

# Lab 1: Introduction to Mbed

Learning outcome: Understand the architecture of an embedded system, and how to build and run a program using Mbed OS

## Introduction

This lab will introduce the hardware platform that is going to be used for subsequent labs: the **ST DISCO L475VG**. It will also introduce the Arm Mbed development ecosystem, to get you familiar with the development process that we will be following in future labs.

## Resources

### Hardware

This lab is based around the **ST DISCO L475VG** development board:



We will refer to this as “the board” throughout the remainder of this, and future labs.

## Board Specification

This is the technical specification of the board, and describes the various hardware features present, and what their capabilities are:

- **Storage:**
  - 64-Mbit Quad-SPI (Macronix) Flash memory
- **Connectivity:**
  - Bluetooth® V4.1 module (SPBTLE-RF)
  - Sub-GHz (868 or 915MHz) low-power-programmable RF module (SPSGRF-868 or SPSGRF-915)
  - Wi-Fi® module Inventek ISM43362-M3G-L44 (802.11 b/g/n compliant)
  - Dynamic Near-field Communication (NFC) tag based on M24SR with its printed NFC antenna
- **Input/Output (IO):**

- Two digital omnidirectional microphones (MP34DT01)
- Capacitive digital sensor for relative humidity and temperature (HTS221)
- High-performance 3-axis magnetometer (LIS3MDL)
- 3D accelerometer and 3D gyroscope (LSM6DSL)
- 260-1260hPa absolute digital output barometer (LPS22HB)
- Time-of-Flight and gesture-detection sensor (VL53L0X)
- Two push-buttons (user and reset)

● **Communication:**

- USB OTG FS with Micro-AB connector

● **Expansion connectors:**

- *Arduino™ Uno V3*
- PMOD

● Flexible power-supply options: ST LINK USB VBUS or external sources

● On-board ST-LINK/V2-1 debugger/programmer with USB re-enumeration capability: mass storage, virtual COM port, and debug port

## Board Pinout


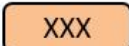





A “*pinout*” is a description of what each physical connection – or pin – does on the board. The following legend shows what the various colours mean, when interpreting the pin diagram.

A “*header*” is a term used to describe a set of pins that are physically grouped together.

### Labels usable in code

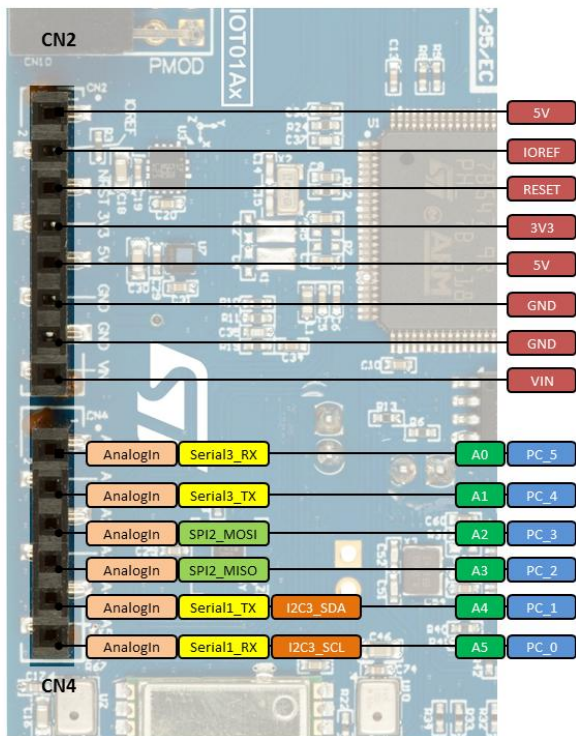
	MCU pin without conflict		Arduino connector names (A0, D1, ...)
	MCU pin connected to other components <i>See <a href="#">PeripheralPins.c</a> (link below) for more information</i>		LEDs and Buttons (LED_1, USER_BUTTON, ...)

### Labels not usable in code (for information only)

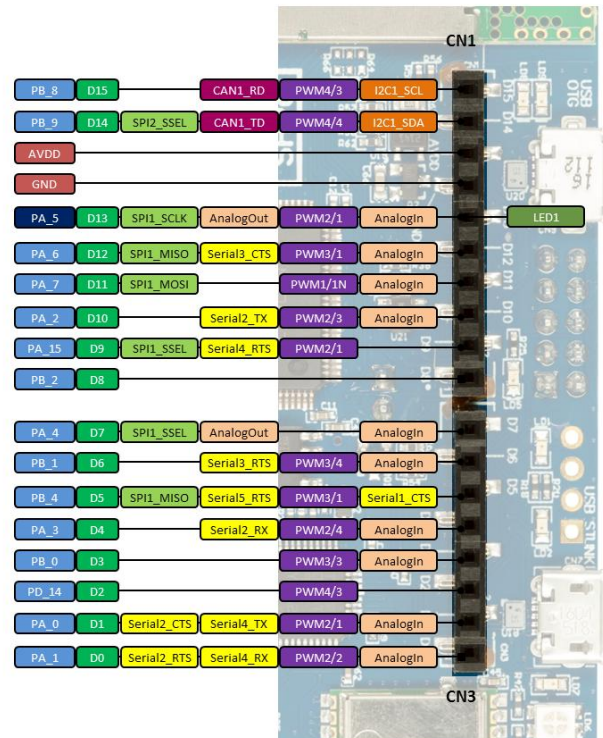
	Serial pins (USART/UART)		AnalogIn (ADC) and AnalogOut pins (DAC)
	SPI pins		CAN pins
	I2C pins		
	PWMOut pins (TIMER n/c[N]) n = Timer number c = Channel N = Inverted channel		Power and control pins (3V3, GND, RESET, ...)

These are the pin diagrams:

## Arduino Compatible Headers

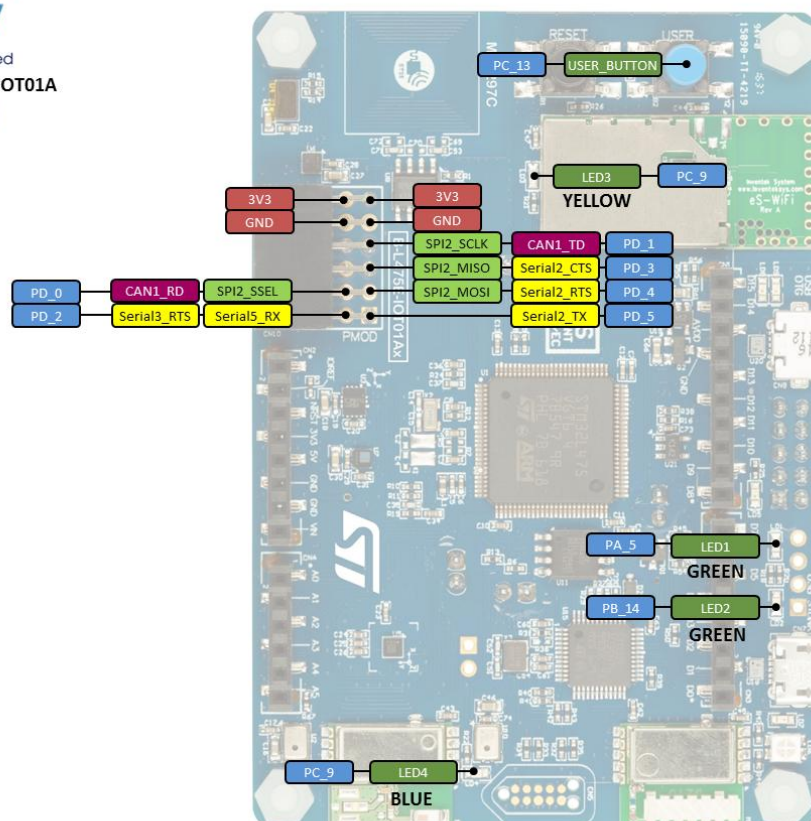


Left-hand Side



right-hand Side

R



Notice that some pins have more than one function. You can configure these pins (in software) to do different things, depending on your needs. Some of these additional functions are grouped together, and require taking over a set of pins – such as SPI or I2C.

For example, if you want to use one of the SPI busses, e.g. SPI2, then pins `PD_1`, `PD_3`, `PD_4`, and `PD_0` need to be reconfigured from GPIOs, to the SPI control lines: `SPI2_SCLK`, `SPI2_MISO`, `SPI2_MOSI`, and `SPI2_SSEL`.

## Software

You will need access to a computer, an internet connection, and you will be introduced to the Mbed Studio IDE. You will also learn about setting up a local development environment, if you prefer.

# Communicating with the board

The board is designed to appear as a *removable disk drive* (like a USB stick) on your computer, when it is plugged in over USB. Plugging in the board, and having this “virtual” disk drive appear is called *mounting*.

To program the board, you simply copy your compiled program into the virtual disk drive, and once the copy operation is complete, the board will be programmed.

**Try this now:** connect the board to your computer, using a USB cable, and see the drive appear.

You should see a drive appear called “*DAPLINK*” (or possibly the name of the board).

## NOTES FOR OPERATING SYSTEMS

- **Windows:** you’ ll find the virtual drive in “Computer”
- **Mac:** you’ ll find the virtual drive appear on your desktop
- **Linux:** the virtual drive may automatically mount for you, if this is configured on your system. Otherwise, you can issue the `mount` command to mount the drive manually. Find out the most recently recognised USB storage device, by looking at `dmesg` output.

# Mbed Studio Setup

Mbed Studio is a free IDE that allows developers to create Mbed OS 6 applications and libraries. This will provide the functionality we require for completing the further labs.

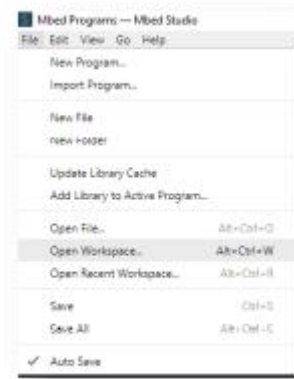
Ensure that you have installed Mbed Studio from the following link: <https://os.mbed.com/studio/>

You will also need to register for an account, and use it to sign in when launching the IDE.

## Preparing a workspace

Upon installation of Mbed Studio, a workspace named “Mbed Programs” is created for you in your home directory. A workspace is a location in the filesystem that contains your Mbed programs. This includes both imported and created projects.

You can change this workspace by clicking “File” and then selecting “Open Workspace”. Navigate to the folder you wish to make the new workspace and select it.

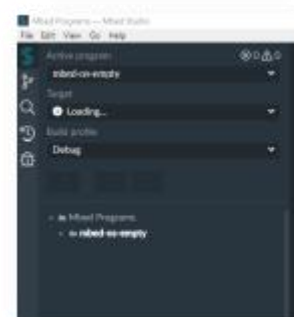


## Creating or importing programs

In Mbed Studio you can create your own programs, or import pre-created programs into the IDE. In order to create a new program:

1. Open the “File” menu and select “New Program”.
2. To start a program from scratch, select “Empty Mbed OS program” from the dropdown list. Or select one of the pre-written example programs.
3. Give the program a name in the “Program Name” field then click “Add Program” (Note: “make this the active program” should automatically be ticked).

The program should then show up in the explorer on the left-hand side of the IDE as shown:

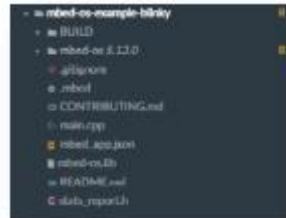


Let’s repeat the process, but this time, we will import a project called “mbed-os-example-blinky”.

1. Open the “File” menu and then select “New Program”.
2. Select “mbed-os-example-blinky” from the dropdown list and then click “Add Program”.

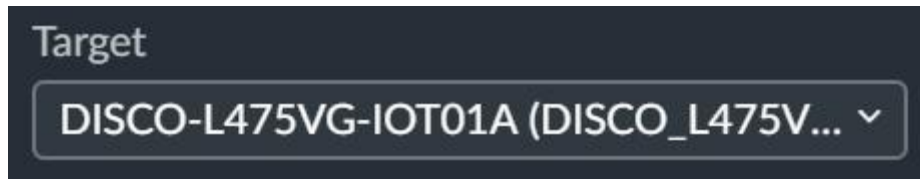
The example program should now appear in the explorer and be populated with the relevant files:



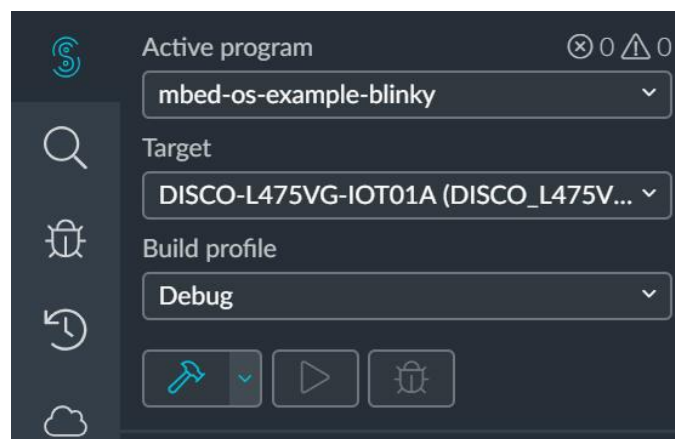


## Building and running a program

Now you can plug in your Mbed enabled board, and select it from the list of targets:



Once the board is selected, ensure that “mbed-os-example-blinky” is the current active program, and then select the “Build program” button:



As a result of the build process, Mbed will create a `.bin` file, which is the result of compiling your program into machine code for whichever device you are using. Just like when using the web-based Mbed, copying this file to your board’s internal storage will have the effect of loading the program onto your board, at which point the program (for instance a blinking LED) should run!

Alternatively, pressing “Run program” will automatically build and copy the program to your board. This is the most efficient way to rapidly develop, deploy, and test your code.

## Exercise

Starting from the example above, write an embedded application that executes a continuous loop in which each LED, 1 and 2 on the board, blink once every second consecutively, then all of them flash simultaneously for a second, and the process repeats.

# Lab 2: Sensors

Learning outcome: Understand how to access on-board sensors, and use serial communication to transfer data

## Introduction

### Lab overview

In this lab you will learn how to interface with environmental sensors using the DISCO-L475VG-IOT01A board and Mbed API.

The DISCO-L475VG-IOT01A device comes with on-board sensors, including a pressure sensor (LPS22HB), 3-axis accelerometer and 3-axis gyroscope (LSM6DSL), a 3-axis magnetometer (LIS3MDL), a humidity sensor (HTS221), a capacitive digital sensor for relative humidity and temperature (HTS221), a Time-of-Flight and gesture-detection sensor (VL53L0X), and two digital omnidirectional microphones (MP34DT01).

In this lab, we will read data from the sensors, transmit them to the PC via UART, and display the readings on a terminal emulator.

## Requirements

### Software functions

The functions that may be used in this lab are listed below:

#### UART functions

You will be using the *BufferedSerial* API to communicate with the PC.

#### On-board Sensor functions

#### Initialization of Sensors

<i>HTS221</i>	Temperature	<i>BSP_TSENSOR_Init()</i>
	Humidity	<i>BSP_HSENSOR_Init()</i>
<i>LPS22HB</i>	Pressure	<i>BSP_PSENSOR_Init()</i>
<i>LIS3MDL</i>	Magnetometer	<i>BSP_MAGNETO_Init()</i>
<i>LSM6DSL</i>	Accelerometer	<i>BSP_ACCELERO_Init()</i>
	Gyroscope	<i>BSP_GYRO_Init()</i>



## Reading Data from sensors

*HTS221*

```
float BSP_TSENSOR_ReadTemp(void)
```

Get the temperature reading from the HTS221 sensor.

```
float BSP_HSENSOR_ReadHumidity(void)
```

Get the humidity reading from the HTS221 sensor.

*LPS22HB*

```
float BSP_PSENSOR_ReadPressure(void)
```

Get the pressure reading from the LPS25H sensor.

*LIS3MDL*

```
void BSP_MAGNETO_GetXYZ(int16_t *pDataXYZ)
```

Get the 3-axis values of the magnetometer.

```
void BSP_MAGNETO_LowPower(uint16_t status)
```

Activates low power mode in the magnetometer.

*LSM6DSL*

```
void BSP_GYRO_GetXYZ(float* pData)
```

Get the 3-axis values of the gyroscope.

```
void BSP_GYRO_LowPower(uint16_t status)
```

Activates low power mode in the gyroscope.

```
void BSP_ACCELERO_AccGetXYZ(int16_t *pDataXYZ)
```

Get the 3-axis values of the accelerometer.

```
void BSP_ACCELERO_LowPower(uint16_t status)
```

Activates low power mode in the accelerometer.

## Getting Set-up

You will need to make sure you've included the board support files in your project, so that you can access the sensor APIs. You will also need to enable floating-point support in `printf` output.

The board support files for the board we're using are here:

[https://os.mbed.com/teams/ST/code/BSP\\_B-L475E-IOT01/](https://os.mbed.com/teams/ST/code/BSP_B-L475E-IOT01/)

To add this to your Mbed Studio project go to File > Add Library to Active Program then paste the URL and select the default/master branch.

To enable floating-point support, you need to enable the `minimal-printf-enable-64-bit` and `minimal-printf-enable-floating-point` settings in the `mbed.lib.json` file.

# Application Code

In this lab exercise we will write a program that reads the temperature, humidity, pressure, magnetometer, accelerometer, and gyroscope values from the sensors on-board, every three seconds. The temperature is measured in degrees Celsius but we will convert it to both Fahrenheit and Kelvins.

The program will then enter sleep mode and wait for interrupts. At the same time the program will blink a LED every second to show it's still responsive. We will inspect the sensor readings via the serial debug interface.

## Program structure

Here is an overview of how we will structure our program:

- Global variables
  - A `BufferedSerial` object for communicating with the PC
  - A `DigitalOut` object for the LED.
  - Two `LowPowerTicker` objects for tracking events:
    - One for flashing the LED every second
    - One for triggering sensor readings
- Initialisation
  - Initialize sensors.
  - Print a welcome message.
  - Activate tickers.
- Ticker Handlers
  - Toggle the LED and update the measurements.
  - Set a flag that indicates that the measurements need to be read and displayed again.
- Main function
  - In a loop:
    - Check if the flag is set.
    - Read from the sensors.
    - Convert the temperature into Fahrenheit and Kelvins.
    - Write the output to the UART.
    - Enter sleep mode.

## Global Variables

Start by creating global variables for the LED and the serial port, and override the console device so we can use `printf`.

```
static DigitalOut led(LED1);
```

```
static BufferedSerial serial_port(USBTX, USBRX, 9600);
```

```
FileHandle *mbed::mbed_override_console(int fd)
{
    return &serial_port;
}
```

## Initialisation

Turn the LED on, and write a welcome message over the serial port. Test your program at this point to make sure it's working.

The code might look like this:

```
int main()
{
    led = true;
    printf("Hello, world.\n");

    return 0;
}
```

## Tickers

Now you've checked that you can compile your program, and can read from the serial port, create the ticker objects and hook them up to functions that will run each time the ticker ticks.

**Global variables:**

```
static LowPowerTicker ledTicker;
static LowPowerTicker readoutTicker;

static bool shouldReadSensors;
```

**Handlers:**

```
static void ledTick()
{
    led = !led;
}
```

```
static void readoutTick()
{
    shouldReadSensors = true;
}
```

Initialisation code:

```
ledTicker.attach(&ledTick, 1s);
readoutTicker.attach(&readoutTick, 3s);
```

At this point, you should compile and run your program again. The LED should toggle its state every second.

## Reading from sensors

Now, we can read and display sensor values. To do this, the sensors must be initialised, so create a function to do that, and call it from main:

```
static void initialiseSensors()
{
    BSP_TSENSOR_Init();
    BSP_HSENSOR_Init();
    BSP_PSENSOR_Init();
    BSP_MAGNETO_Init();
    BSP_ACCELERO_Init();
    BSP_GYRO_Init();
}
```

```
int main()
{
    ...
    initialiseSensors();
    ...
}
```

And finally, create a function to read the sensors and print out their values:

```
static void readSensors()
{
    printf("Sensor values:\n");
```

```
float temp = BSP_TSENSOR_ReadTemp();
printf("* Temperature: %f C, %f F, %f K\n",
temp,
convertCelsiusToFahrenheit(temp),
convertCelsiusToKelvin(temp));

...
}
```

You will need to implement the temperature conversion functions yourself, and you will need to display sensor data from the other sensors.

Now, in your main loop, check to see if `shouldReadSensors` is true, and if it is, call your `readSensors` function, and clear the flag:

```
while (true) {
    sleep();
    if (shouldReadSensors) {
        readSensors();
        shouldReadSensors = false;
    }
}
```

## Expected Output

When your program is running, you should see output like the following:

```
Hello, world.
Sensor values:
* Temperature: 24.369190 C, 75.864540 F, 297.519196 K
* Humidity: 53.434387
* Pressure: 1000.169983
* Magneto: 80 -433 221
* Gyro: 0.000000 -1120.000000 770.000000
* Accelerometer: 5 -1 1011
```

(Your sensor values will be different)

---

# Lab 3: Local Connectivity

Learning outcome: Understand how Bluetooth LE communication works, and how it can be used to transmit and receive basic data

## Introduction

### Lab overview

In this lab, you will learn how to communicate with the outside world through Bluetooth, and more specifically Bluetooth Low Energy — or BLE. We will program the embedded device to transmit live on-board sensor data to a mobile app that can communicate with BLE devices.

## Hardware and Software Requirements

- The DISCO-L475VG-IOT01A board
- An Android mobile phone, with a Bluetooth LE Scanner App
- Mbed Studio or another suitable development environment

Before getting started, you will need to include the BLE utility library from here in your Mbed Studio project:

<https://github.com/ARMmbed/mbed-os-ble-utils>

Just follow the same process for adding the board support library (which will also need to be included) to the project in the earlier lab.

## Working with Bluetooth

Detailed information on the Mbed OS Bluetooth APIs is available here:

<https://os.mbed.com/docs/mbed-os/v6.10/apis/bluetooth-apis.html>

And, you can find a range of examples here:

<https://github.com/ARMmbed/mbed-os-example-ble/>

## Configuration

Bluetooth LE can appear quite complex to get started with, but by writing our code in an efficient manner, we can make it quite straightforward.

## Application Code

We will program the embedded device to advertise its presence over BLE, and transmit live readings from the temperature, humidity, and pressure sensors that we used in the previous lab. We will also allow the on-board LED to be toggled remotely.

## High Level Overview

- Initialise BLE
- Advertise presence
- Accept connections, and periodically transmit sensor readings over the connection
- Respond to write events, to allow the on-board LED to be toggled

## Initialising BLE

To get started with BLE, we need to create some utility classes that will manage the interface to the BLE API.

First, create a class to manage the BLE process:

```
class Lab3ServerProcess : public BLEProcess {
public:
    Lab3ServerProcess(events::EventQueue &event_queue, BLE &ble_interface)
        : BLEProcess(event_queue, ble_interface) {}

    const char *get_device_name() override {
        static const char name[] = "Lab 3 Server";
        return name;
    }
};
```

Here, we've created a class called `Lab3ServerProcess`, which subclasses `BLEProcess`, and simply returns the name that we want to advertise.

Now, we need to create a server class that will manage the service and characteristics that we support:

```
class Lab3Server : ble::GattServer::EventHandler {
public:
    Lab3Server() {}
    ~Lab3Server() {}

    void start(BLE &ble, events::EventQueue &event_queue) {
        const UUID uuid = GattService::UUID_ENVIRONMENTAL_SERVICE;

        GattCharacteristic *charTable[] = {};
```



```

GattService sensorService(uuid, charTable,
                           sizeof(charTable) / sizeof(charTable[0]));

ble.gattServer().addService(sensorService);
ble.gattServer().setEventHandler(this);

printf("Service started.\n");
}
};

```

This should be enough to advertise our embedded system's presence over BLE, so to try it out, initialise BLE in the `main` function:

```

static EventQueue event_queue(10 * EVENTS_EVENT_SIZE);

int main() {
    BLE &ble = BLE::Instance();

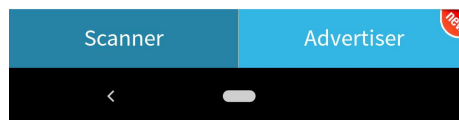
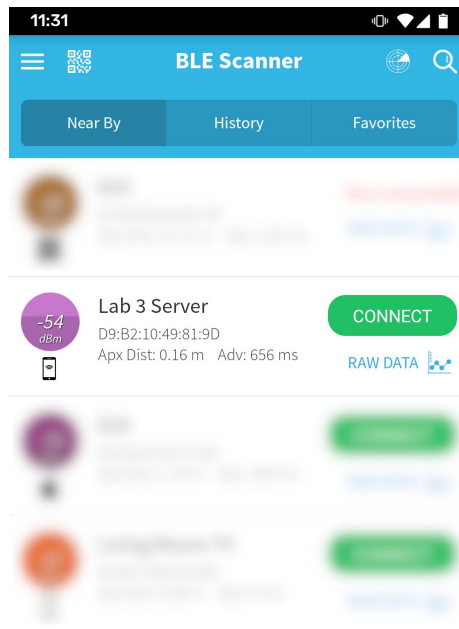
    printf("Lab 3 - BLE");

    Lab3ServerProcess ble_process(event_queue, ble);
    Lab3Server server;
    ble_process.on_init(callback(&server, &Lab3Server::start));
    ble_process.start();

    return 0;
}

```

Build and run this on the embedded device, and open up the BLE scanning app on your mobile device. Depending on which BLE scanning app you use, you should see something like the following:



If you connect to the advertised server, you will see that an “Environmental” service exists, but with no further details. This is the service that we will be populating with sensor data.

## Advertising Sensor Data

As in Lab 2, we have to start by initialising the sensors we are going to use. We will be using the Temperature, Humidity, and Pressure sensors in this lab, so create a function to initialise them, and augment `main` to call this before the Bluetooth setup code:

```
static void initialiseSensors() {
    BSP_TSENSOR_Init();
    BSP_HSENSOR_Init();
    BSP_PSENSOR_Init();
}
```

Now that the sensors are initialised, we can think about how we transmit their readings.

To do this, for each sensor we are going to create a BLE Characteristic that is associated with the Environmental service we have already created. To help us do this, we’ll create a helper class for managing the “current value” of that characteristic.

The helper class should look like the following:

```
class SensorReadingCharacteristic : public GattCharacteristic {
```

public:

```
SensorReadingCharacteristic(const UUID &uuid)
    : GattCharacteristic(
        uuid, (uint8_t *)&stringRepr_, sizeof(stringRepr_),
        sizeof(stringRepr_),
        GattCharacteristic::BLE_GATT_CHAR_PROPERTIES_READ |
        GattCharacteristic::BLE_GATT_CHAR_PROPERTIES_NOTIFY),
        internalValue_(0) {
    updateStringRepr();
}
```

```
void updateValue(BLE &ble, float newValue) {
    internalValue_ = newValue;
    updateStringRepr();

    ble.gattServer().write(getValueHandle(), (uint8_t *)&stringRepr_,
        sizeof(stringRepr_));
}
```

private:

```
float internalValue_;
uint8_t stringRepr_[16];

void updateStringRepr() {
    sprintf((char *)stringRepr_, "%f", internalValue_);
}
};
```

What's happening here is we're tracking the current value of the sensor in the `internalValue_` field, and then converting it into a string representation, when the value is updated. Later on, we'll call `updateValue` for each characteristic representing a sensor, and pass in the current reading of that sensor.

When `updateValue` is called, the internal value is updated, the string representation is generated, and the value of the string representation is transmitted over BLE. We're using a string representation here so that you can see the value in your BLE scanner app. If you were writing an Android app, there'd be no need to do the string conversion.

We're also saying that these characteristics can be read from, and they can also notify the client when their value changes. This lets us repeatedly update the value, and the client will see those changes when it is connected.

Now that we have this class, we'll create an instance of it for each of our sensors. To do this, we'll create local fields in our `Lab3Server` class, and initialise them accordingly:

```
class Lab3Server : ble::GattServer::EventHandler {
public:
    Lab3Server()
        : temperature_(GattCharacteristic::UUID_TEMPERATURE_CHAR),
          humidity_(GattCharacteristic::UUID_HUMIDITY_CHAR),
          pressure_(GattCharacteristic::UUID_PRESSURE_CHAR) {}

    // ... existing code ... //

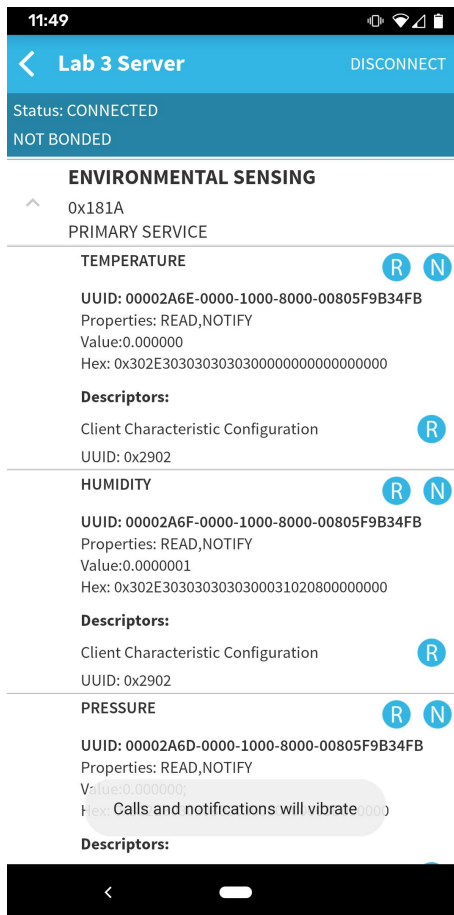
private:
    SensorReadingCharacteristic temperature_;
    SensorReadingCharacteristic humidity_;
    SensorReadingCharacteristic pressure_;
};
```

What we've done is to create three characteristics, one each for temperature, humidity, and pressure. We've also initialised them (in the constructor) with the correct identifiers, so that BLE clients understand what these values are.

Finally, we have to tell our service to advertise these characteristics, by modifying the characteristics array in the start method to include them:

```
GattCharacteristic *charTable[] = {&temperature_, &humidity_, &pressure_};
```

Once you've done this - try it out! If all goes well, after you build and run, you should be able to connect to the device, and see those three characteristics appear:



But – all the values are zero!

This is because we haven't written the logic to update the values yet -- however, it's important you reach this step before we go any further.

We'll update these values once every second, so to do this, we'll create a function that performs the update, and connect it to the event queue.

Create the following function in your `Lab3Server` class:

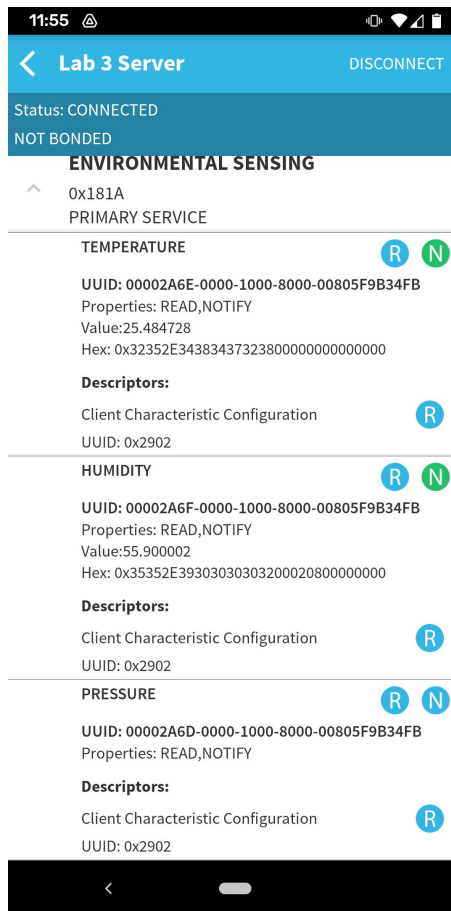
```
void updateSensors(BLE &ble) {
    printf("Updating sensors...\n");

    temperature._updateValue(ble, BSP_TSENSOR_ReadTemp());
    humidity._updateValue(ble, BSP_HSENSOR_ReadHumidity());
    pressure._updateValue(ble, BSP_PSENSOR_ReadPressure());
}
```

Any time this function is called, it will call `updateValue` on each characteristic, with the latest reading from the sensor. To trigger this function every second, update the start method to include the following line, at the end of the routine:

```
event_queue.call_every(1000ms, [this, &ble] { updateSensors(ble); });
```

This tells the event queue to call the `updateSensors` function every second. When you connect and read the characteristic values -- or use the notify feature on the BLE scanner app -- you should see the various readings appear:



## Toggling the LED

The last piece of the puzzle is to support remote toggling of the on board LED. To achieve this, we will create a writable characteristic, i.e. one that can be written to by the client. When the value written is zero, the LED will be turned off. When the value written is one, the LED will be turned on.

The BLE API provides us with a very useful characteristic class called a `ReadWriteGattCharacteristic`, so we'll use this to implement the functionality.

We need to add some additional fields to `Lab3Server` to keep track of things:

```
DigitalOut led_;
uint8_t ledValue_;
ReadWriteGattCharacteristic<uint8_t> ledChar_;
```

Next, we need to update the constructor to initialise them:

```
Lab3Server()
: temperature_(GattCharacteristic::UUID_TEMPERATURE_CHAR),
```

```
humidity_(GattCharacteristic::UUID_HUMIDITY_CHAR),  
pressure_(GattCharacteristic::UUID_PRESSURE_CHAR), led_(LED1),  
ledValue_(0), ledChar_(0xA000, &ledValue_) {}
```

Here, we initialise the LED object to point to LED1 on the board, set the initial value to zero, and initialise the characteristic with a custom identifier, 0xA000.

Now that we have the basic controls in place, we can add the characteristic to the char table list:

```
GattCharacteristic *charTable[] = {&temperature_, &humidity_, &pressure_, &ledChar_};
```

And finally implement the method that will actually control the LED:

```
virtual void onDataWritten(const GattWriteCallbackParams &params) override {  
    if ((params.handle == ledChar_.getValueHandle()) && (params.len == 1)) {  
        printf("New LED value: %x\r\n", *(params.data));  
        ledValue_ = *(params.data);  
        led_ = ledValue_;  
    }  
}
```

You will notice that this method is marked “*override*” because it overrides a method in the base class, and is called when new data is written from a client. In the method, we check that it is intended for the LED, and if so, update the LED value and state accordingly.

When you build and run your program, you should see the new characteristic appear, and when you write 1 (01) to it (using byte representation, not text) the LED on the board will turn on. Writing a 0 (00) will turn off the LED:



12:32

Lab 3 ServerDISCONNECT

Status: CONNECTED

NOT BONDED

0x181A

PRIMARY SERVICE

TEMPERATURE

UUID: 00002A6E-0000-1000-8000-00805F9B34FB

Properties: READ,NOTIFY

Descriptors:

Client Characteristic Configuration

UUID: 0x2902

HUMIDITY

UUID: 00002A6F-0000-1000-8000-00805F9B34FB

Properties: READ,NOTIFY

Descriptors:

Client Characteristic Configuration

UUID: 0x2902

PRESSURE

UUID: 00002A6D-0000-1000-8000-00805F9B34FB

Properties: READ,NOTIFY

Descriptors:

Client Characteristic Configuration

UUID: 0x2902

CUSTOM CHARACTERISTIC

UUID: 0000A000-0000-1000-8000-00805F9B34FB

Properties: READ,WRITE

Write Type:WRITE REQUEST

12:32

Lab 3 ServerDISCONNECT

Status: CONNECTED

NOT BONDED

0x181A

PRIMARY SERVICE

TEMPERATURE

UUID: 00002A6E-0000-1000-8000-00805F9B34FB

Properties: READ,NOTIFY

Descriptors:

Client Characteristic Configuration

UUID: 0x2902

HUMIDITY

UUID: 00002A6F-0000-1000-8000-00805F9B34FB

Properties: READ,NOTIFY

Descriptors:

Client Characteristic Configuration

UUID: 0x2902

PRESSURE

UUID: 00002A6D-0000-1000-8000-00805F9B34FB

Properties: READ,NOTIFY

Descriptors:

Client Characteristic Configuration

UUID: 0x2902

CUSTOM CHARACTERISTIC

UUID: 0000A000-0000-1000-8000-00805F9B34FB

Properties: READ,WRITE

Write Type:WRITE REQUEST

Write Value

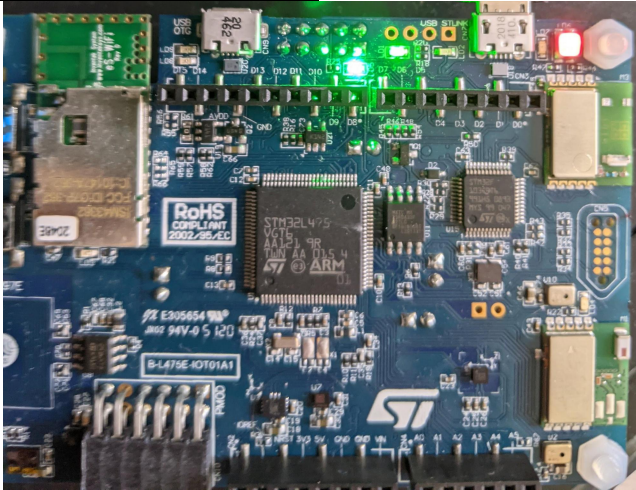
NEW

Byte Array

01

CANCEL

OK



# 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/](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:

## New Project



You have 22 projects remaining in your quota. Request an increase or delete projects. [Learn more](#)

[MANAGE QUOTAS](#)

Project name \*

arm-edx-lab4



Project ID: arm-edx-lab4-387807. It cannot be changed later. [EDIT](#)

Location \*

 No organisation

[BROWSE](#)

Parent organisation or folder

[CREATE](#)

[CANCEL](#)

**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:

## Get started

### 1 Select a Cloud Firestore mode — 2 Choose where to store your data

Cloud Firestore is the next generation of Cloud Datastore. You can use Cloud Firestore in either Native mode or Datastore mode, each with distinct system behaviour optimised for different types of projects. [Pricing](#) for both modes is based on location, stored data, operations and network egress, with a daily free quota for each. [Learn more about choosing a mode](#)

The mode you select here will be permanent for this project

	Native mode	Datastore mode
	Enable all of Cloud Firestore's features, with offline support and real-time synchronisation.	Leverage Cloud Datastore's system behaviour on top of Cloud Firestore's powerful storage layer.
	<a href="#">SELECT NATIVE MODE</a>	<a href="#">SELECT DATASTORE MODE</a>
API	Firestore	Datastore
Scalability	Automatically scales to millions of concurrent clients	Automatically scales to millions of writes per second
App engine support	Not supported in the App Engine standard Python 2.7 and PHP 5.5 runtimes	All runtimes
Max writes per second	No limit	No limit
Real-time updates	✓	✗
Mobile/web client libraries with offline data persistence	✓	✗

[SHOW MORE](#)

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

## Get started

### ✓ Select a Cloud Firestore mode — 2 Choose where to store your data

You've selected Cloud Firestore in Native mode. Now choose a database location.

The location of your database affects its cost, availability and durability. Choose a regional location (lower write latency, lower cost) or a multi-region location (higher availability, higher cost). [Learn more](#)

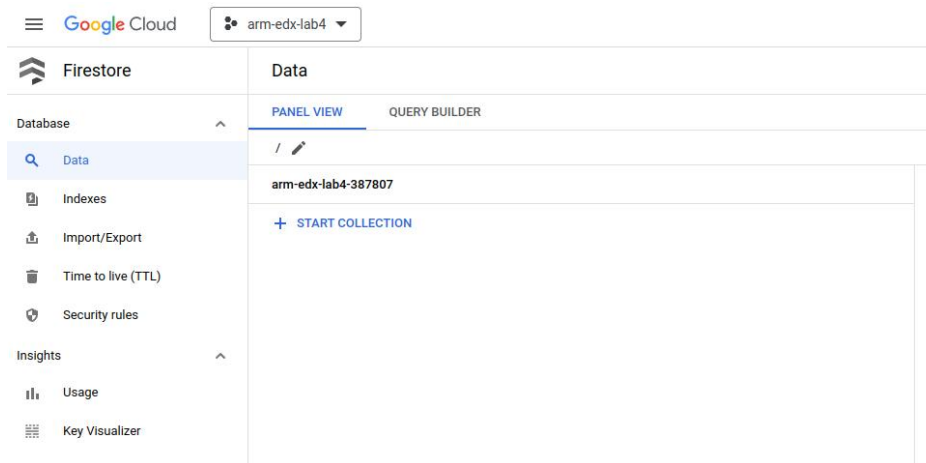
Your location selection is permanent

Select a location  
eur3 (Europe)

To improve performance, store your data close to the users and services that need it

[CREATE DATABASE](#) [BACK](#)

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):

✓ Service account details

2 Grant this service account access to the project (optional)

Grant this service account access to arm-edx-lab4 so that it has permission to complete specific actions on the resources in your project. [Learn more](#)

Role  
Cloud Datastore User

Provides read/write access to data in a Cloud Datastore database. Intended for application developers and service accounts.

IAM condition (optional) ?  
[+ ADD IAM CONDITION](#)

[+ ADD ANOTHER ROLE](#)

[CONTINUE](#)

3 Grant users access to this service account (optional)

[DONE](#)

[CANCEL](#)

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:

### Create private key for 'datastore'

Downloads a file that contains the private key. Store the file securely because this key cannot be recovered if lost.

**Key type**

☒ JSON  
Recommended

☐ P12  
For backward compatibility with code using the P12 format

[CANCEL](#) [CREATE](#)

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:

**1 Configuration** — **2 Code**

### Basics

**Environment**  
1st gen ▼ ?


**Function name \***  
StoreSensorData ?

**Region**  
europe-west2 ▼ ?

### Trigger

**⌚ HTTP**

**Trigger type**  
HTTP ▼

**URL** 

**Authentication**  

☒ **Allow unauthenticated invocations**  
Check this if you are creating a public API or website.

☐ **Require authentication**  
Manage authorised users with Cloud IAM.

☐ **Require HTTPS** ?

**SAVE**

CANCEL

### Runtime, build, connections and security settings



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

Runtime  
Node.js 18

Source code  
Inline Editor

+

index.js ...

package.json ...

Entry point \*  
storeData

Press Alt+F1 for accessibility options.

```

1 // Import the Firestore module
2 const Firestore = require('@google-cloud/firestore');
3
4 // Connect to the Firestore database
5 const db = new Firestore({
6   projectId: '<YOUR-PROJECT-ID>',
7   keyFilename: 'key.json'
8 });
9
10 // Cloud Function entry point
11 exports.storeData = async (req, res) => {
12   // Read and validate incoming sensor data message
13   const messageData = req.body;
14
15   if (messageData === undefined || messageData === null) {
16     console.error('Unable to read message');
17     res.status(400).send();
18     return;
19   }
20
21   // Insert the prediction into the database
22   const timestamp = Date.now()
23
24   await db.collection('sensor-data').add({
25     timestamp,
26     ...messageData
27   });
28
29   // Return a successful result
30   res.status(200).send();
31 };
32

```

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 endpoint, 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"
  }
}

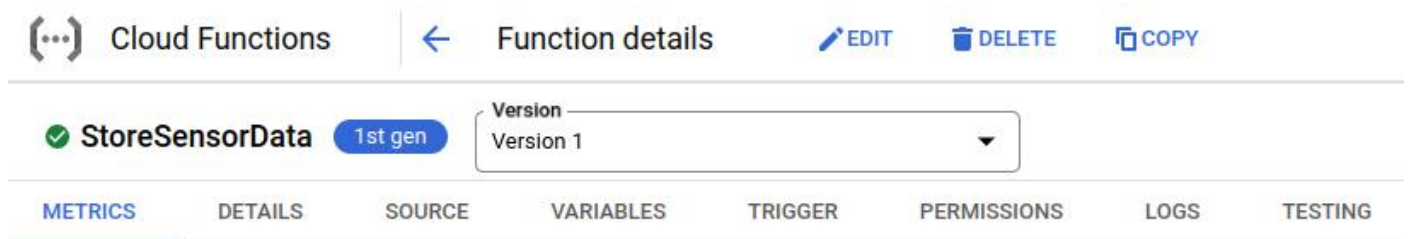
```

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:

Grant access to 'StoreSensorData'

Grant principals access to this resource and add roles to specify what actions the principals can take. Optionally, add conditions to grant access to principals only when a specific criteria is met. [Learn more about IAM conditions](#)

**Resource**

StoreSensorData

**Add principals**

Principals are users, groups, domains or service accounts. [Learn more about principals in IAM](#)

New principals \*

allUsers

**Assign Roles**

Roles are composed of sets of permissions and determine what the principal can do with this resource. [Learn more](#)

Role \*

Cloud Functions Invoker

Ability to invoke HTTP functions with restricted access.

+ ADD ANOTHER ROLE

SAVE CANCEL

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

Permissions

VIEW BY PRINCIPALS VIEW BY ROLES

+ GRANT ACCESS - REMOVE ACCESS

Filter Enter property name or value

Type	Principal ↑	Name	Role	Inheritance	
<input type="checkbox"/>	allUsers		Cloud Functions Invoker		
<input type="checkbox"/>		App Engine default service account	Editor	arm-edx-lab4	
<input type="checkbox"/>	service-159134531828@gcf-admin-robot.iam.gserviceaccount.com	Google Cloud Functions Service Agent for Project	Cloud Functions Service Agent	arm-edx-lab4	
<input type="checkbox"/>			Owner	arm-edx-lab4	

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:

The screenshot shows the Google Cloud Functions console for a function named 'StoreSensorData'. The 'TESTING' tab is active. On the left, the 'Configure triggering event' section shows a JSON payload: `{ "temperature": 20.5, "humidity": 66.1234, "pressure": 999.45 }`. On the right, the 'Test command' section shows a curl command: `curl -m 70 -X POST undefined \ -H "Authorization: bearer $(gcloud auth print-identity-token)" \ -H "Content-Type: application/json" \ -d '{ "temperature": 20.5, "humidity": 66.1234, "pressure": 999.45 }'`. Below these sections, the 'Output' section shows 'Complete' status. The 'Logs' section at the bottom shows a log entry: 'Function execution started' and 'Function execution took 1362 ms, finished with status code: 200'.

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:

The screenshot shows the Google Cloud Firestore console. The left sidebar shows the 'Database' section with 'Data' selected. The main area shows the 'sensor-data' collection. The collection is empty, with a '+ START COLLECTION' button. The 'Data' tab is active, showing a table with one document. The document ID is '02xB4kiVFaPg5fsp113U'. The document contains the following fields: 'humidity' (66.1234), 'pressure' (999.45), 'temperature' (20.5), and 'timestamp' (1686914351052).

## 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*” :

The screenshot shows the Google Cloud console interface for creating a new Cloud Function. At the top, the Google Cloud logo and the project name 'arm-edx-lab4' are visible. Below the navigation bar, the 'Cloud Functions' section is active, with a 'Create function' button. The 'Configuration' tab is selected, showing the 'Basics' section. The 'Environment' is set to '1st gen', the 'Function name' is 'GetRecentSensorData', and the 'Region' is 'europe-west2'. The 'Trigger' section is expanded, showing 'HTTP' as the trigger type. The 'Trigger type' dropdown is set to 'HTTP'. The 'URL' field is empty. Under 'Authentication', the 'Allow unauthenticated invocations' option is selected, with a note: 'Check this if you are creating a public API or website.' The 'Require authentication' option is unselected, with a note: 'Manage authorised users with Cloud IAM.' The 'Require HTTPS' option is also unselected. At the bottom of the configuration section, there are 'SAVE' and 'CANCEL' buttons. Below the configuration section, the 'Runtime, build, connections and security settings' section is partially visible.

Google Cloud arm-edx-lab4

Cloud Functions Create function

1 Configuration — 2 Code

**Basics**

Environment  
1st gen

Function name \*  
GetRecentSensorData

Region  
europe-west2

**Trigger**

HTTP

Trigger type  
HTTP

URL

**Authentication**

☒ Allow unauthenticated invocations  
Check this if you are creating a public API or website.

☐ Require authentication  
Manage authorised users with Cloud IAM.

☐ Require HTTPS

SAVE CANCEL

Runtime, build, connections and security settings

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:

The screenshot displays the Google Cloud Functions console interface for a function named 'GetRecentSensorData'. The 'Testing' tab is active, showing a 'Test command' section with a curl command: `curl -s -H 'Authorization: bearer $(gcloud auth print-identity-token)' -d {}`. Below this, the 'Output' section shows the JSON response: `{\"temperature\":29.5,\"humidity\":66.1234,\"pressure\":999.45,\"timestamp\":1686914351852}`. The 'Logs' section shows the function execution started and finished with status code 200.

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:

Data

Cloud Firestore in Native mode ⓘ Database location: c

PANEL VIEW

QUERY BUILDER

/ > sensor-data > GEcn1zgpA7CpWGLxj5a ✎

arm-edx-lab4-387807

+ START COLLECTION

sensor-data >

sensor-data

+ ADD DOCUMENT

02xB4kiVFpPg5f5p113U

1ffbSq3RwbP7We7aZXds

8A6y9k4c0sOyiqDcf9MS

9bFeW8FBVWk7Zpux2J

AwrmcovYZqUS4uATkCiE

GEcn1zgpA7CpWGLxj5a >

Lx6Gfh9e321AzPdC6z2

QHMhpnLDLgIKQqGogs0d

XQnGzTTC4qviQCIOJ7gj

ZMtZoMXIExU1JSr14JFF

a3Hrptm8YM8ZvbXARXPU

csHhZBAs8xe06p7YBmzL

icUsP50yXZgGv0vrdi8E

GEcn1zgpA7CpWGLxj5a

+ START COLLECTION

+ ADD FIELD

humidity: 43.059196

pressure: 1017.460022

temperature: 30.876495

timestamp: 1686929777900

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:



# Sensor Data

Time	Temperature	Humidity	Pressure
16/06/2023, 12:19:11	20.5 °C	66%	999.5
16/06/2023, 16:35:40	30.8 °C	43%	1017.7
16/06/2023, 16:35:45	30.8 °C	43%	1017.7
16/06/2023, 16:36:04	31.0 °C	43%	1017.5
16/06/2023, 16:36:05	31.0 °C	43%	1017.5
16/06/2023, 16:36:06	30.9 °C	43%	1017.4
16/06/2023, 16:36:08	31.0 °C	43%	1017.5
16/06/2023, 16:36:09	31.0 °C	43%	1017.5
16/06/2023, 16:36:10	30.9 °C	43%	1017.6
16/06/2023, 16:36:11	30.9 °C	43%	1017.6

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.

---

## 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” :

Cloud Functions | Create function

1 Configuration — 2 Code

**Basics**

Environment  
1st gen

Function name \*  
ProcessSensorData

Region  
europe-west2

**Trigger**

HTTP

Trigger type  
HTTP

URL  
https://europe-west2-arm-edx-lab4-387807.cloudfunctions.net/ProcessSensorData

**Authentication**

☒ Allow unauthenticated invocations  
Check this if you are creating a public API or website.

☐ Require authentication  
Manage authorised users with Cloud IAM.

☐ Require HTTPS

SAVE CANCEL

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”:



## Grant access to 'ProcessSensorData'

Grant principals access to this resource and add roles to specify what actions the principals can take. Optionally, add conditions to grant access to principals only when a specific criteria is met. [Learn more about IAM conditions](#)

### Resource

 ProcessSensorData

### Add principals

Principals are users, groups, domains or service accounts. [Learn more about principals in IAM](#)

New principals \*  
allUsers  

### Assign Roles

Roles are composed of sets of permissions and determine what the principal can do with this resource. [Learn more](#)

Role \*  
Cloud Functions Invoker    
Ability to invoke HTTP functions with restricted access.

[+ ADD ANOTHER ROLE](#)

[SAVE](#) [CANCEL](#)

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:

## Configure triggering event

Press Alt+F1 for accessibility options.

```
1 {
2   "readings": [
3     {
4       "accel": {
5         "x": 1,
6         "y": 1,
7         "z": 1
8       },
9       "gyro": {
10        "x": 1,
11        "y": 1,
12        "z": 1
13      }
14    }
15  ]
16 }
```

[TEST THE FUNCTION](#)

Testing in the Cloud console has a five-minute timeout. Note that this is different from the limit set in the function configuration.

## Test command [TEST IN CLOUD SHELL](#)

```
curl -m 70 -X POST undefined \
-H "Authorization: bearer $(gcloud auth print-identity-token)" \
-H "Content-Type: application/json" \
-d '{
  "readings": [
    {
      "accel": {
        "x": 1,
        "y": 1,
        "z": 1
      },
      "gyro": {
        "x": 1,
        "y": 1,
        "z": 1
      }
    }
  ]
}'
```

## Output Complete

\$

## Logs Fetches (up to 100 entries). [View all logs](#)

SEVERITY	TIMESTAMP	SUMMARY
No older entries found matching current filter.		
>	2023-06-19 11:48:07.099 BST	ProcessSensorData en7nm5z562kc Function execution started
>	2023-06-19 11:48:07.131 BST	ProcessSensorData en7nm5z562kc Activity data processed: { label: 'still' }
>	2023-06-19 11:48:10.640 BST	ProcessSensorData en7nm5z562kc Function execution took 3540 ms, finished with status code: 200

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:

## Data

[PANEL VIEW](#) [QUERY BUILDER](#)

/ > activity-data > v591mgw5hZ85ZaLYxm3U

arm-edx-lab4-387807	activity-data	v591mgw5hZ85ZaLYxm3U
+ START COLLECTION	+ ADD DOCUMENT	+ START COLLECTION
activity-data	v591mgw5hZ85ZaLYxm3U	+ ADD FIELD
sensor-data		activity: "still"
		timestamp: 1687171687131

## 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”:

Cloud Functions

Create function

1 Configuration

2 Code

### Basics

Environment  
1st gen

Function name \*  
GetActivityData

Region  
europe-west2

### Trigger

HTTP

Trigger type  
HTTP

URL  
https://europe-west2-arm-edx-lab4-387807.cloudfunctions.net/GetActivityData

Authentication

- ☒ Allow unauthenticated invocations  
Check this if you are creating a public API or website.
- ☐ Require authentication  
Manage authorised users with Cloud IAM.
- ☐ Require HTTPS

SAVE CANCEL

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:

#### Configure triggering event ?

Press Alt+F1 for accessibility options.

1 {}

(...) TEST THE FUNCTION

Testing in the Cloud console has a five-minute timeout. Note that this is different from the limit set in the function configuration.

#### Test command TEST IN CLOUD SHELL

```
curl -m 70 -X POST undefined \
-H "Authorization: bearer $(gcloud auth print-identity-token)" \
-H "Content-Type: application/json" \
-d '{}'
```

Output Complete

```
$ [{"activity": "still", "timestamp": 1687171687131}]
```

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/](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 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!

/ > activity-data > 3w64zFLzVjYECsQfHS6z

arm-edx-lab4-387807	activity-data	3w64zFLzVjYECsQfHS6z
+ START COLLECTION	+ ADD DOCUMENT	+ START COLLECTION
activity-data	2KyIVRdHbG2fGLZ5cxEf	+ ADD FIELD
sensor-data	2TFSA96TTXFQnEcsjQzi	activity: 'running'
	3w64zFLzVjYECsQfHS6z	timestamp: 1687177966145
	3yfNZbhlFOunBEXxDEi0	
	5gkIXU1Gp13ZzOHcoAeW	
	7h9tP7jQEqIFsyd3M06U	
	8D2si3Jvh4RRsRbZ0nD7	
	8HPUnnNLYBxH6wIRXzxp	
	DlaaCcmKU0riAofzRjrN	
	G3SbHUP2gMtgGN9rt8sf	
	JlaBe067aU0Z68tkT4ym	

# 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

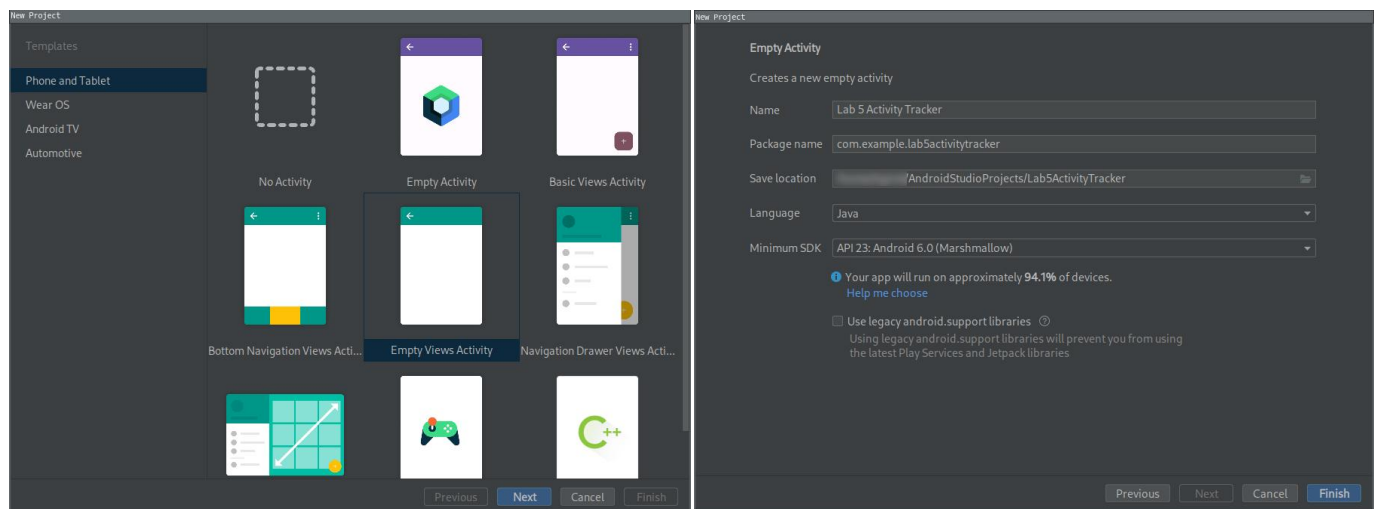
### Getting started

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!

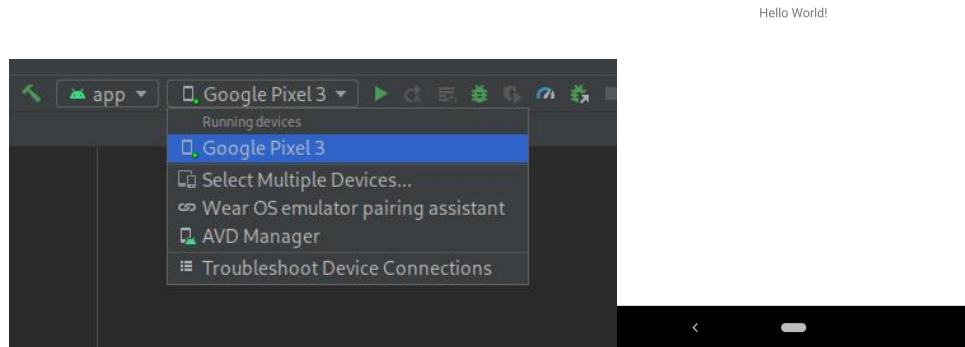
To get started with Android development, you will need an Android phone that’s in developer mode (details available online on how to do this for your specific model), and an installation of the latest version of Android Studio. Android Studio is available here:

<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.

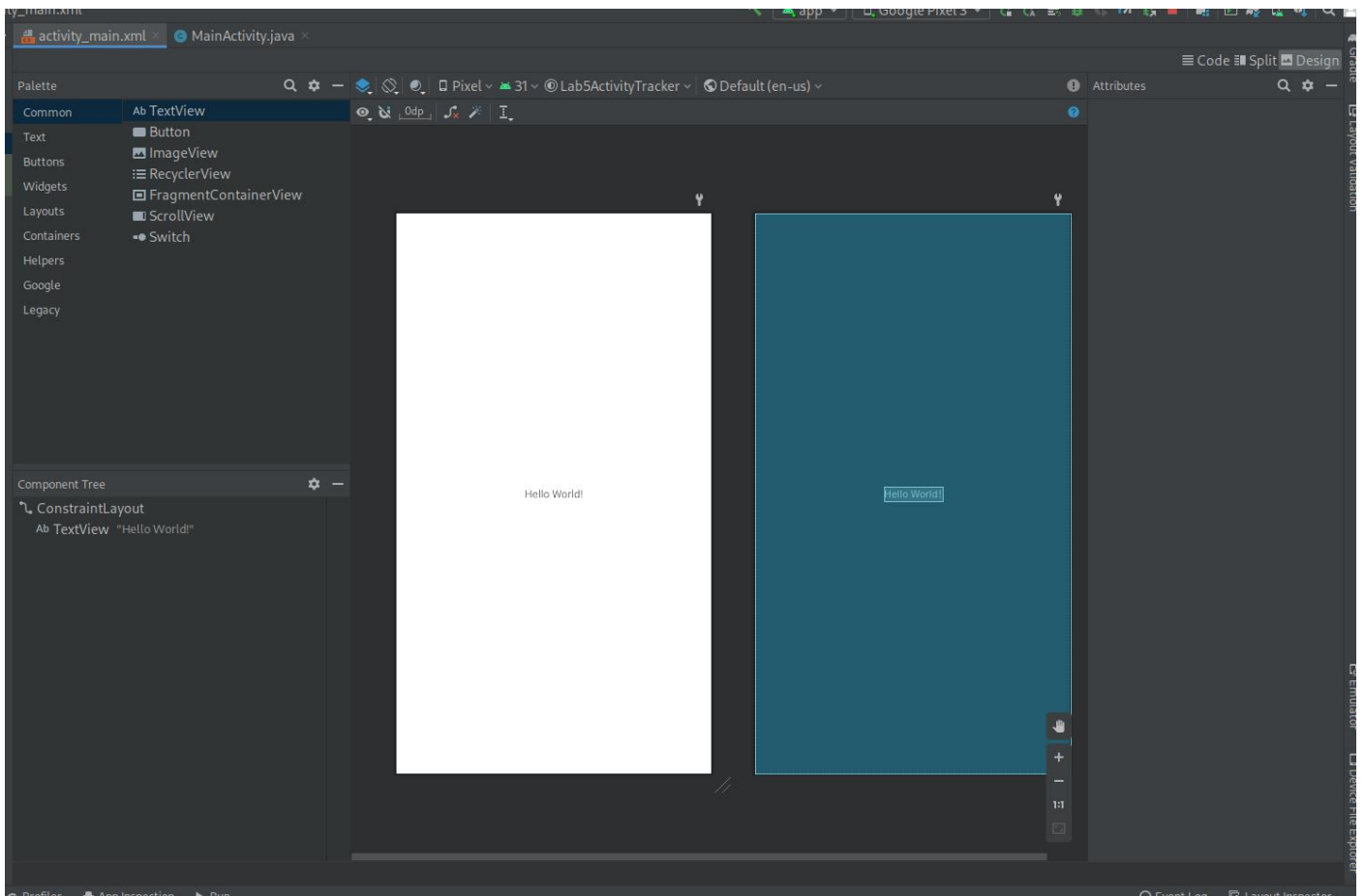
## Designing the User Interface

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

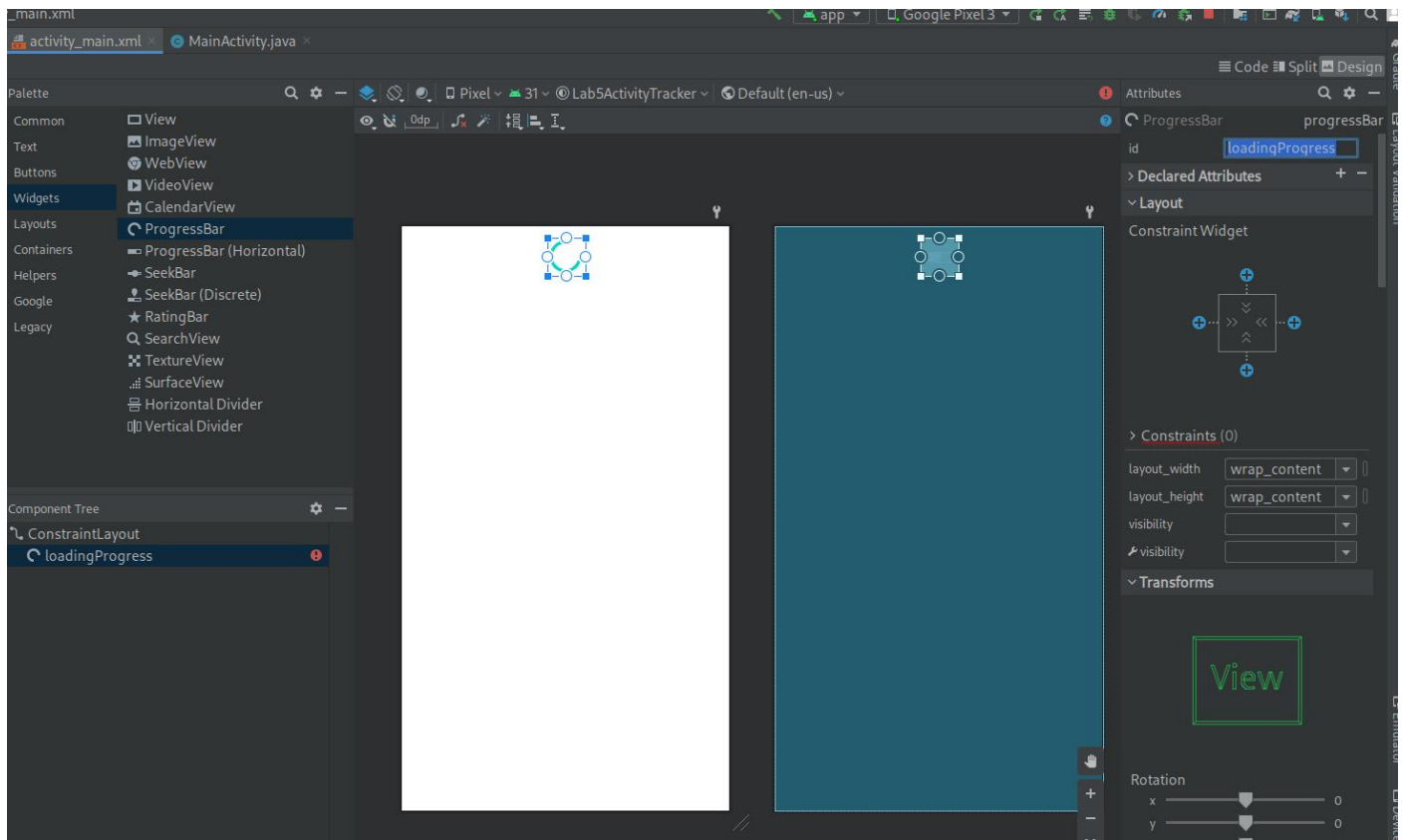
Start by opening up the `activity_main.xml` file in Android Studio, and you should be presented with the UI designer:



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".

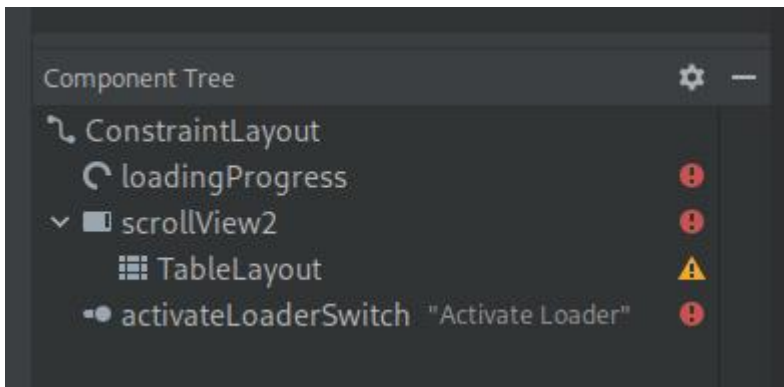


Next, navigate to the “Containers” section of the Palette, and drag in a **ScrollView** component. Once added, it should fill the remaining available area. Inside the **ScrollView** is a **LinearLayout** component that was automatically added. Delete this by expanding the **ScrollView** component in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

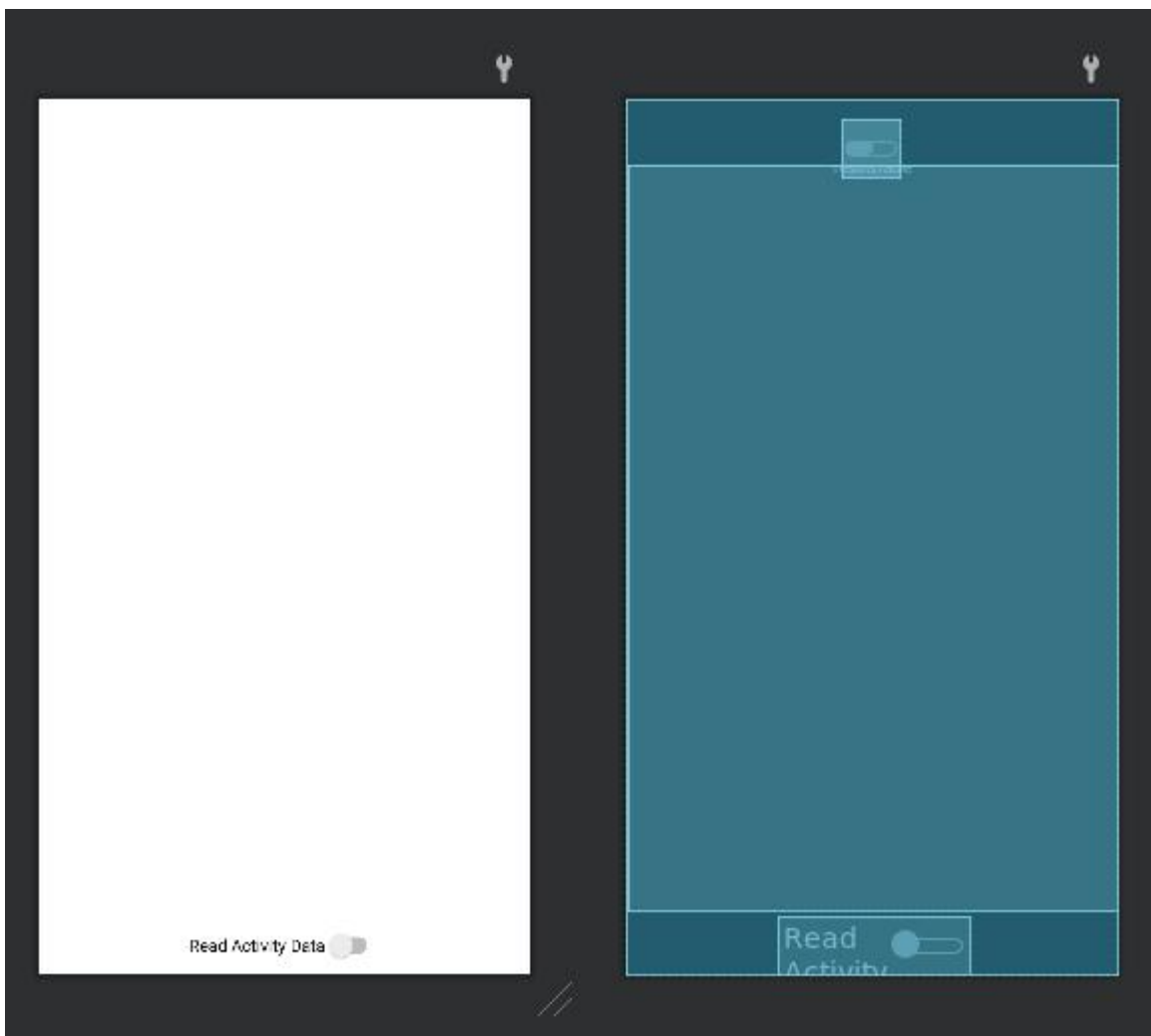
Finally, navigate to the “Buttons” section of the Palette, and add a **Switch** component. You may have to drag this directly into the **ConstraintLayout** component, so that it doesn’t become a child of the **ScrollView**. Give the switch an ID: “**activateLoaderSwitch**”, and give it some descriptive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

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



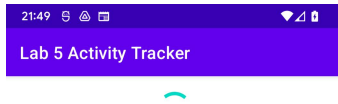
But, the layout won't look very good. The **ConstraintLayout** is a way of laying out components so that they sit relative to each other, and we'll be using this concept to position everything nicely.

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:

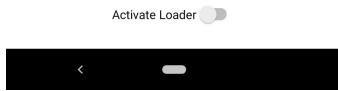


As you can see in the image on the right, the progress bar is at the top, the scroll view is in the middle, and the "Read Activity Data" switch is at the bottom.

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:

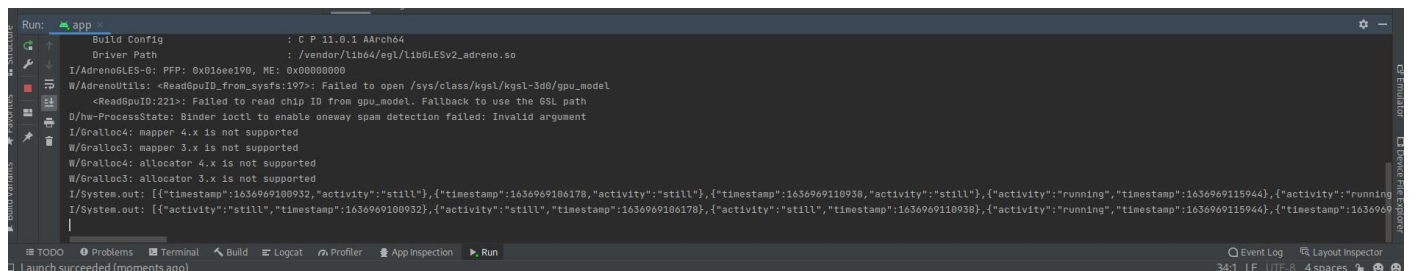
```
String url = "URL_TO_CLOUD_FUNCTION";
this.requestQueue.add(new StringRequest(Request.Method.GET, url, this, this));
```

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.getContext());

    DateFormat fmt = DateFormat.getDateInstance();

    TextView startTimestamp = new TextView(this.getContext());
    startTimestamp.setText(fmt.format(periodStart));
    row.addView(startTimestamp);

    Space s = new Space(this.getContext());
    s.setMinimumWidth(32);
    row.addView(s);

    TextView endTimestamp = new TextView(this.getContext());
    endTimestamp.setText(fmt.format(periodEnd));
    row.addView(endTimestamp);

    Space s2 = new Space(this.getContext());
    s2.setMinimumWidth(32);
    row.addView(s2);

    TextView activityText = new TextView(this.getContext());
    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).

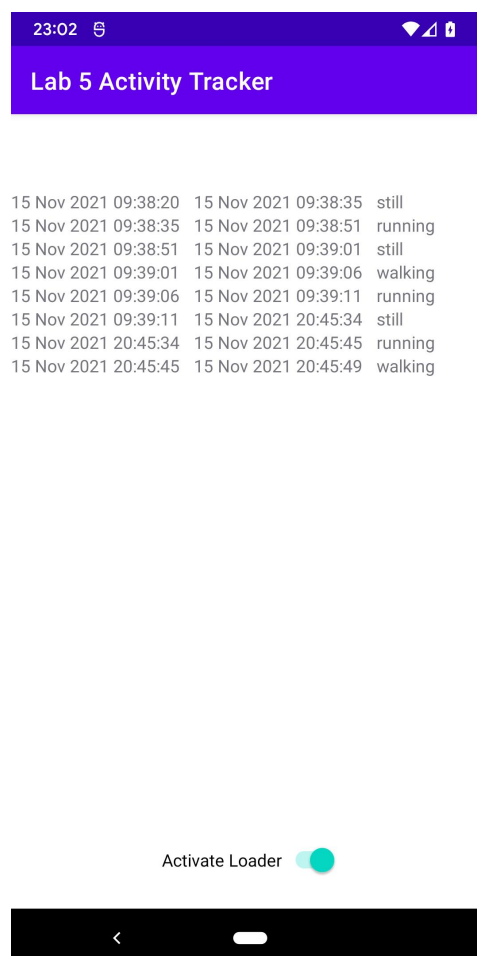
In this case, we encapsulate our main functionality in the `try` block, to detect errors. If we detected an error, then we print the error out to the console for debugging. In either case, in our `finally` block, we hide the loading progress bar, and clear the loading flag.

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.getContext());

    DateFormat fmt = DateFormat.getDateInstance();

```

```

    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>
```