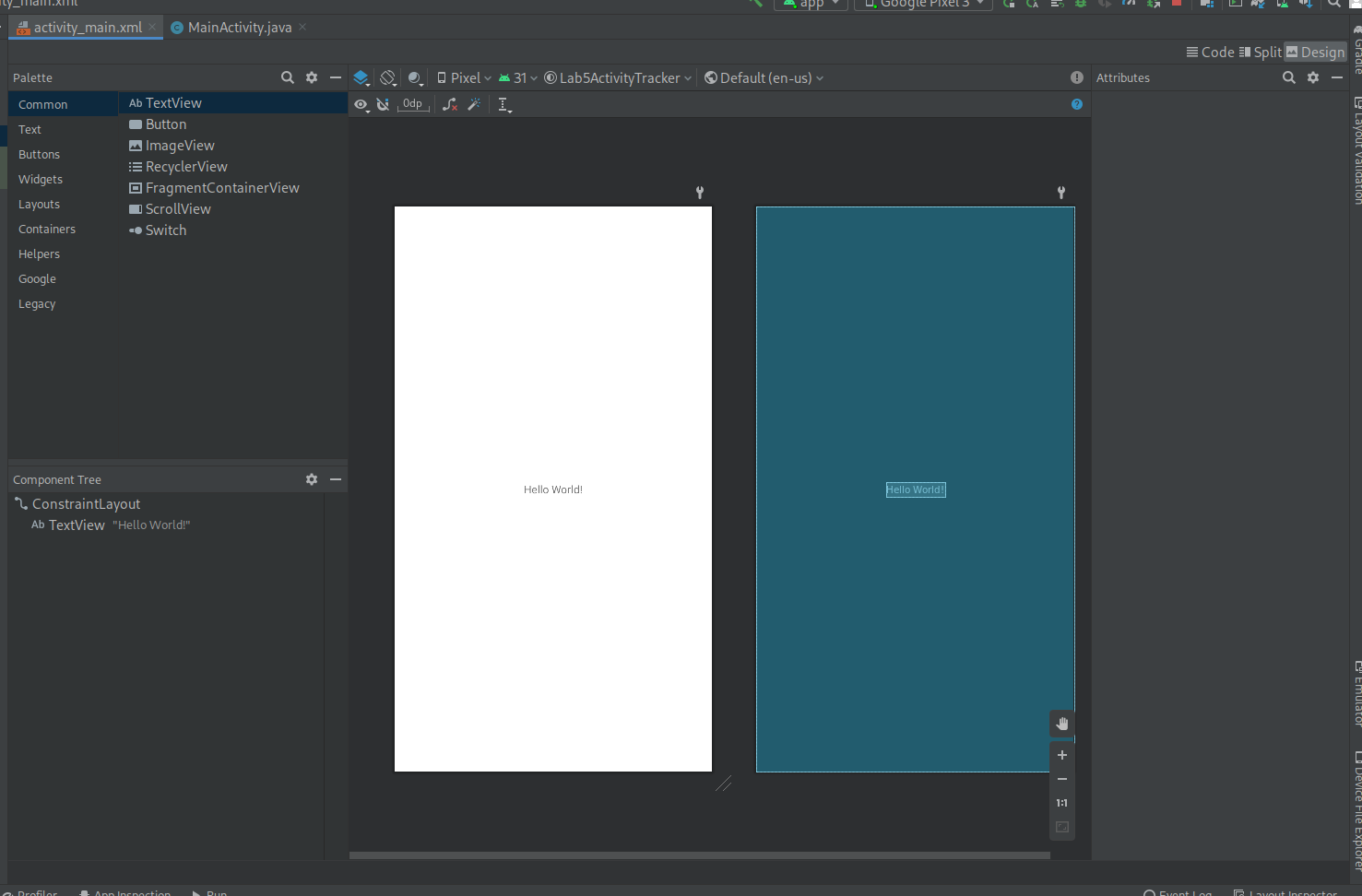
## Designing the User Interface

The first thing we will do is to create the user interface (UI). As mentioned earlier, you’re free to be as creative as you want here - provided you understand how different parts of the UI fit together, and how they will be later referenced in the code.

Android UIs are hierarchical, in that you have components within containers, which can also be in containers. This hierarchy can be seen in the UI designer. Our UI will comprise the following hierarchy:

* Main Layout
  + Progress Indicator
  + Scrollable Layout
    - Results Table
  + Read Activities Timer Switch

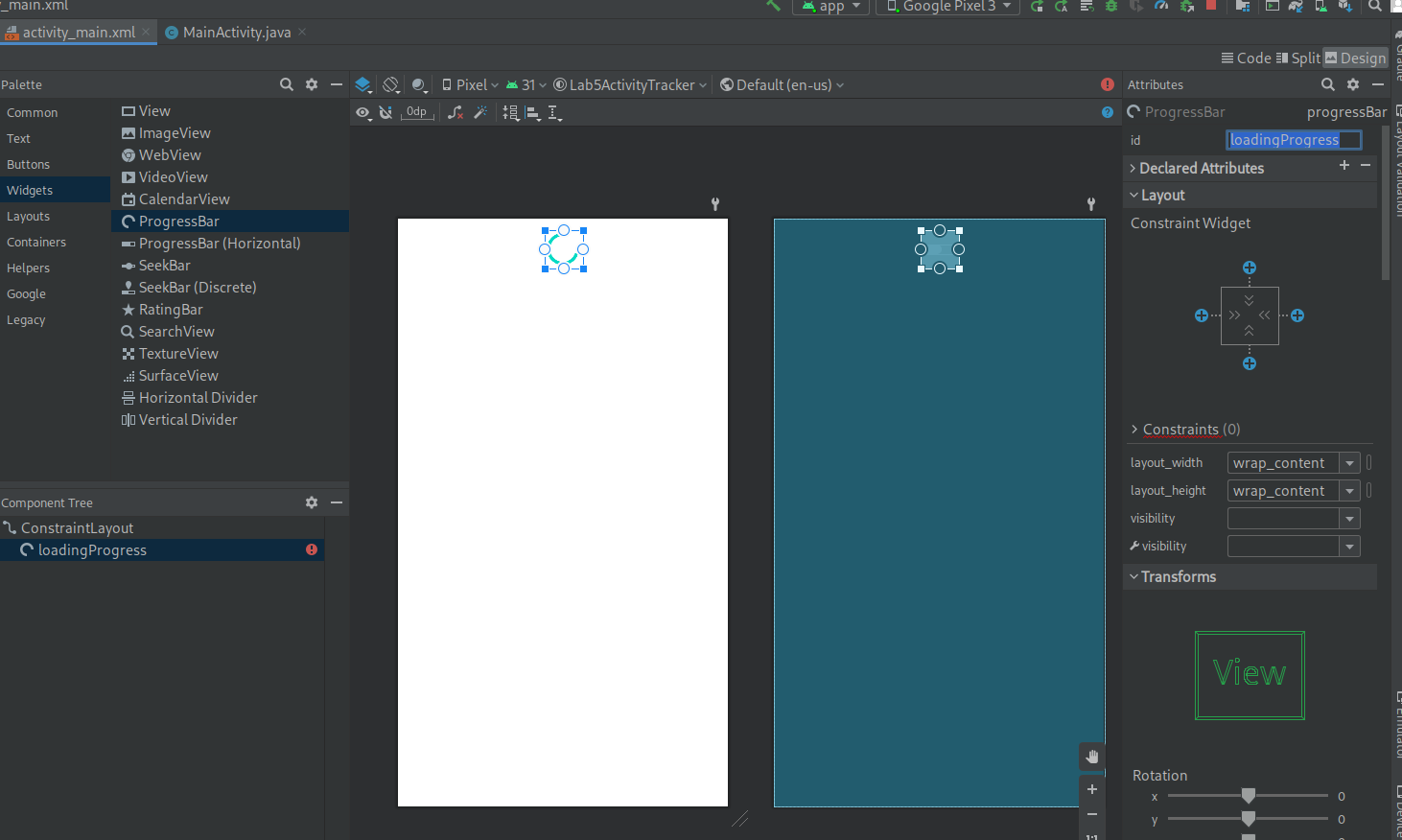
Start by opening up the activity\_main.xml file in Android Studio, and you should be presented with the UI designer:



Here, at the top-left you can see a list of components that can be used in the designer. The bottom left shows the component hierarchy, and is very useful for making sure everything is structured correctly. The central main pane is the designer interface, and is where we’ll be adding components.

First, delete the existing “Hello World!” text label (click it, and press delete), and insert a progress indicator, by navigating to the “Widgets” section in the palette, then dragging the “Progress Bar” component onto the design surface.

All components need a unique identifier, and will be given one by default. To make it clearer what this is referring to, change the component’s ID, by looking in the “Attributes” pane on the right, and changing the “id” property to “loadingProgress”.

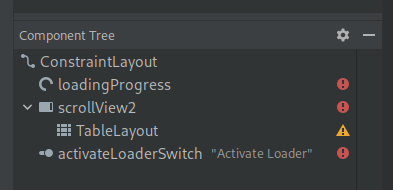


Next, navigate to the “Containers” section of the Palette, and drag in a ScrollView component. Once added, it should fill the remaining available area. Inside the ScrollView is a LinearLayout component that was automatically added. Delete this by expanding the ScrollView component in the component tree, selecting LinearLayout, and pressing delete.

Navigate to the “Layouts” section of the Palette, and drag a TableLayout component into the middle of the ScrollView component. This should appear as a child of the ScrollView component in the Component Tree. Give this an ID: “resultsTable”. Expand the TableLayout, and delete all of the automatically created TableRow components.

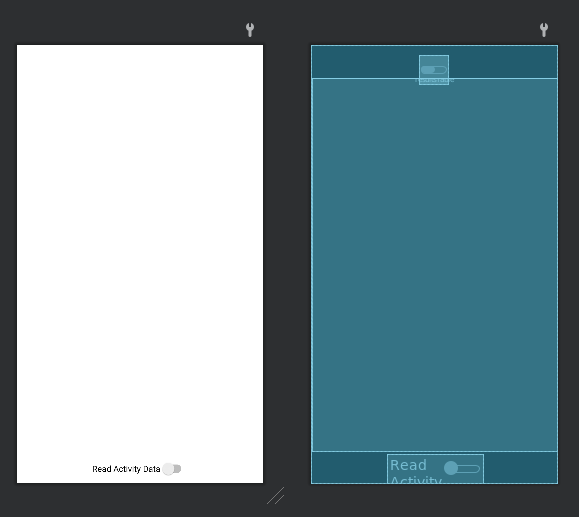
Finally, navigate to the “Buttons” section of the Palette, and add a Switch component. You may have to drag this directly into the ConstraintLayout component, so that it doesn’t become a child of the ScrollView. Give the switch an ID: “activateLoaderSwitch”, and give it some descriptive text, such as “Activate Loader”.

At this point, your component tree should look like this:



But, the layout won’t look very good. The ConstraintLayout is a way of laying out components so that they sit relative to each other, and we’ll be using this concept to position everything nicely.

The first thing to do is position the switch, by dragging it into a central position, and then constraining the bottom of it to the parent. Do this by dragging the bottom anchor point downwards. Next, position the ScrollView by dragging its bottom anchor point, to the top anchor point of the switch, so that it sits in between the ProgressBar and the Switch. If you’ve done this, your layout should look like this:



As you can see in the image on the right, the progress bar is at the top, the scroll view is in the middle, and the “Read Activity Data” switch is at the bottom.

Give your app a try, by clicking the play button at the top:

If everything worked, you should see the progress bar spinning at the top, a blank area in the middle, and a toggle switch (which changes state when you tap it) at the bottom.

The design on the user interface is now complete, and we can move to the coding.

## Reading and processing Activity Data

We can now start writing code to read and process activity data. There is quite a bit of code to get this working, so we’ll build it up and test it as we go. Open up MainActivity.java to see where we will be writing the code.

In this file already is a class that represents the activity (Android parlance for “view” or “screen”) of the App we’ve just designed. It contains a method onCreate, which is invoked as the view is being created, and before it is shown to the user.

The first thing to take care of is getting hold of references to the user interface elements that we want to alter:

| **public** **class** MainActivity **extends** AppCompatActivity {  **private** Switch activateLoaderSwitch;  **private** View loadingIndicator;  **private** TableLayout resultsTable;  *@Override*  **protected** **void** onCreate(Bundle savedInstanceState) {  **super**.onCreate(savedInstanceState);  setContentView(R.layout.activity\_main);  loadingIndicator = findViewById(R.id.loadingProgress);  loadingIndicator.setVisibility(View.INVISIBLE);  activateLoaderSwitch = findViewById(R.id.activateLoaderSwitch);  resultsTable = findViewById(R.id.resultsTable);  }  } |
| --- |

As you enter code in, the IDE may start complaining about missing classes -- but it will help you. If you see an error, navigate to the error, and it will suggest a fix. The fix is almost always, “import class” - and the IDE will automatically insert import statements at the top of the source code for you.

Here, we’ve created three fields:

* activateLoaderSwitch: for referencing the activateLoaderSwitch UI element
* loadingIndicator: for referencing the progress bar
* resultsTable: for referencing the table that will show our activities

And, in the onCreate method, we’ve gotten references to those elements by their ID. We’ve also hidden the progress bar, by setting its visibility property to INVISIBLE.

The next thing we’re going to do is to detect when readActivitiesSwitch has changed, i.e. the user has either turned it on or off. To do this, we need to register a checked change event listener, which involves adding an interface implementation to the class:

| **public** **class** MainActivity **extends** AppCompatActivity **implements** CompoundButton.OnCheckedChangeListener { |
| --- |

And then adding a new method, after onCreate:

| *@Override*  **public void** onCheckedChanged(CompoundButton buttonView, **boolean** isChecked) {  **if** (buttonView == activateLoaderSwitch) {  **if** (isChecked) {  // When the switch is turned ON  } **else** {  // When the switch is turned OFF  }  }  } |
| --- |

Now things start to get interesting. When the read activities switch is turned on, we want to schedule an action to occur every five seconds: namely, make a web request to the Cloud function, and process the results.

To do this, we need to use a Timer object. Add a new field in the class definition:

| **private** Timer readerTimer; |
| --- |

And update the onCheckedChanged method:

| **if** (isChecked) {  readerTimer = **new** Timer();  readerTimer.scheduleAtFixedRate(**new** TimerTask() {  *@Override*  **public** **void** run() {  // Invoked every FIVE seconds  }  }, 0, 5000);  } **else** {  readerTimer.cancel();  readerTimer = **null**;  } |
| --- |

What’s happening here is when the switch is turned on, a new Timer object is created, and a task is scheduled to run immediately, and then every 5000 ms (i.e. 5 seconds) after that. If the switch is turned off, the timer is cancelled and the object reference disposed.

We’ll now turn our attention to what action we’re going to perform when the timer elapses. To do this, we need to use something called a Handler, because we need to make sure the timer action is performed on the same thread as the user interface. It’s not possible to change UI elements from other threads, as this would make thread synchronisation very tricky, so instead we use Handlers to perform actions on the UI thread.

Add two new fields in the class definition:

| **private** Handler doLoadDataHandler;  **private** **boolean** loading; |
| --- |

Then, in the onCreate method, add the following code:

| MainActivity outer = **this**;  doLoadDataHandler = **new** Handler(**this**.getMainLooper()) {  *@Override*  **public** **void** handleMessage(Message msg) {  outer.doLoadData();  }  }; |
| --- |

What’s happening here is that we’re creating a new Handler object, and storing it in the doLoadDataHandler field. This handler simply calls another method doLoadData, on the main class. This is for convenience, so we can keep our processing logic together, and out of the way of everything else. We also need to add the doLoadData method to the class:

| **private** **void** doLoadData() {  **if** (**this**.loading) {  **return**;  }  **this**.loading = **true**;  loadingIndicator.setVisibility(View.VISIBLE);  } |
| --- |

Finally, we need to invoke the handler in the timer function:

| readerTimer.scheduleAtFixedRate(**new** TimerTask() {  *@Override*  **public** **void** run() {  doLoadDataHandler.obtainMessage().sendToTarget();  }  }, 0, 5000); |
| --- |

What’s exciting now is that we’re at a stage where we can perform a quick test of our App, to ensure things are working up to this point. When you start your App (click Play), you should be able to click on the switch at the bottom, and the loading indicator will be displayed.

We haven’t written any other logic at the moment, so the indicator will not disappear - but it should be encouraging that we’ve gotten this far.

To actually read the data from the Cloud, we need to add another field to the class definition:

| **private** RequestQueue requestQueue; |
| --- |

A request queue is a way to dispatch an HTTP request, and wait for the response without blocking any threads. Add the following line to the onCreate method:

| requestQueue = Volley.newRequestQueue(**this**); |
| --- |

And, the request queue is ready to use. The IDE will at first complain about “Volley”, but it will suggest to include a reference to the library automatically for you.

We also need to add two more interface implementations to the class definition:

| **public** **class** MainActivity **extends** AppCompatActivity **implements** CompoundButton.OnCheckedChangeListener, Response.Listener<String>, Response.ErrorListener { |
| --- |

And then add corresponding methods:

| *@Override*  **public** **void** onResponse(String response) {  System.out.println(response);  loadingIndicator.setVisibility(View.INVISIBLE);  **this**.loading = **false**;  }  *@Override*  **public** **void** onErrorResponse(VolleyError error) {  System.out.println(error.getMessage());  loadingIndicator.setVisibility(View.INVISIBLE);  **this**.loading = **false**;  } |
| --- |

We can also add the following lines to the end of doLoadData, which will trigger the request:

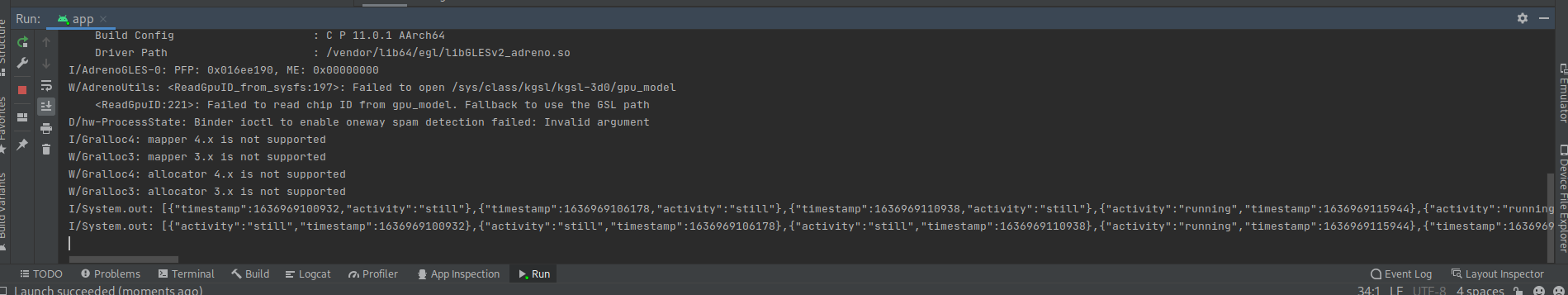
| String url = "**URL\_TO\_CLOUD\_FUNCTION**";  **this**.requestQueue.add(**new** StringRequest(Request.Method.GET, url, this, this)); |
| --- |

You will need to fill in the URL string, with the correct path to the “GetActivityData” Cloud Function we created previously. You can get this from the Google Cloud Console, when you go into the function view, and navigate to the “Trigger” tab.

There is now enough code to debug calls to the cloud function, but we need to give the App permission to access the internet before we can test it. To do this, open the manifests/AndroidManifest.xml file, and add the following element, before </manifest> at the bottom:

| <uses-permission android:name="android.permission.INTERNET" /> |
| --- |

Now, click the Play button, and toggle the switch. If you click on the “Run” tab at the bottom of the IDE, you will see debug messages appear, which should (hopefully) contain the result of calling the cloud function:



The last piece of the puzzle is to parse the resulting data, and display it in the table.

Our table is going to look like this:

| Activity Start | Activity End | Activity Type |
| --- | --- | --- |

With one row for each distinct activity. Recall, however, that our cloud function returns classifications of batches of sensor readings, at a given point in time. So, what this means is, if there are two adjacent activities in the results that are the same type, we will merge them into one. For example, if the Cloud function returns the following:

| 10:00 | Walking |
| --- | --- |
| 10:05 | Walking |
| 10:15 | Running |
| 10:20 | Walking |
| 10:25 | Running |
| 10:30 | Running |

Then we will display this:

| **Activity Start** | **Activity End** | **Activity Type** |
| --- | --- | --- |
| 10:00 | 10:15 | Walking |
| 10:15 | 10:20 | Running |
| 10:20 | 10:25 | Walking |
| 10:25 | Now | Running |

We’ll need a helper function to add rows to the table, so add in the following:

| **private** **void** appendActivityPeriod(Date periodStart, Date periodEnd, String activity) {  TableRow row = **new** TableRow(**this**.getBaseContext());  DateFormat fmt = DateFormat.getDateTimeInstance();  TextView startTimestamp = **new** TextView(**this**.getBaseContext());  startTimestamp.setText(fmt.format(periodStart));  row.addView(startTimestamp);  Space s = **new** Space(**this**.getBaseContext());  s.setMinimumWidth(32);  row.addView(s);  TextView endTimestamp = **new** TextView(**this**.getBaseContext());  endTimestamp.setText(fmt.format(periodEnd));  row.addView(endTimestamp);  Space s2 = **new** Space(**this**.getBaseContext());  s2.setMinimumWidth(32);  row.addView(s2);  TextView activityText = **new** TextView(**this**.getBaseContext());  activityText.setText(activity);  row.addView(activityText);  resultsTable.addView(row);  } |
| --- |

This might look complicated, but it’s quite straightforward. The function takes three parameters (periodStart, periodEnd, and activity), which are the three cells that will be added as one row to the table.

A new row object is created, and a DateFormat object is created for displaying date strings. Then, each cell is created (via a TextView object), and added to the new row, with a spacer between them.

Now, replace the onResponse method with the following:

| @Override  **public** **void** onResponse(String response) {  **try** {  resultsTable.removeAllViews();  JSONArray activities = **new** JSONArray(response);  JSONObject lastActivity = null;  Date activityPeriodStart = null;  **for** (**int** i = 0; i < activities.length(); i++) {  JSONObject activity = activities.getJSONObject(i);  System.out.println(activity);  // Check to see if there was a previous activity  **if** (lastActivity != null) {  // Now, compare the activity types  **if** (!lastActivity.getString("activity").equalsIgnoreCase(activity.getString("activity"))) {  // If the activity types are different, then terminate the current activity period,  // and start a new one.  appendActivityPeriod(activityPeriodStart, **new** Date(activity.getLong("timestamp")), lastActivity.getString("activity"));  activityPeriodStart = **new** Date(activity.getLong("timestamp"));  }  } **else** {  // There was no previous activity, so start a new activity period.  activityPeriodStart = **new** Date(activity.getLong("timestamp"));  }  lastActivity = activity;  }  } **catch** (JSONException e) {  e.printStackTrace();  } **finally** {  loadingIndicator.setVisibility(View.INVISIBLE);  **this**.loading = **false**;  }  } |
| --- |

This function is a bit more involved, so let’s break it down.

First, we’re going to look at the high-level structure of this function, which comprises a try...catch...finally block. This construct allows us to detect errors. Inside the try block, we can run code that might produce an error - or more precisely, raise an exception. If the code does raise an exception, it immediately stops and jumps to the catch block (provided that the catch block is defined for the right type of exception). The finally block is run regardless of whether or not the code raised an exception, and is always executed last (i.e. after the try block finished successfully, or the catch block completed).

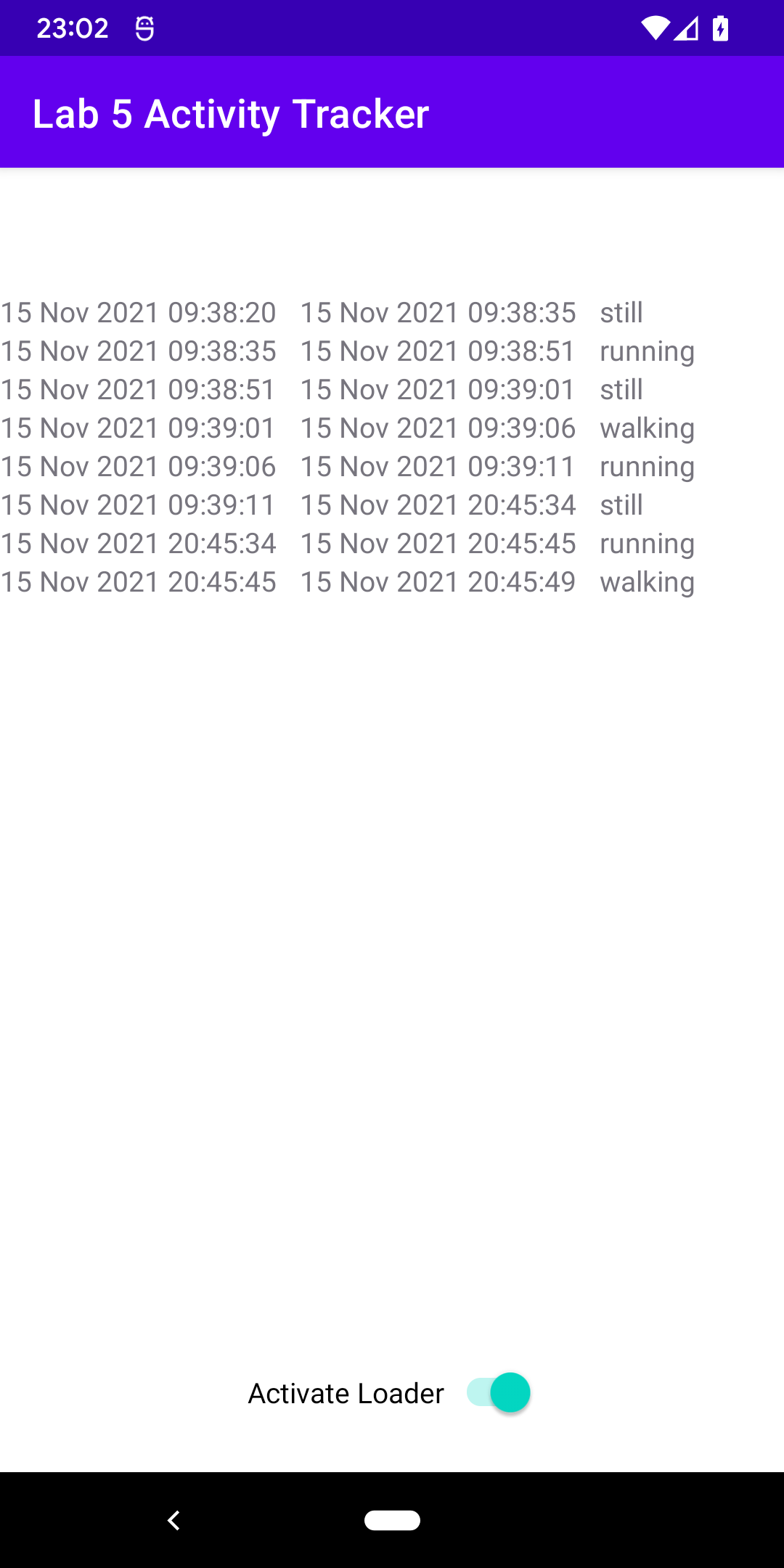
In this case, we encapsulate our main functionality in the try block, to detect errors. If we detected an error, then we print the error out to the console for debugging. In either case, in our finally block, we hide the loading progress bar, and clear the loading flag.

Now, inside the try block comes the main part of the implementation. First, any existing elements in the table are removed (resultsTable.removeAllViews();).

Next, we parse the returned data into a JSONArray structure, so that we can iterate over it in the for loop. For each activity that is returned to us by the Cloud, we check to see if it’s the same type as the one we’ve just seen (in which case we merge it), or if it’s different, then we start a new activity period.

When we’ve detected a complete activity period, we use our helper method (appendActivityPeriod) to add it to the table.

Try this out now, by clicking the Play button. Once the App has loaded, flick the switch, and wait for the data to come in. You should see the latest activity data sent by your device appear:



Now turn on your embedded device, and once it has connected to the Cloud, every five seconds you should see new data coming in, and being added to the end of the table. Make sure you simulate walking and running on your device, by giving it a shake!

## Reference Class and UI Implementation

If you get stuck, and haven’t been able to follow along, here is a full reference implementation of the activity class source code file (MainActivity.java)

| package com.example.lab5activitytracker;  import androidx.appcompat.app.AppCompatActivity;  import android.os.Bundle;  import android.os.Handler;  import android.os.Message;  import android.view.View;  import android.widget.CompoundButton;  import android.widget.Space;  import android.widget.Switch;  import android.widget.TableLayout;  import android.widget.TableRow;  import android.widget.TextView;  import com.android.volley.Request;  import com.android.volley.RequestQueue;  import com.android.volley.Response;  import com.android.volley.VolleyError;  import com.android.volley.toolbox.StringRequest;  import com.android.volley.toolbox.Volley;  import org.json.JSONArray;  import org.json.JSONException;  import org.json.JSONObject;  import java.text.DateFormat;  import java.util.Date;  import java.util.Timer;  import java.util.TimerTask;  public class MainActivity extends AppCompatActivity implements CompoundButton.OnCheckedChangeListener, Response.Listener<String>, Response.ErrorListener {  private Switch activateLoaderSwitch;  private View loadingIndicator;  private TableLayout resultsTable;  private Timer readerTimer;  private Handler doLoadDataHandler;  private boolean loading;  private RequestQueue requestQueue;  @Override  protected void onCreate(Bundle savedInstanceState) {  super.onCreate(savedInstanceState);  setContentView(R.layout.activity\_main);  loadingIndicator = findViewById(R.id.loadingProgress);  loadingIndicator.setVisibility(View.INVISIBLE);  activateLoaderSwitch = findViewById(R.id.activateLoaderSwitch);  resultsTable = findViewById(R.id.resultsTable);  activateLoaderSwitch.setOnCheckedChangeListener(this);  MainActivity outer = this;  doLoadDataHandler = new Handler(this.getMainLooper()) {  @Override  public void handleMessage(Message msg) {  outer.doLoadData();  }  };  requestQueue = Volley.newRequestQueue(this);  }  @Override  public void onCheckedChanged(CompoundButton buttonView, boolean isChecked) {  if (buttonView == activateLoaderSwitch) {  if (isChecked) {  readerTimer = new Timer();  readerTimer.scheduleAtFixedRate(new TimerTask() {  @Override  public void run() {  doLoadDataHandler.obtainMessage().sendToTarget();  }  }, 0, 5000);  } else {  readerTimer.cancel();  readerTimer = null;  }  }  }  private void doLoadData() {  if (this.loading) {  return;  }  this.loading = true;  loadingIndicator.setVisibility(View.VISIBLE);  String url = "YOUR\_CLOUD\_FUNCTION\_URL";  this.requestQueue.add(new StringRequest(Request.Method.GET, url, this, this));  }  @Override  public void onResponse(String response) {  try {  resultsTable.removeAllViews();  JSONArray activities = new JSONArray(response);  JSONObject lastActivity = null;  Date activityPeriodStart = null;  for (int i = 0; i < activities.length(); i++) {  JSONObject activity = activities.getJSONObject(i);  System.out.println(activity);  // Check to see if there was a previous activity  if (lastActivity != null) {  // Now, compare the activity types  if (!lastActivity.getString("activity").equalsIgnoreCase(activity.getString("activity"))) {  // If the activity types are different, then terminate the current activity period,  // and start a new one.  appendActivityPeriod(activityPeriodStart, new Date(activity.getLong("timestamp")), lastActivity.getString("activity"));  activityPeriodStart = new Date(activity.getLong("timestamp"));  }  } else {  // There was no previous activity, so start a new activity period.  activityPeriodStart = new Date(activity.getLong("timestamp"));  }  lastActivity = activity;  }  } catch (JSONException e) {  e.printStackTrace();  } finally {  loadingIndicator.setVisibility(View.INVISIBLE);  this.loading = false;  }  }  @Override  public void onErrorResponse(VolleyError error) {  System.out.println(error.getMessage());  loadingIndicator.setVisibility(View.INVISIBLE);  this.loading = false;  }  private void appendActivityPeriod(Date periodStart, Date periodEnd, String activity) {  TableRow row = new TableRow(this.getBaseContext());  DateFormat fmt = DateFormat.getDateTimeInstance();  TextView startTimestamp = new TextView(this.getBaseContext());  startTimestamp.setText(fmt.format(periodStart));  row.addView(startTimestamp);  Space s = new Space(this.getBaseContext());  s.setMinimumWidth(32);  row.addView(s);  TextView endTimestamp = new TextView(this.getBaseContext());  endTimestamp.setText(fmt.format(periodEnd));  row.addView(endTimestamp);  Space s2 = new Space(this.getBaseContext());  s2.setMinimumWidth(32);  row.addView(s2);  TextView activityText = new TextView(this.getBaseContext());  activityText.setText(activity);  row.addView(activityText);  resultsTable.addView(row);  }  } |
| --- |

And here is the UI definition file (activity\_main.xml):

| <?xml version="1.0" encoding="utf-8"?>  <androidx.constraintlayout.widget.ConstraintLayout xmlns:android="http://schemas.android.com/apk/res/android"  xmlns:app="http://schemas.android.com/apk/res-auto"  xmlns:tools="http://schemas.android.com/tools"  android:layout\_width="match\_parent"  android:layout\_height="match\_parent"  tools:context=".MainActivity">  <ProgressBar  android:id="@+id/loadingProgress"  style="?android:attr/progressBarStyle"  android:layout\_width="wrap\_content"  android:layout\_height="wrap\_content"  android:layout\_marginStart="182dp"  android:layout\_marginTop="14dp"  android:layout\_marginEnd="181dp"  app:layout\_constraintEnd\_toEndOf="parent"  app:layout\_constraintStart\_toStartOf="parent"  app:layout\_constraintTop\_toTopOf="parent" />  <ScrollView  android:id="@+id/scrollView2"  android:layout\_width="409dp"  android:layout\_height="667dp"  app:layout\_constraintBottom\_toTopOf="@+id/activateLoaderSwitch"  app:layout\_constraintTop\_toBottomOf="@+id/loadingProgress"  app:layout\_constraintVertical\_bias="0.0"  tools:layout\_editor\_absoluteX="1dp">  <TableLayout  android:id="@+id/resultsTable"  android:layout\_width="match\_parent"  android:layout\_height="wrap\_content" />  </ScrollView>  <Switch  android:id="@+id/activateLoaderSwitch"  android:layout\_width="wrap\_content"  android:layout\_height="wrap\_content"  android:layout\_marginStart="133dp"  android:layout\_marginEnd="133dp"  android:layout\_marginBottom="16dp"  android:minHeight="48dp"  android:text="Activate Loader"  app:layout\_constraintBottom\_toBottomOf="parent"  app:layout\_constraintEnd\_toEndOf="parent"  app:layout\_constraintStart\_toStartOf="parent" />  </androidx.constraintlayout.widget.ConstraintLayout> |
| --- |