

EXPERIMENT 1

AN INTRODUCTION TO THE INTERNET OF THINGS (IoT)

DOCUMENT INFORMATION	
COURSE TITLE	ELECTRONICS DESIGN II
COURSE CODE	ELET2415
DATE	
STUDENT ID	

DOCUMENT LAST REVISED
WINTER 2024

Blank Page

ABOUT THIS EXPERIMENT:

Experiment 1 – An Introduction to the Internet of Things Systems – Example 1. This laboratory activity serves as an introduction to the fundamentals of IoT architecture and development. Students are tasked with engaging in a series of hands-on activities, that involves the utilization of development tools, encompassing both Hardware and Software. These tools play an essential role in the design, implementation, and debugging of embedded systems and technologies, including those related to the Internet of Things (IoT). It is important to acknowledge that some information in this document has been sourced from various references, including but not limited to, **Yana Electronics** (<https://www.yanatronics.com/>), **Microchip Technology** (<https://www.microchip.com/>) and **Arduino** (<https://www.arduino.cc/>).

EXPERIMENT OBJECTIVES:

- ☐ Garner a Working Knowledge of the Arduino Integrated Development Environment (IDE).
- ☐ Design, write and debug simple application programs in the Arduino IDE.
- ☐ Garner an understanding the Internet of Things (IoT) System Architecture.
- ☐ Garner a Working Knowledge of Technologies Utilized in Backend IoT Development.
- ☐ Garner a Working Knowledge of Technologies Utilized in Frontend IoT Development.

PRE-LABORATORY REQUIREMENTS:

Before attempting the laboratory activities, students are required to:

- Read through this experiment manual.
- Purchase the required course components/parts.
- Download and install the software for each section (Hardware, Backend, and Frontend), listed on the course webpage, and carry out all necessary configurations.
- Research and get a basic overview of the installed software.
- Research and get a basic overview of Arduino products. The ***Getting started with Arduino products*** tutorial, <https://www.arduino.cc/en/Guide> is recommended. Also highly recommended is a few introductory videos on the Arduino IDE and programming, the Arduino Nano, and ESP32 boards. Many are available online.

EXTREMELY IMPORTANT:

- Fork the “RNG” IoT template repository from <https://github.com/iotHub1/RNG.git> to your own GitHub account.
- Use the command line terminal (CMD terminal on windows) to clone the RNG IoT template repository from your own GitHub and carry out the instructions outlined in the **README.md** file to setup and run your application.

EQUIPMENT AND COMPONENTS:

Table 1 list the components and equipment required to carry out the laboratory activities. Computers loaded with the Arduino IDE (Integrated Development Environment) can be made available. However, students are encouraged to utilise their personal laptop and as such required to download and install the latest version of the software listed on the course webpage.

Table 1 – Equipment and Components Required for Experiment 1

<i>Equipment</i>	<ul style="list-style-type: none">• Digital Multimeter, Breadboard, Set Jumper Wires, 5V Power Supply• A Windows computer loaded with the latest version of all the required software. See the course webpage.
<i>Component</i>	<ul style="list-style-type: none">• Arduino Nano and ESP32 Development Boards

HARDWARE COMPONENT DESIGN AND IMPLEMENTATION:

Before venturing into the development of the presented IoT hardware component. Here is a quick introduction to the Arduino system. It is presented here, as Arduino boards will be the primary focus of the hardware system.

Arduino is an open-source electronics prototyping platform. The system was created at the Ivrea Interaction Design Institute and is geared towards students without a background in electronics and programming, (<https://www.yanatronics.com/arduino/>). The Arduino platform consists of two parts:

1. **A Hardware Section:** A hardware module (board), which is one of several physical hardware platforms. These boards are commonly referred to as an Arduino board. Examples include the Arduino Uno, the Arduino Nano (Figure 1), the Arduino Mega and the ESP32. The first Arduino board, the Arduino Uno, was designed and implemented utilising the ATmega328 8-bit AVR microcontroller. Today there are several Arduino boards on the market most of which are still designed and built using AVR microcontrollers. There are also boards which utilise other microcontrollers, such as the ESP32 from Espressif Systems. An overview of the more popular boards is available at (<https://www.yanatronics.com/arduino-boards-2/>).

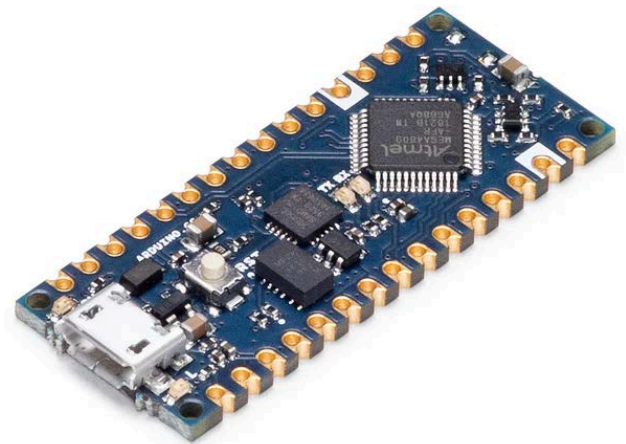


Figure 1 – An Arduino Nano

2. **A Software Section:** This is the application program or firmware that resides and runs on the microcontroller inbuilt on the Arduino board. The software coordinates the operations of the Arduino hardware platform. This software is normally written in the Arduino Integrated Development Environment (IDE) then downloaded to the microcontroller. Programs written for Arduino boards in the IDE have a structure similar to the C programming language. These programs are referred to as sketches. The structure of an Arduino program or sketch is shown in Figure 2. Every Sketch has a setup function for configurations and the initialization of libraries. The setup function runs only once. Additionally there is the loop function that runs forever. This is where the application tasks are written.

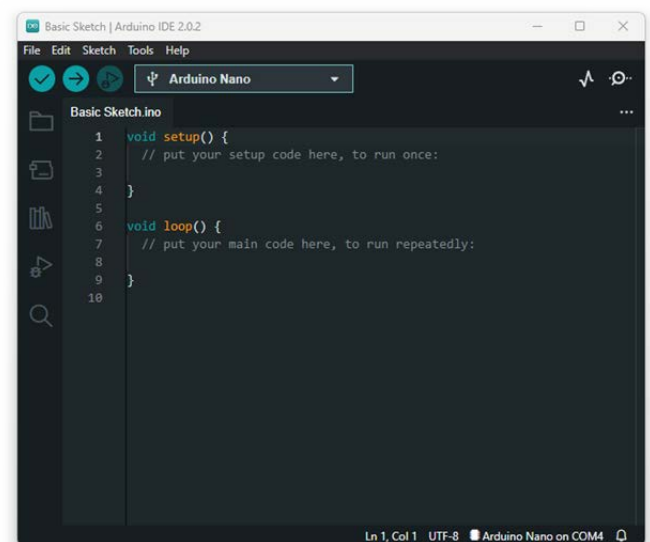


Figure 2 – Arduino Sketch

An Introduction to Arduino Digital I/O Operations

Digital inputs and outputs, on an Arduino board, are done the General-Purpose Input Output (GPIO) pins. Arduino boards are equipped with various GPIO pins, as illustrated in Figure 3 for an Arduino Nano. These GPIO can be configured for either digital input or output. The configurations and functionalities done through the use of three functions:

- ❖ **pinMode()**.Used to configure a GPIO pin as either OUTPUT or INPUT.
- ❖ **digitalRead()**.Used to read the logic value present on a GPIO pin configured as INPUT.
- ❖ **digitalWrite()**.Used to write a logic high or low value on a GPIO pin configured as OUTPUT.

Additional information of these concepts is available at <https://docs.arduino.cc/learn/microcontrollers/digital-pins>.

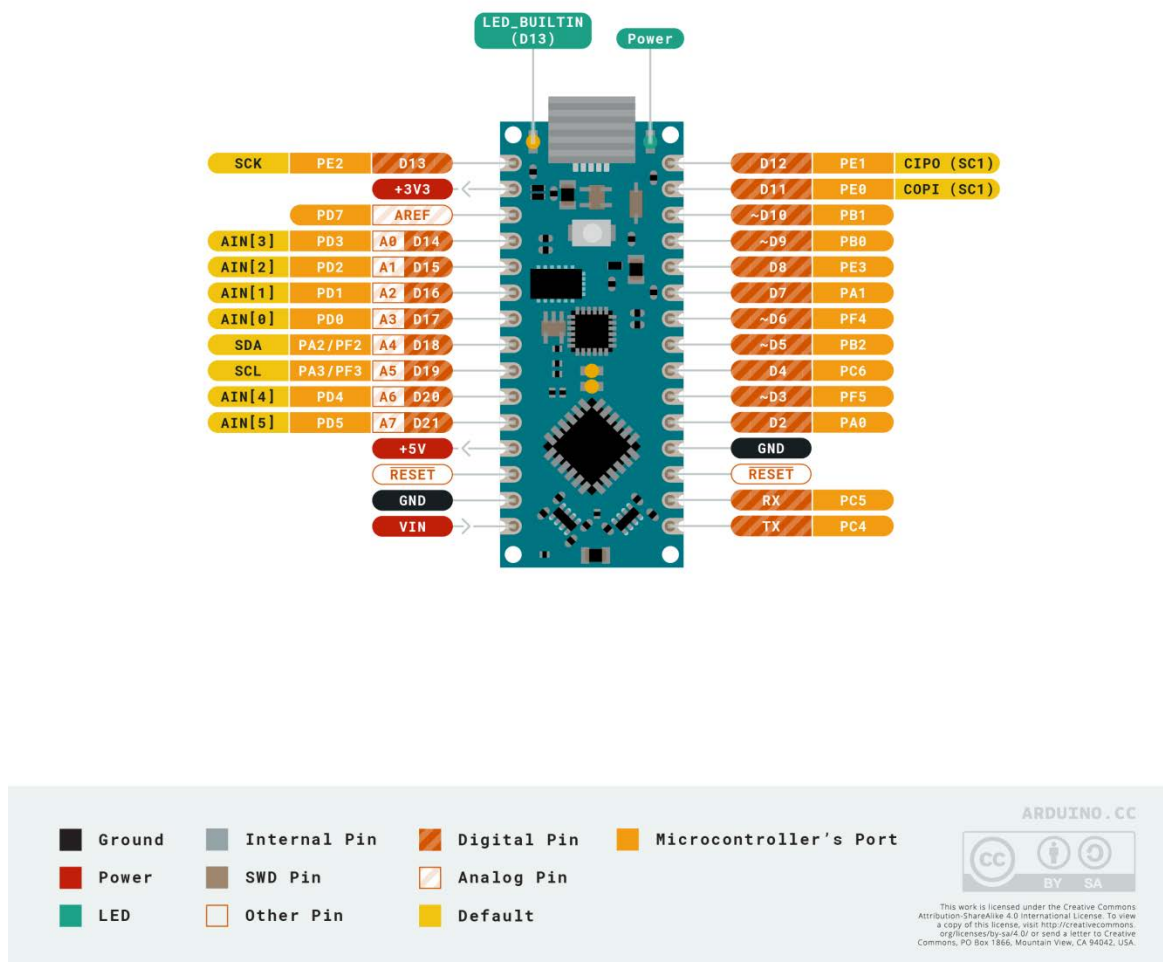


Figure 3 – Arduino Nano Board Layout

Tutorial 0: Digital I/O

In this activity, students are required to build a system that utilise a switch to control the state of an LED. The objective is to have the LED blink (ON for 2 seconds, then OFF for 2 seconds) once the switch depressed. Build the circuit outlined in Figure 4, compile and upload the provided sketch to the Arduino Nano. Depress the switch in the circuit and observe the behaviour.

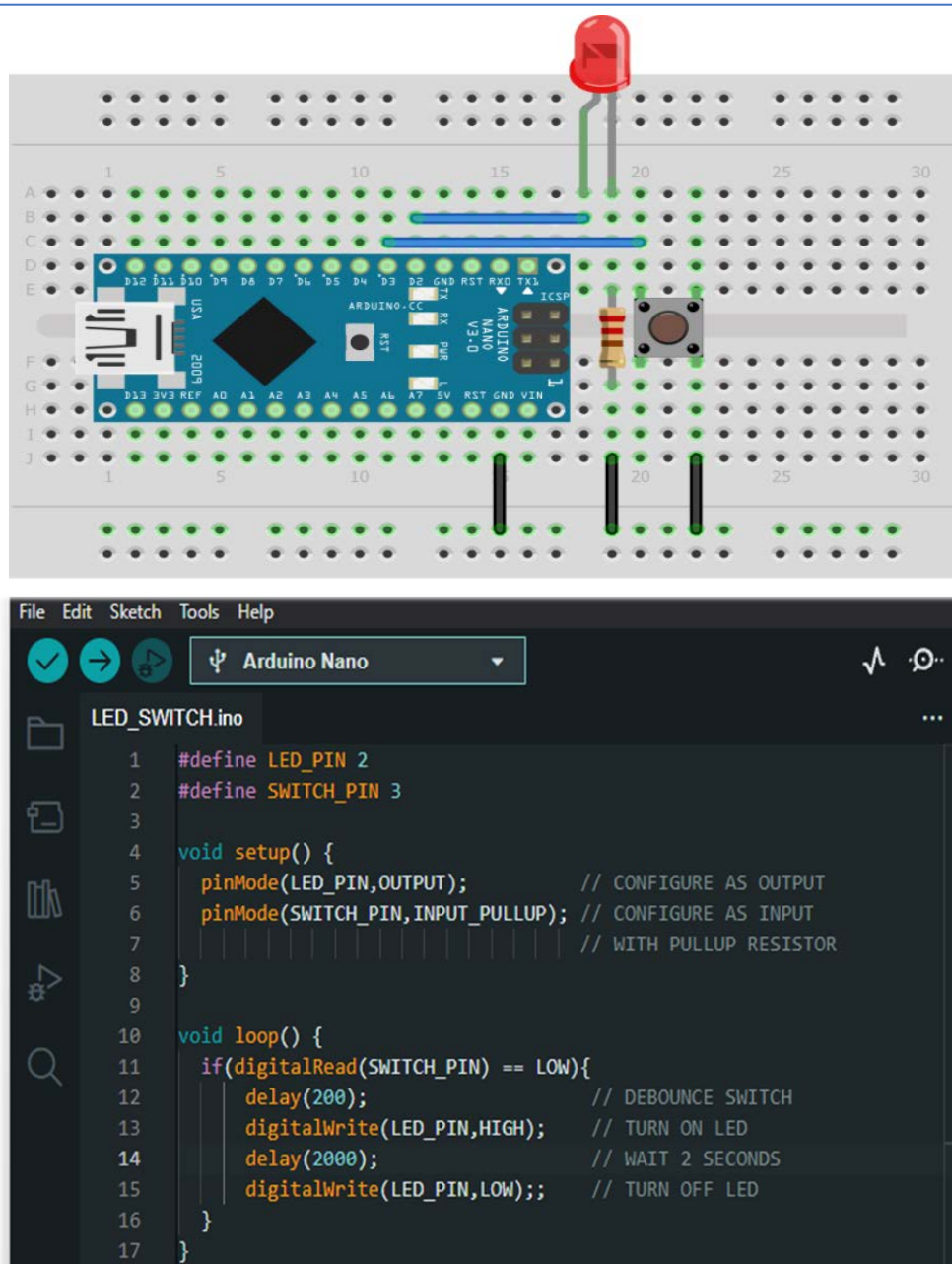
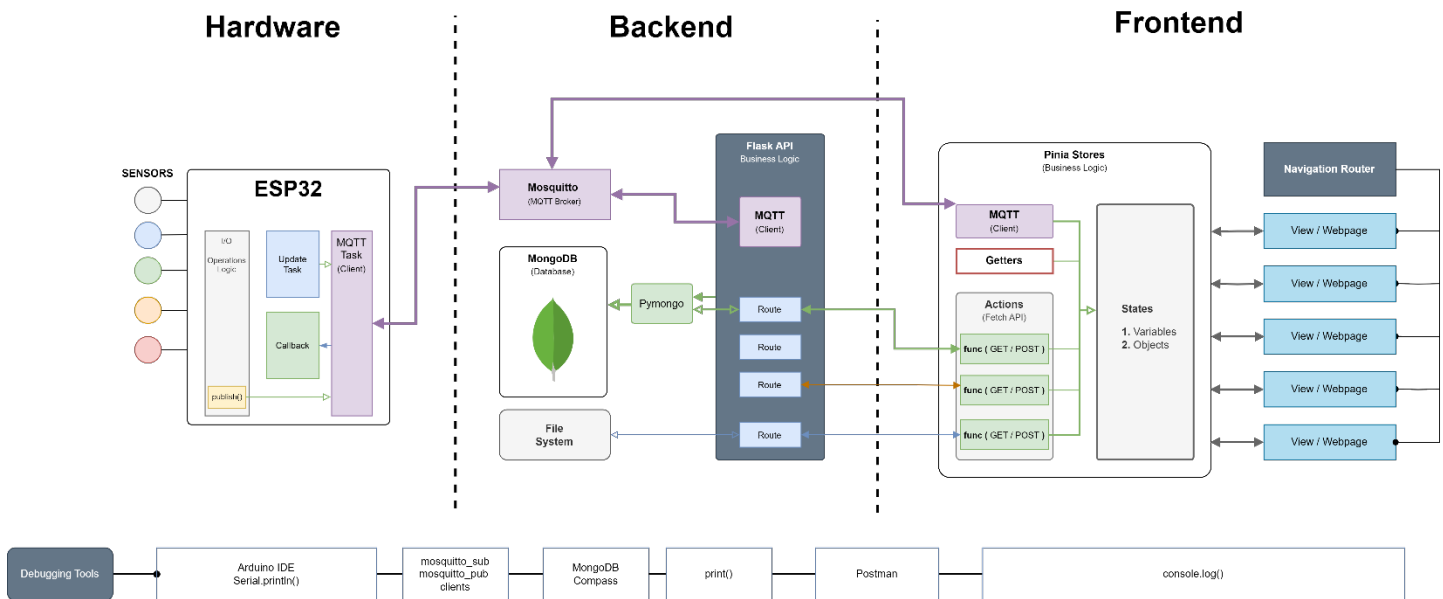


Figure 4 – Circuit and Sketch for introductory activity

IoT SYSTEM ARCHITECTURE AND IMPLEMENTATION:



In this laboratory students are assigned the task of constructing an IoT system based on the architecture illustrated in the provided system diagram above. Each of the three (3) sections comes with specifications outlining the specific tasks needed for system development.

The **Hardware** component has two tasks: (1) To generate a random integer on the press of a button, which is then displayed on a seven-segment display. In addition, this integer should be published to a MQTT topic, see the system diagram. Note that this topic is accessible to both the backend and frontend segments of the system. (2) Process, messages received from subscribed MQTT topics, and executes the requested action.

The **Backend** component also has two tasks: (1) Store published data from the hardware component in a database. This database should adhere to a specified schema. (2) Facilitate the access of the stored data by the frontend. This should be accomplished through the implementation of API routes.

The **Frontend** component is tasked with: (1) Displaying the most recently generated random integer published by the hardware on a webpage. (2) Provide a user interface (UI) featuring buttons to manipulate the state of two LEDs controlled by the hardware. (3) Another webpage, which should integrate a visual representation, such as a chart or graph, to depict the frequency distribution of each randomly generated integer stored in the database. (4) Should incorporate an interface, presented in a card format, to fetch and display a count of the instances when each LED was activated.

Activity 1 - Implementing the IoT Hardware Component:

Now let's move on to implementing the hardware components of the IoT system. The hardware component specifications are as follows, note that the first two specifications have already been completed for your convenience. Note: The starter code for the hardware component is contained in the **hardware.ino** file located in the RNG/hardware folder for the repository you cloned.

- ✓ Implement the Network Time Protocol (NTP). Which will allow for the system to maintain time with a one second accuracy.
- ✓ Implement the MQTT protocol for bi-directional communications between **Hardware**, **Backend** and **Frontend** sections.
- ☐ Display the integer '8', on the seven-segment display on system start-up.
- ☐ On a button press:
 - Generate a random single digit integer in the range [0,9].
 - Display the generated integer on the seven-segment display.
 - Publish the generated integer to the backend and frontend according to the following schema.
`{ "id": "student_id", "timestamp": 1702212234, "number": 9, "ledA": 0, "ledB": 0 }`
- ☐ Process and action messages published by frontend with the `{ "type": "toggle", "device": "LED A" }` or other similar schemas.

Learning objectives:

- ☐ Establishing a Connection with a Wi-Fi Network.
- ☐ Configuring a MQTT Client.
- ☐ Working with JSON on Arduino.

Useful resources:

C/C++ : <https://www.w3schools.com/cpp/default.asp>.

MQTT: <https://www.youtube.com/watch?v=k5o0DiO6ny4>

In addition, students should have a working understanding of JSON objects in Arduino. This is necessary as the MQTT clients utilized across the IoT system will send and receive messages in the form of JSON strings. A comprehensive guide is available in the ArduinoJson documentation, accessible at <https://arduinojson.org/>.

It is recommended that you begin with the serialization and deserialization examples and tutorials provided in the quick start section.

Before proceeding with the implementation of the remaining hardware specifications, it is critical to adjust the **hardware/hardware.ino** code to incorporate specific configuration details. Specifically: (1) Updating the Wi-Fi credentials to enable internet connectivity for the ESP32 on any accessible network. (2) Edit the MQTT client configuration settings to include the necessary server and port details, and the default topic for hardware publishing, to include all relevant topics for hardware subscription. Some guidance can be obtained from the ‘**Wi-fi and MQTT configuration on Arduino**’ in the Appendix.

Students are required to interface a seven-segment display with an ESP32 board. Refer to the datasheet for your specific seven-segment display for the relevant pinout information. The pinout information for the ESP32 board is provided in the Appendix.

- A. Design a circuit in KiCAD employing the ESP32, two (2) LEDs, one (1) tactile switch and a 7-segment display. Ensure that the connections of the seven-segment display to the ESP32 board are established via the following designated pins: 15, 32, 33, 25, 26, 27, 14, and 12. Construct this circuit on a breadboard.

[Required Effort Level 4.0]

- B. Complete the definitions section for all pins mapped to the seven-segment display. [Required Effort Level 2.0]

```
hardware.ino
// DEFINE THE PINS THAT WILL BE MAPPED TO THE 7 SEG DISPLAY BELOW, 'a' to 'g'
#define a      15
/* Complete all others */
```

- C. Define variables for each LED and buttons. [Required Effort Level 1.0]

```
hardware.ino
// DEFINE VARIABLES FOR TWO LEDs AND The BUTTON
/* Example. Define a variable name 'LED_A' and assign it to pin 4 on the microcontroller */
#define LED_A 4
/* Complete for LED_B, BTN_A*/
```

- D. In the setup function of the Arduino hardware sketch, finalize the pin configurations. Configure all seven-segment and LED designated pins as output. Ensure that all buttons are configured as input with pull-up resistor enabled. [Required Effort Level 3.0]

```
hardware.ino
void setup() {
  Serial.begin(115200); // INIT SERIAL

  // CONFIGURE THE ARDUINO PINS OF THE 7SEG AS OUTPUT
  pinMode(a,OUTPUT);
  /* Configure all others here */

  initialize();          // INIT WIFI, MQTT & NTP
  vButtonCheckFunction(); // UNCOMMENT IF USING BUTTONS
}
```

- E. Complete all utility functions according to the hardware specifications. Ensure all functions are declared. [Required Effort Level 3.5 + 2.5 + 2.5 + 2.0 + 2.5 + 5.0]

```
hardware.ino
//#####
//#                               UTIL FUNCTIONS                               #
//#####

void vButtonCheck( void * pvParameters ){
  // CHECK FOR AND ACTION BUTTON EVENTS
}
void Display(unsigned char number){
  // WRITE INTEGER TO 7-SEG DISPLAY
}
int8_t getLEDStatus(int8_t LED) {
  // RETURNS THE STATE OF A SPECIFIC LED. 0 = LOW, 1 = HIGH
}
void setLEDState(int8_t LED, int8_t state){
  // SETS THE STATE OF A SPECIFIC LED
}
void toggleLED(int8_t LED){
  // TOGGLES THE STATE OF SPECIFIC LED
}
void GDP(void){
  // GENERATE, DISPLAY THEN PUBLISH INTEGER
}
```

Concerning the GDP function, it is imperative that messages published by the hardware strictly follow the JSON string format. To accomplish this, your initial step is to construct a JSON document, incorporating all the necessary key-value pairs in accordance with the specified schema outlined in the hardware specifications. Following this, generate a C++ char array to store the JSON string after serializing your JSON document. Finally, publish the formatted message to the intended topic.

- F. The function outlined in the provided code block serves as a callback function triggered each time a message is received on a topic subscribed to by the microcontroller. This function is partially implemented in the hardware.ino file. Your task is to finalize the function to handle and respond to messages originating from either the backend or frontend in accordance with the defined schemas within the system specifications.

The MQTT messages arriving follow a JSON string format and must be transformed into JSON objects for further processing. Ensure that the completion of this function aligns with the specified message structures outlined in the relevant sections of the system specifications. **[Required Effort Level 4.0]**

```
hardware.ino
void callback(char* topic, byte* payload, unsigned int length) {
    // ##### MQTT CALLBACK #####
    // RUNS WHENEVER A MESSAGE IS RECEIVED ON A TOPIC SUBSCRIBED TO

    Serial.printf("\nMessage received : ( topic: %s ) \n",topic);
    char *received = new char[length + 1] {0};

    for (int i = 0; i < length; i++) {
        received[i] = (char)payload[i];
    }

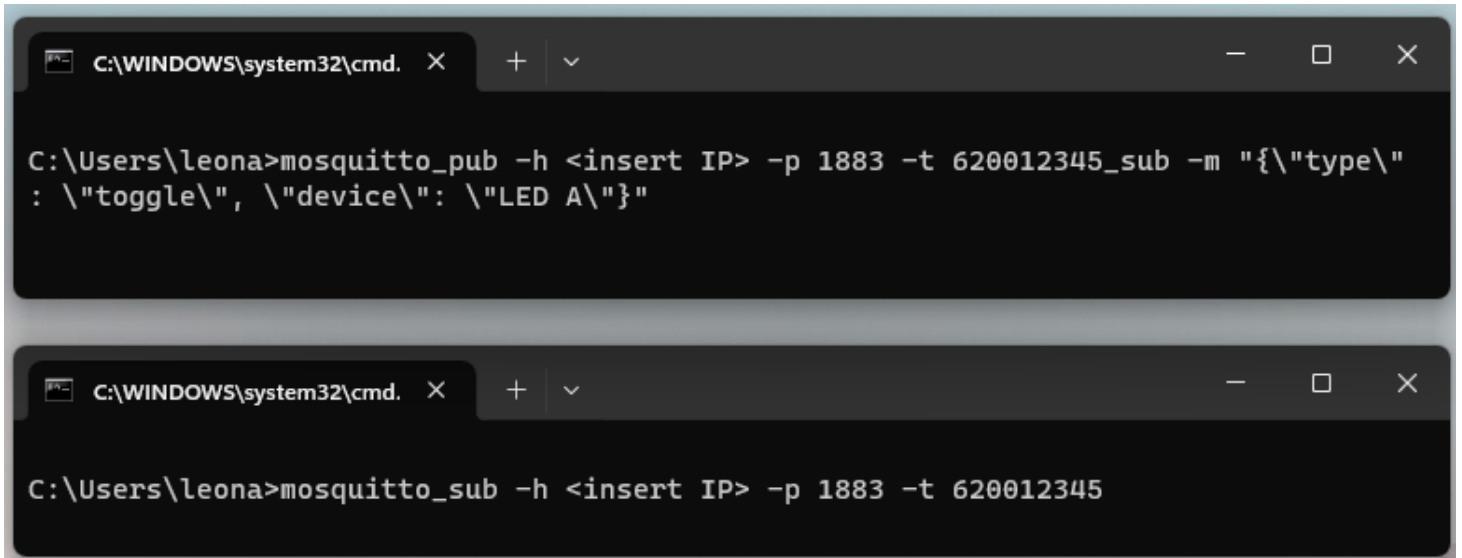
    // PRINT RECEIVED MESSAGE
    Serial.printf("Payload : %s \n",received);
}
```

This would be a great time to commit and push your code to your GitHub account.

Open a command line terminal in your project folder (**RNG/**) and then run the following five (5) commands. Ensure you push your code to GitHub as often as possible. You should also add your own custom commit messages whenever you make a commit.

```
CMD terminal
git status
git add .
git status
git commit -m "Hardware Section Complete"
git push
```

Confirm Task Completion: Have a demonstrator check the operation of the completed hardware component. Completion of each specification should be verified. The MQTT operations can be verified using local mosquitto clients. See the figure below on how to create a MQTT publish and subscribe client. Replace the <insert IP> with the IP address or domain name of the MQTT broker.



The image displays two separate Windows command prompt windows. The top window shows the command `mosquitto_pub -h <insert IP> -p 1883 -t 620012345_sub -m "{\"type\": \"toggle\", \"device\": \"LED A\"}"` being entered at the prompt `C:\Users\leona>`. The bottom window shows the command `mosquitto_sub -h <insert IP> -p 1883 -t 620012345` being entered at the prompt `C:\Users\leona>`. Both windows have a title bar indicating the path `C:\WINDOWS\system32\cmd.`

```
C:\WINDOWS\system32\cmd. X + v - □ X  
C:\Users\leona>mosquitto_pub -h <insert IP> -p 1883 -t 620012345_sub -m "{\"type\"  
: \"toggle\", \"device\": \"LED A\"}"  
  
C:\WINDOWS\system32\cmd. X + v - □ X  
C:\Users\leona>mosquitto_sub -h <insert IP> -p 1883 -t 620012345
```

BACKEND COMPONENT DESIGN AND IMPLEMENTATION:

The specifications for the backend component are outlined below. Although a significant portion of the backend specifications is already provided, the main objective is for students to undertake independent research. This entails exploring the introduced technologies such as the Pymongo driver, MongoDB CRUD & Aggregation operations, Flask API, MQTT, etc., to deepen one's comprehension of the IoT system's backend. The emphasis is on personal initiative in gaining insight and proficiency in the backend technologies. **Students' understanding of these technologies will be evaluated during lab demonstrations, contributing to the overall grade.**

- ✓ Insert updated messages published by hardware to mongoDB database using CRUD function.
- ✓ Create a route to return the frequency of each randomly generated number inserted into the database.
- ✓ Create a route(s) that returns a count of the number of times a specific LED was turned on.

Learning objectives:

- ☐ Configuring the Backend
- ☐ Configuring the MQTT client for connecting to broker, subscribing to topics and processing received messages.
- ☐ Create API routes for GET request, which takes no argument(s).
- ☐ Create API routes for GET request, which takes argument(s).
- ☐ Create API routes for POST request.
- ☐ **Learn MongoDB CRUD and Aggregation operations online at MongoDB University.**
- ☐ Use MongoDB Compass for writing and exporting Aggregation pipelines to Python code for use with the Pymongo Driver.

The backend component is **exclusively** implemented in Python. Consequently, prior to undertaking the upcoming activity, it is imperative for students to revisit essential Python concepts. This review should encompass, but is not limited to, working with **Variables, Data types, Operators, List, Tuples, Dictionaries, Functions, If-Else, Classes/Objects, JSON, Loops, and Try-Except**. Additional foundational Python concepts can be revisited at <https://www.w3schools.com/python/default.asp>.

Should any guidance be needed, please feel free to contact your laboratory personnel before the due date of the labs.

Useful links:

Python Flask: https://www.youtube.com/watch?v=7M1MaAPWnYg&list=PLB5jA40tNf3vX6Ue_ud64DDRVSryHHr1h

Very Important

Prior to advancing to the subsequent tasks, it is crucial to implement the necessary backend and MQTT configurations. Furthermore, establish a local MongoDB user account specifically for your Flask API. For guidance, please refer to the instructions outlined in the Appendix.

Activity 2 - Creating API Routes

In Flask, the process of establishing routes for a RESTful API entail specifying functions that manage distinct HTTP methods (e.g., GET, POST, PUT, DELETE) at designated endpoints. As your API becomes more complex, you may want to explore features like request parsing, response formatting, and route organization. Flask provides extensions and best practices to handle these aspects efficiently. Students are required reference the flask documentation as needed at <https://flask.palletsprojects.com/en/stable/>.

Before moving to the next task, ensure to read and become familiar with the flask documentation for Routing at <https://flask.palletsprojects.com/en/stable/quickstart/#routing>.

All routes must be defined in the **backend/app/views.py** file. In this activity you will be introduced to three (3) routes that will implement the remaining backend specifications. If not previously done, edit the file to include these routes. Note that all routes begin with **'/api/'**.

1. The code block below defines a route for handling HTTP GET requests at the specified path **'/api/numberfrequency'**. The **get_numberFrequency()** function will be executed when a GET request is made to this route/endpoint. This route returns the frequency of each randomly generated number stored in the database and responds with a JSON object indicating the status of the operation (whether it succeeded or failed) along with the relevant data. The code includes error handling to print any exceptions that might occur during the process.

```
views.py
@app.route('/api/numberfrequency', methods=['GET'])
def get_numberFrequency():
    '''Returns list of frequency'''
    if request.method == "GET":
        try:
            frequency = mongo.numberFrequency()
            if frequency:
                return jsonify({"status": "found", "data": frequency})

        except Exception as e:
            print(f"get_numberFrequency error: {str(e)}")
    return jsonify({"status": "failed", "data": []})
```

2. The code block below defines a route for handling HTTP GET requests at the specified path **'/api/oncount/<ledName>'**. The **get_onCount(ledName)** function will be executed when a GET request is made to this route. It takes a single parameter/argument **'ledName'** from the URL. This route fetches, from the database, a count representing the amount of time a specific LED was turned on. It responds with a JSON object indicating the status of the operation (whether it succeeded or failed) along with the relevant data. The code includes error handling to print any exceptions that might occur during the process.

```
@app.route('/api/oncount/<ledName>', methods=['GET'])
def get_onCount(ledName):
    '''Returns number which represents the amount of time a specific LED was turned on'''
    if request.method == "GET":
        try:
            LED_Name = escape(ledName)
            count = mongo.onCount(LED_Name)
            if count:
                return jsonify({"status": "found", "data": count})
        except Exception as e:
            print(f"get_onCount error: {str(e)}")
    return jsonify({"status": "failed", "data": 0})
```

3. The code block below defines a route for handling HTTP POST requests at the specified path `/api/oncount`. The `get_onCountPOST()` function will be executed when a POST request is made to this route. This route extracts the LED name from the form embedded in the request, then fetches, from the database, a count representing the amount of time that specific LED was turned on. It responds with a JSON object indicating the status of the operation (whether it succeeded or failed) along with the relevant data. The code includes error handling to print any exceptions that might occur during the process.

```
@app.route('/api/oncount', methods=['POST'])
def get_onCountPost():
    '''Returns number which represents the amount of time a specific LED was turned on'''
    if request.method == "POST":
        try:
            form = request.form

            LED_Name = escape(form.get("LED_Name"))
            count = mongo.onCount(LED_Name)
            if count:
                return jsonify({"status": "found", "data": count})

        except Exception as e:
            print(f"get_onCountPost error: {str(e)}")
    return jsonify({"status": "failed", "data": 0})
```

These API route examples serve multiple purposes. In addition to being straightforward solutions, regard them as instructive models and templates. Their primary aim is to guide and support you in crafting your own routes for future lab experiments. It is recommended that students supplement their learning of these concepts by reading through the relevant sections of the Flask documentation.

The previously established routes, incorporated various MongoDB CRUD (*create*, *read*, *update*, and *delete*) operations, alongside aggregation functions, as seen in the code block below. These operations and functions are defined within the `backend/app/functions.py` file. This file houses a Python class named `DB`, serving as the container for all database querying functions. It is imperative to define all database-related functions within this class. Once these functions are defined, they are accessible through the routes/endpoints you will be creating in the `backend/app/views.py` file as seen in previous code blocks.


```
def numberFrequency(self):
    '''RETURNS A LIST OF OBJECTS. EACH OBJECT CONTAINS A NUMBER AND ITS FREQUENCY'''
    try:
        remotedb = self.remoteMongo('mongodb://%s:%s@%s:%s' % (self.username, self.password, self.server, self.port), tls=self.tls)
        result = list(remotedb.ELET2415.update.aggregate([ { '$group': { '_id': '$number', 'frequency': { '$sum': 1 } } }, { '$sort': {
'_id': 1 } } }, { '$project': { '_id': 0, 'number': '$_id', 'frequency': 1 } } ]))
    except Exception as e:
        msg = str(e)
        print("numberFrequency error ",msg)

    else:
        return result

def onCount(self,LED_Name):
    '''RETURN A COUNT OF HOW MANY TIME A SPECIFIC LED WAS TURNED ON'''
    try:
        remotedb = self.remoteMongo('mongodb://%s:%s@%s:%s' % (self.username, self.password, self.server, self.port), tls=self.tls)
        result = remotedb.ELET2415.update.count_documents({LED_Name:{"$eq":1}})
    except Exception as e:
        msg = str(e)
        print("onCount error ",msg)

    else:
        return result
```

Introduction to MongoDB database.

Create an account at mongoDB University then register for the Introduction to MongoDB online course at <https://learn.mongodb.com/learning-paths/introduction-to-mongodb>. The course on MongoDB Introduction provides comprehensive guidance on essential skills and knowledge required for initiating your journey with MongoDB. This encompasses aspects such as establishing connections to MongoDB databases, performing basic CRUD operations, and delving into crucial subjects like aggregation, indexing, data modeling, and transactions.

Once registered, you are required to complete the following units:

- ❖ *Start Here - Intro to MongoDB*
- ❖ *Getting Started with MongoDB Atlas*
- ❖ *MongoDB and the Document Model*
- ❖ *MongoDB CRUD Operations: Insert and Find Documents*
- ❖ *MongoDB CRUD Operations: Replace and Delete Documents*
- ❖ *MongoDB CRUD Operations: Modifying Query Results*
- ❖ *MongoDB Aggregation*
- ❖ *MongoDB Indexes*



Upon completing the online course, it is mandatory to visit the Electronics lab one day before the scheduled deadline of this lab experiment. During this session, your lab demonstrator will guide you through the procedure of creating and exporting an Aggregation pipeline in MongoDB Compass. Additionally, they will demonstrate how to use the exported pipeline, to create your own database querying functions using PyMongo.

As you progress in this course, familiarizing yourself with the following aggregation and CRUD operations is imperative for successfully completing your lab assignments. Additionally, it is a requisite to thoroughly read through the official MongoDB documentation at <https://www.mongodb.com/docs/manual/crud/> and <https://www.mongodb.com/docs/manual/reference/operator/aggregation-pipeline/> as needed.

While mongoDB CRUD and Aggregation operations are explained in the official documentation as well as in the online short course at mongoDB University. The backend section uses Pymongo driver to interface with the database. PyMongo is a Python package equipped with utilities for interacting with MongoDB and stands as the recommended approach for Python-based MongoDB interactions. Its documentation found at <https://pymongo.readthedocs.io/en/stable/index.html> attempts to explain everything you need to know to use **PyMongo**.

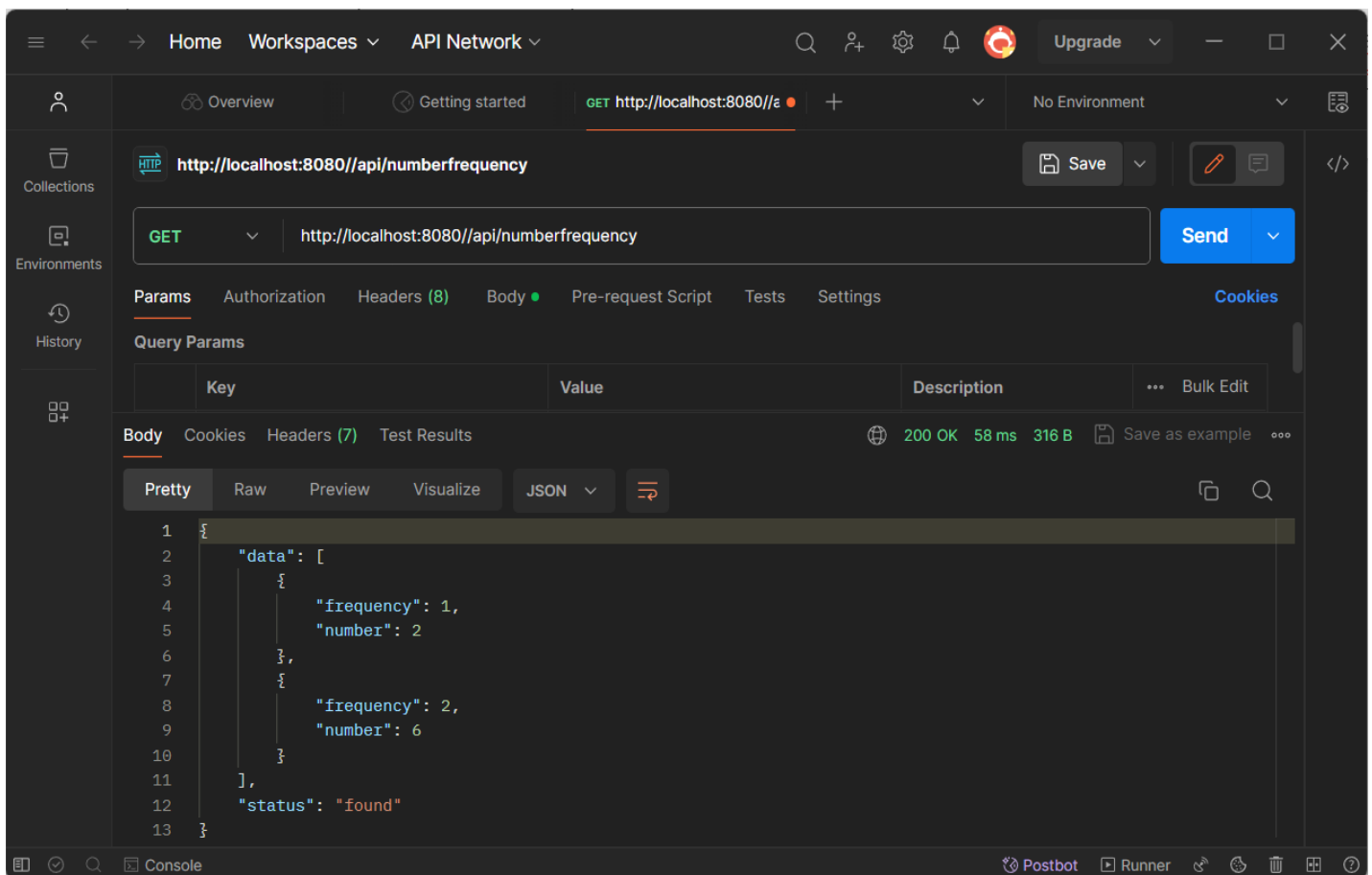
All CRUD and Aggregation operations outlined above are implemented in PyMongo. This means that the theoretical concepts are covered by the mongoDB documentation as well as in the online short course while the

implementation is done using the PyMongo Python driver and therefore you must become familiar with all three.

The tutorial found at <https://pymongo.readthedocs.io/en/stable/tutorial.html> serves as an introduction to working with **MongoDB** and **PyMongo**. Find details on all CRUD and Aggregation operations at <https://pymongo.readthedocs.io/en/stable/api/pymongo/collection.html#pymongo.collection.Collection>.

This would be a great time to commit and push your code to your GitHub account.

Confirm Task Completion: Have a demonstrator check the operation of the completed backend component. Completion of each specification should be verified. The routes operations can be verified using Postman, see example below.



FRONTEND COMPONENT DESIGN AND IMPLEMENTATION:

Webpage 1: route name Intro

- ☐ Display the most recent randomly generated number published by hardware in update message.
- ☐ Add card for each LED that should be controlled by the webpage. Card must use icons to indicate the on/off state of the LED as well as a button to toggle the state of said LED. On button click, publish toggle message with `{ "type": "toggle", "device": "LED A" }` schema to hardware.

Webpage 2: route name Graph

- ☐ Graph the frequency of each number using a bar chart. Include button that will refresh graph according to all the data present in the database whenever the button is clicked.
- ☐ Display a count of how many times each LED was turned on. Include button that will update this count according to all the data present in the database whenever the button is clicked.

Learning objectives:

- ☐ Creating new webpage from Vue js template.
- ☐ Learn how to create responsive webpage layouts using the Grid system.
- ☐ Adding Vuetify components to your layout to create the desire user interface (UI)
- ☐ Configuring MQTT Paho client in the `frontend/src/store/mqttStore.js` file for connecting to broker, subscribing to topics, publishing to topics and processing received messages.
- ☐ Learn how to create and use Vuejs `ref` variables.
- ☐ Learning how to use Vuejs `v-if` and `v-else-if` directives.
- ☐ Using `@click` events to call JavaScript functions.
- ☐ Creating functions, in the `frontend/src/store/appStore.js` file, for making API calls to the backend API routes created in activity 2, using the Fetch API. Subsequently export these functions so that they can be used throughout the frontend.
- ☐ Plotting data stored in our database using Chart.js library.

The Frontend will be developed using HTML, CSS and JavaScript. Therefore, while attempting the following activity, it is essential for students to review fundamental of these technologies as needed, which must include, but not be limited to,:

HTML

Basic, Elements, Attributes, Styles, Comments, Images, Tables, List, Div, Classes, Id, Forms found at <https://www.w3schools.com/html/default.asp>

JavaScript

working with Variables, Data types, Comments, Let, Const Operators, Assignment, Functions, Arrow functions, Events, String methods, String templates, Numbers, Number methods, Arrays, Array methods, Dates, Date formats, If-Else, Objects, JSON, and Loops found at <https://www.w3schools.com/js/default.asp>

CSS

Working with Margins, Paddings, Selectors, width, Height, Combinators, Color, Backgrounds, Text, Align, Fonts at <https://www.w3schools.com/css/default.asp>

If you require any guidance, do not hesitate to contact your lab demonstrators before labs are due.

Activity 3.1 - Create View/Webpage

[Required Effort Level 2.0 + 2.0 + 2.0]

Create a new View for the 'Intro' navigation route. Refer to the appendix section of this lab manual on "Creating Views/Webpages for Navigation Routes". Note the Appendix goes through the process of creating a navigation route for a view called Game as an example.

Now that we have a new blank webpage, let's continue other modifications to implement the front-end specifications for webpage 1: route name Intro.

Splitting the computer's display in two showing the text editor and the web browser is recommended at this point. This allows for the viewing of the webpage as it is being built. This also allows for a better understanding of how each line of code changes the webpage. Also try clicking on the icon in the top right corner of the webpage, this will change the webpage display to dark mode.

Tasks:

- Display the most recent random generated number published by the hardware component.
- Add card for each LED that should be controlled by the webpage. Card must use icons to indicate the on/off state of the LED as well as a button to toggle the state of said LED. On button click, publish toggle message with `{ "type": "toggle", "device": "LED A" }` schema to hardware for that specific LED.

These are accomplished through three (3) activities.

1. Creating the UI layout in template section using grid system and add the required CSS styling to the style section.
2. Add desired Vuetify components found at <https://vuetifyjs.com/en/components/all/>.

3. Add business logic to the webpage by adding JavaScript code to the script section and the corresponding store found in src/store to update all state variables and objects using the data received from both hardware (via MQTT) and backend (via API calls).

The steps above are general, and should be applied, in order to effect the frontend specifications given throughout the course.

Activity 3.2 - Create UI Layout

[Required Effort Level 6.0]

1. Build a layout using grid system in **<template>** section. Visit <https://vuetifyjs.com/en/components/grids/> for more information and examples on creating layouts using the grid system. We will also add our CSS style in the **<style>** section. We recommend learning CSS at <https://www.w3schools.com/css/default.asp> as needed. **Remove the <div> element from the template before continuing.**
2. Add a container using the **<v-container></v-container>** tag. Set the **class** attribute to **'container'** we will use these class names to add CSS style to our webpage.
 - Create a `.container{}` object in **<style>** section. Set the height to 100% and give it a border of 1px solid blue. Refresh the webpage if the change doesn't reflect automatically.
 - Add two (2) rows inside the container using the **<v-row> </v-row>** tag. Each row must have a class of 'row' and it's justify property set to 'center'. Furthermore, the first row should have an additional class of "bg-surfaceVariant". to set the background color for that specific row.
 - Create a `.row{}` object in **<style>** section. Set the width to 100%, padding to 10px, top and bottom margin to 10px, left and right margin to 0px and give it a border of 1px solid purple. Again, refresh the webpage if the changes don't reflect automatically.
 - Add two (2) columns to the first row using the **<v-col></v-col>** tag. Set the class attribute of first column to 'col col1' and the 'col col2' for the second column.
 - Add one (1) column to the second row using the **<v-col></v-col>** tag. Set the class attribute of this column to "bg-black col col3".
 - Set the align property for all columns to "center" and the justify property for all rows to "center".
 - In the **<style>** section :
 - Create a `.col` class style. Set the border to 1px solid green
 - Create a `.col1` and `.col2` styles. Set the max-width property to 200px. This can be done with shorthand like this `.col1, .col2{}`
 - Create a `.col3` class style. Set the following properties. Height: 320px, max-width to 270px and a border of 2px solid lightslategray.

- Create a `.digit` class style with the following properties: font-family: 'Noto Sans Symbols 2' and font-size: 250px and import `url('https://fonts.googleapis.com/css2?family=Noto+Sans+Symbols+2&display=swap')`.

The `<template>` and `<style>` sections should now reflect the code blocks below.

Intro.vue

```
<template>
  <v-container class="container">
    <v-row class="row" justify="center">
      <v-col class="col col1" align="center"> </v-col>
      <v-col class="col col2" align="center"> </v-col>
    </v-row>

    <v-row class="row" justify="center">
      <v-col class="bg-black col col3" align="center"> </v-col>
    </v-row>
  </v-container>
</template>
```

Intro.vue

```
<style scoped>
  /** CSS STYLE HERE */
  @import url('https://fonts.googleapis.com/css2?family=Noto+Sans+Symbols+2&display=swap');

  .digit {
    font-family: 'Noto Sans Symbols 2';
    font-size: 250px;
  }

  .container{
    height: 100%;
    border: 1px solid blue;
  }

  .row{
    width: 100%;
    margin: 10px 0px;
    padding: 10px;
    border: 1px solid purple;
  }

  .col{
    border: 1px solid green;
    margin: 0px 10px;
  }

  .col3{
    max-width: 270px;
    height: 320px;
    border: 2px solid lightslategray;
  }

  .col1, .col2{
    max-width: 200px;
  }
</style>
```

Activity 3.3 - Add Vuetify Components

[Required Effort Level 8.0]

Vuetify comes with a suite of components that we can use to build and style webpages. While completing this and all future lab experiments, students are required to visit <https://vuetifyjs.com/en/components/all/> for documentation and examples on how to implement and customize any component use.

We will be adding a Card component to each column in the first row as well as a Button and a few Icon components to each Card. First, we will create a Card for LED A.

Add a Card component to the first column in the first row using the `<v-card></v-card>` component. Set the following properties:

- title : LED A
- width : 150
- density : compact
- rounded : md
- set border property
- set flat property

Add a `<v-divider></v-divider>` component and two (2) `<v-card-item></v-card-item>` elements inside the Card. Your card should reflect the code block below.

```
<v-card title="LED A" width="150" density="compact" border flat rounded="md">
  <v-divider></v-divider>
  <v-card-item></v-card-item>
  <v-card-item></v-card-item>
</v-card>
```

Intro.vue

Add two (2) icons to the first `<v-card-item></v-card-item>` inside the card using a `<v-icon></v-icon>` component.

Set the size property of each icon to “50”

Set the icon property of the icon components to “mdi:mdi-lightbulb” and “mdi:mdi-lightbulb-on” respectively.

Set the color property of the second icon to “yellow”

Add one (1) Button component to the second `<v-card-item></v-card-item>` using the `<v-btn></v-btn>` component and give it a class of “text-caption”. Set the following properties for the button:

- text : Toggle
- variant : tonal
- color : primary
- density : compact

The card component for LED A should now reflect the code block below.


```

<v-card title="LED A" width="150" density="compact" border flat rounded="md">
  <v-divider></v-divider>

  <v-card-item>
    <v-icon size="50" icon="mdi:lightbulb"></v-icon>
    <v-icon size="50" icon="mdi:lightbulb-on" color="yellow"></v-icon>
  </v-card-item>

  <v-card-item>
    <v-btn class="text-caption" text="Toggle" variant="tonal" color="primary" density="compact" ></v-btn>
  </v-card-item>
</v-card>

```

Create 7 Seg HTML component.

Add a `` element to the column in the second row and set its class attribute to “digit text-error”. In this `` insert nine (9) other `` elements, one for each 7 Seg digit. The table below contains all the required Unicode value for each digit. In each of the nine (9) `` insert the Unicode value for the respective digit. For example, `🯰` for the first digit.

Digit	HTML Unicode
0	🯰
1	🯱
2	🯲
3	🯳
4	🯴
5	🯵
6	🯶
7	🯷
8	🯸
9	🯹

The column in the second row of our container should now reflect the code block below. The webpage will display the numbers simultaneously; however only one number should be displayed at a time and it must be the most recent number published by the hardware. This will be implemented in the next section where we will add the business logic to the webpage using JavaScript in the `<script>` section of the Intro.vue file.

```

<v-col class="bg-black col col3" align="center">
  <span class="digit text-error">
    <span >&#x1fbf0;</span>
    <span >&#x1fbf1;</span>
    <span >&#x1fbf2;</span>
    <span >&#x1fbf3;</span>
    <span >&#x1fbf4;</span>
    <span >&#x1fbf5;</span>
    <span >&#x1fbf6;</span>
    <span >&#x1fbf7;</span>
    <span >&#x1fbf8;</span>
    <span >&#x1fbf9;</span>
  </span>
</v-col>

```

Activity 3.4 - Implement Logic

[Required Effort Level 6.0]

This webpage will process messages published from the hardware. Any webpage that requires MQTT functionality must import and use the `MqttStore` located in `frontend/src/store/mqttStore.js`. For guidance, see the 'Frontend MQTT Configuration' in the Appendix section of this lab manual. **Ensure you request the correct server and port information from your lab demonstrator.**

Before attempting the next tasks, ensure you have read the tutorials and watch all videos, if any, from the Vuejs documentation about conditional rendering at <https://vuejs.org/guide/essentials/conditional.html>.

To display the most recent randomly generated number published by the hardware, we will use the Vuejs `v-if` & `v-else-if` directives with the number property/key (`payload.number`) of the payload object to show, on the webpage, only the number that exists in the payload variable imported from the `Mqtt` store. Remember that this variable is updated whenever the frontend `Mqtt` client received a message from a topic it subscribes to and that the format of the message stored in the payload variable will change (if now different) to reflect the schema of the received message. Please do not publish messages with different schemas to one topic.

Update the `` element found in the second row's column to reflect the code block shown below. Test the current implementation by pressing an LED button on the hardware side.

```
Intro.vue
<span class="digit text-error">
  <span v-if="payload.number == 0" >&#x1fbf0;</span>
  <span v-else-if="payload.number == 1" >&#x1fbf1;</span>
  <span v-else-if="payload.number == 2" >&#x1fbf2;</span>
  <span v-else-if="payload.number == 3" >&#x1fbf3;</span>
  <span v-else-if="payload.number == 4" >&#x1fbf4;</span>
  <span v-else-if="payload.number == 5" >&#x1fbf5;</span>
  <span v-else-if="payload.number == 6" >&#x1fbf6;</span>
  <span v-else-if="payload.number == 7" >&#x1fbf7;</span>
  <span v-else-if="payload.number == 8" >&#x1fbf8;</span>
  <span v-else-if="payload.number == 9" >&#x1fbf9;</span>
</span>
```

Similarly, we want to use the `ledA` and `ledB` keys found in the payload variable to show the appropriate icon in the Card component.

Using the `v-if` directive, located the “LED A” Card, and then update the icon components to reflect the code block below.

```
Intro.vue
<v-card-item>
  <v-icon v-if="payload.ledA == 0" size="50" icon="mdi:lightbulb"></v-icon>
  <v-icon v-if="payload.ledA == 1" size="50" icon="mdi:lightbulb-on" color="yellow"></v-icon>
</v-card-item>
```

You are required to implement a similar logic for the “LED B” Card.

Adding click event to button

Currently the button components in the cards are not functional. If pressed, nothing will happen. The purpose of these buttons is to control the LEDs on the hardware side. Once clicked, these buttons must trigger an event that will send a message to the hardware using `{ "type": "toggle", "device": "LED A" }` schema. The options for the `"device"` key in this object are `"LED A"` and `"LED B"`. This is used to inform the Hardware Component of which LED to toggle.

In the `<script>` section of the `Intro.vue` file, add a new function that will publish a message according to the above-mentioned schema, to a topic that the hardware subscribed to. The function should be named `'toggle'` and must take only one argument, which is the name of the LED to toggle.

```
const toggle = (name) => {  
  let message = JSON.stringify({ "type": "toggle", "device": name }); // Create message and convert to a json string  
  Mqtt.publish("topic", message); // Publish message to appropriate topic  
}
```

Add a click event to the button of the “LED A” Card, which should call the toggle function when the button is pressed. Pass the name “LED A” to the function. The button component should be update as seen in the code block below.

```
<v-card-item>  
  <v-btn class="text-caption" text="Toggle" variant="tonal" color="primary" density="compact" @click="toggle('LED A')"></v-btn>  
</v-card-item>
```

You are required to implement a similar logic for the “LED B” Card.

Update the CSS style in the style section to either remove or comment out all border properties except the border define in `.col3`. This will remove all the coloured borders added to the webpage to distinguish between the grid system elements (container, rows and columns)

Activity 3.5 - Create a webpage for Graph route

[Required Effort Level 8.0]

In this task you are required to create a webpage based on the specifications for the second webpage outlined for the frontend section. This is a four-step process consisting of:

1. Creating a blank View/Webpage template.
2. Creating the UI layout in template section using grid system and add the required CSS styling to the style section.
3. Add desired Vuetify components found at <https://vuetifyjs.com/en/components/all/>.
4. Add business logic to the webpage by adding JavaScript code to the script section and the corresponding store found in `src/store` to update all state variables and objects using the data received from both hardware(via MQTT) and backend(via API calls).

These steps were covered in detail in the previous tasks and the following are guidelines to help you complete the task.

Creating new webpage

Create a new webpage for navigation route `‘/graph’`. Give the template file the name `‘Graph.vue’`. You will be using the Bar chart at <https://www.chartjs.org/docs/latest/samples/bar/border-radius.html> to plot the frequency for each number stored in the update collection of your database.

Creating UI layout

Create a container. Set its fluid property and also set its align property to center.

Add three (3) rows inside the container. Set the justify property of the first and third row to center. In addition, set a max width of 1200px in the style attribute of all rows.

Add two (2) columns inside the first row and set their cols property to 12 and their align property to center.

Add a Divider component to the second row. Set the margin-top and margin-bottom properties in the class attribute to 5. Therefore class= `‘mt-5 mb-5’`.

Activity 3.6 - Add Vuetify Components

[Required Effort Level 6.0]

Add a Button component to the first row, second column with the following configurations:

- text: Refresh Graph
- color: secondary
- variant: outlined
- flat

Add a card components to the third row with four (4) `<v-card-item></v-card-item>` elements inside. Add a margin of 2 on all sides. Additionally, configure the card with the following:

- subtitle : LED A
- width:150,
- align; center,
- flat,
- border

Inside the first CardItem, add the text “Turned on”.

Add a span element inside the second CardItem. Insert `{{led_A}}` inside the span as a placeholder. Give the span the following classes: text-h5, text-primary, font-weight-bold.

In the third Card Item add the text “times”.

Give each CardItem, except for the last, a padding of zero (0).

Add a Button component inside the last CardItem, configured as follows:

- text: Update
- rounded: pill
- color: secondary
- variant: tonal
- flat
- Add a margin of 1 on all sides and set the text as caption.

The code block below represents the complete code for the Card.

```
Graph.vue
<VCard class="ma-2" subtitle="LED A" width="150" flat border align="center">
  <VCardItem class="pa-0" >Turned on </VCardItem>
  <VCardItem class="pa-0"><span class="text-h5 text-primary font-weight-bold">{{ led_A}}</span> </VCardItem>
  <VCardItem class="pa-0"> times</VCardItem>
  <VCardItem>
    <VBtn text="Update" class="ma-1 text-caption" rounded="pill" flat color="secondary" variant="tonal"></VBtn>
  </VCardItem>
</VCard>
```

Create an additional card for “LED B” inside the third row. Create this card, the same as the card for “LED A”

Activity 3.7 - Implement Logic

[Required Effort Level 8.0]

Creating a basic bar graph with the Chartjs library.

Inside the first column of the first row, inside the `<template>` section add a canvas element as seen in the code block below. This element will render the graph.

```
Graph.vue
<canvas id="myChart"></canvas>
```

Import the chart js library in the script section

```
Graph.vue
import Chart from 'chart.js/auto';
```

Create five (5) variables in the script section, to store the chart object, chart data and chart configurations.

```

const led_A      = ref(0); // Store count for LED A
const led_B      = ref(0); // Store count for LED B

let chart        = null; // Chart object
const data       = { labels: ['Red', 'Blue', 'Yellow', 'Green', 'Purple', 'Orange'],
  datasets: [
    {
      label: 'Fully Rounded',
      data: [0, 0, 0, 0, 0, 0],
      borderColor: '#1ECBE1',
      backgroundColor: '#4BD5E7',
      borderWidth: 2,
      borderRadius: 5,
      borderSkipped: false,
    }
  ]
};

const config     = { type: 'bar',
  data: data,
  options: {
    responsive: true,
    plugins: {
      legend: {
        position: 'top',
      },
      title: {
        display: true,
        text: 'Chart.js Bar Chart'
      }
    }
  }
};

```

Update the onMounted lifecycle hook as follows.

```

// FUNCTIONS
onMounted(()=>{
  // THIS FUNCTION IS CALLED AFTER THIS COMPONENT HAS BEEN MOUNTED
  const ctx = document.querySelector('#myChart'); // Select canvas for rendering chart
  chart = new Chart(ctx, config ); // create chart
});

```

Create a function for updating the chart with new data and labels.

```

// Update graph with labels and new data
const updateData = ( chart, label, newData) => {
  chart.data.labels = label;
  chart.data.datasets[0].data = newData;
  chart.update();
}

```

Now update the appStore located at **frontend/src/store/appStore** with the necessary API functions for making API calls to the backend routes/endpoints created in activity 2.

Inside appStore add the function given in the code block below. This function makes a GET request, using the Fetch API, to **'/api/numberfrequency'** route hosted by the backend.

```
appStore.js
// ACTIONS
const getFrequencies = async ()=>{
  // FETCH REQUEST WILL TIMEOUT AFTER 20 SECONDS
  const controller = new AbortController();
  const signal = controller.signal;
  const id = setTimeout(()=>{controller.abort()},60000);
  const URL = `/api/numberfrequency`;

  try {
    const response = await fetch(URL,{ method: 'GET', signal: signal });
    if (response.ok){
      const data = await response.json();
      let keys = Object.keys(data);

      if(keys.includes("status")){

        if(data["status"] == "found" ){
          console.log(data["data"] )
          return data["data"];
        }
        if(data["status"] == "failed" ){
          console.log("Inventory Item Not Found");
        }
      }
    }
    else{
      const data = await response.text();
      console.log(data);
    }
  }
  catch(err){
    console.error('getFrequencies error: ', err.message);
  }
  return []
}
```

Inside appStore add the function given in the code block below. This function makes a POST request, using the Fetch API, to **'/api/oncount'** route hosted by the backend.

```

const getOnCount = async(LED_Name) => {

  const URL          = '/api/oncount'

  // FETCH REQUEST WILL TIMEOUT AFTER 60 SECONDS
  const controller   = new AbortController();
  const signal       = controller.signal;
  const id           = setTimeout(()=>{controller.abort()},60000);

  const form         = new FormData(); // Create form
  form.append("LED_Name",LED_Name); // Add variable to form

  try {
    const response = await fetch(URL,{ method: 'POST',body:form, signal: signal });

    if(response.ok){
      const data    = await response.json();
      let keys      = Object.keys(data);

      // console.log(data);

      if (keys.includes("status")){

        if (data["status"] === "found"){
          console.log(data["data"]);
          return data["data"]
        }
        else if (data["status"] === "failed"){
          console.log("Unable to get LED state");
        }

      }

    }
    else{
      const data    = await response.text();
      console.warn(data);
    }

  }
  catch(err){
    loading.value    = false;
    if( err.message === "The user aborted a request."){
      console.log("REQUEST TIMEDOUT");
    }
    console.error('getOnCount error: ', err.message);
  }
  return 0
}

```

Export or return both functions so that they can be used, where this store is imported

```

return {
  getOnCount,
  getFrequencies
}

```


Import the appStore in the Graph.vue view.

```
// IMPORTS
import { useAppStore } from "@/store/appStore";

// VARIABLES
const AppStore = useAppStore();
```

appStore.js

Now that the API functions have been created and the appStore imported in the Graph.vue webpage, the functions can be called from the webpage and the data they return, used to update the webpage.

Create two functions in the script section of the webpage that will be used to fetch data from the backend and subsequently update the webpage.

Add the functions given in the code block below to the script section of the webpage.

```
// FUNCTIONS

// Fetch new data and update graph
const updateGraph = async () =>{
  let result = await AppStore.getFrequencies();
  let labels = [];
  let data = [];

  if (result.length > 0){
    result.forEach(obj => {
      labels.push(obj["number"])
      data.push(obj["frequency"])
    });

    updateData(chart,labels,data);
  }
}

// Fetch new data and update cards
const updateLEDCount = async(name)=>{
  let result = await AppStore.getOnCount(name);
  // console.log(result);
  if (name == "ledA"){
    led_A.value = result;
  }

  if (name == "ledB"){
    led_B.value = result;
  }
}
```

Graph.vue

Add click events to the appropriate buttons, calling the functions necessary to update the webpage.

```
<VBtn text="Refresh Graph" flat color="secondary" variant="outlined" @click="updateGraph()"></VBtn>
```

Graph.vue

```
<VBtn text="Update" class="ma-1 text-caption" rounded="pill" flat color="secondary" variant="tonal"
@click="updateLEDCount('ledA')"></VBtn>
```

Graph.vue

```
<VBtn text="Update" class="ma-1 text-caption" rounded="pill" flat color="secondary" variant="tonal"
@click="updateLEDCount('ledB')"></VBtn>
```

This would be a great time to commit and push your code to your GitHub account.

End of Experiment 1