**IOT CA2**

**The Smart Coaster (Alpha Release)**



Jamie Lawlor

Caitlin Maguire

Luke Hilliard

Shahzad Shabeer

**Version 4**

**24 November 2024**

**Table Of Contents**

1. [**SYSTEM ARCHITECTURE……...………………………………………………………..**.](#Hardware_Section) **3**
   1. IoT Elements………………………………………...………………..………………………..…...
   2. Web Server…………….……………………………………………….………...……………..…...
   3. Database……………………………………………………………………………………………..
   4. Flow of Data………………………………………………………………………………………….
   5. Security in transit, at rest……………………………………………………………………………
   6. Fritzing sketch……………………………………………………………………………………….
2. [**ALPHA PROTOTYPE.………….…………………….**](#Data_Section)**..........................................................**
   1. [Hardware](#Data_Gathering)…….………………………...…………………………………………………...………..
   2. [Web](#Data_Storage) Server...……………………………………………………………………………….………..
   3. Hosted on AWS……..……………………………………………………………………………..
   4. PubNub Communication…………………………………………………………………………….
   5. Database……………………………………………………………………………………………..
   6. Security……………………………………………………………………….……………………..
3. [**DEPLOYED TO AWS…….…………………………………………………….**](#Security_Section)**...................**

**3.1** [Custom](#Physical_Security) Domain..…………………………………………………………………………………….

**3.2** AWS……….………………………………………………………………………………................

1. [**PUBNUB……………………...………………………….………………………………….**](#UI_Section)
   1. Communication…………………………………………………………………………...………
2. [**SECURITY……………………...……………………….………………………………….**](#UI_Section)
   1. IoT device………..………………………………………………………………………...………
   2. Access to communication channels………………………………………………………………..
   3. Database………………………………………………………………………………………….
   4. Web Server…………………………………………………………………………………………
   5. Data In Transit……………………………………………………………………………………..
3. **System Architecture**

**A diagram of a network

Description automatically generated**

Figure System Architecture diagram created on Draw.io (Jamie)

**1.1 IoT Elements -**

Our system is comprised of 3 IoT elements, a light sensor, temperature sensor and LEDs. The light sensor is used to detect when a cup is placed on the coaster. This will then trigger monitoring of a drink. The temperature sensor monitors drink temperature from the bottom of a cup as the sensor itself is integrated into the surface of the coaster. As the temperature is being read in, this data is being fed to the LEDs where they will trigger based on the temperature falling within certain thresholds. Red to hot, Blue to cold and Green when it’s just right.

**1.2 Web Server – Jamie**

**Ubuntu server**

Using AWS I created a t3 micro ubuntu instance to act as our web server for the project.

This cloud server acted as our central connection for the various parts that controls this project. The server retrieves messages and data from the IoT device using pubnub as a means of connection. The server contacts the database for queries that need to be run on the database and returns the output to the front-end. It also acts as the front-end’s security, establishing a https connection with an SSL certificate thanks to certbot. It acts as a controller for the various web pages the user can access.

**1.3 Database - Caitlin**

**MongoDB Atlas**

Using MongoDB Atlas I created a drinks collection within the Sipify database. The drinks data holds the drink name and temperature ranges. The drinks data is displayed on the front end using a post request from the server, the user will select a drink, and the name of the selected drink will come back to the database. Within the drink status collection, a new record will be created with the selected drink name.

Once the current temperature comes in from the server this will also be added to the drink status record. The current temperature will be read and compared against the minimum and maximum temperatures of the selected drink and a notification will be issued based on this. The updated drink status record will then be sent to the front end to be displayed on the barista mode page. The user will then be able to view the name of the selected drink, the current temperature and the corresponding notification.

**1.4 Flow of Data – Jamie**

**Traversal through the system**

The data begins by being gathered by the light-dependent resistor and the temperature sensor. This data is stored on our Raspberry Pi, which publishes a message to our pubnub channel to begin sending the data. On our web server, we have subscribed to this pubnub channel to listen for messages being sent. Once the message is received, the server will determine if it is valid and begin the secure transfer of data from pubnub to the server. The server will then take this data and send it to the database to be stored. The project data is stored inside the server for deployment which creates our front-end and applies the SSL certificate created using certbot for a secure user connection. The database sends data to the front-end so that the user may select a drink. The user’s input is sent to the server through a POST request. The server sends this data to the database and returns this data alongside the data received from the raspberry Pi, back to the front-end for viewing.

At the same time as this, the server sends a message to pubnub to be received from the raspberry pi. This data is then used to activate the appropriate LED to display the drink status to the user from the hardware itself.

**1.5 Security in Transit, at rest –**

**1.6 Fritzing sketch –Luke**



**2. Alpha Prototype**

**2.1 Hardware – Luke**

Our smart coaster uses 3 different circuits to provide feedback to the server and to the user. A light sensor circuit is built to detect objects placed on the surface of the coaster. A temperature sensor is embedded into the surface to gather temperature data from cups. Finally, a set of 3 LEDs of colours R, G, B stick out from the surface to provide visual feedback on temperature.

For our first prototype I created a cardboard version of the coaster with access to the middle. In the centre I placed a breadboard inside with a hole out to back so I could feed jumper wires into the coaster. Here is what that looked like while setting up the LED warning circuit.

A cardboard box with wires and wires

Description automatically generated

Luckily this breadboard fits perfecting within the prototype giving me a great working space to add more hardware. In the above picture, each colour wire corresponds to an LED, these wires connect to board pins 40, 38, 36 with each having their current passed through a 100Ω resistor. The breadboard is grounded on one side by the white wire and each LED has there cathode connected to this bank. This approach results in less wires and fewer connections within the limited space

Before the coaster can react with LEDs it must know if a cup is present or not to begin reading temperature data. To achieve this I have implemented a light sensor circuit use a Light Dependant Resistor and a Capacitor. This works but setting up the LDR and capacitor in junction with each other. The LDR will adjust its resistance based on light intensity resulting in the capacitor to charge at a slower or faster rate. If the resistance is high the light levels are low which means an object (cup) is placed on the surface.

Below is the completed prototype circuit with the LDR components marked -

A box with wires and wires

Description automatically generated

One of the most important sensors in our project is the temperature sensor, without it we would have no value to trigger reactions on the hardware or UI side. To gather this data, I have gone with the DS18B20 Digital Thermometer. This piece of hardware is very small which is what is necessary as it will sit on the surface of the coaster and needs to take up as little space so as not to disturb a placed cup.  One issue I came across while developing the temperature circuit was that the temperature from the bottom of a cup will be different to middle where the actual temperature is. To investigate this further, I developed a test which uses a version of the DS18B20 that is submergible in parallel with the surface mounted one to gather data in a set period and check the difference.

The recording and outcome of this test can be found here - [Difference in Temperature Tests](https://www.dropbox.com/scl/fo/luxr37dbvs66syyenv16e/APyKMzuWbKSve_PMf9lBuNE?rlkey=ylnephiqb2c2tfeqgwzts253p&dl=0)

A box with wires and wires

Description automatically generated

Above is the temperature circuit integrated into the cardboard prototype. The sensor uses the 4.7k Ω resistor as a pull-up. A pull-up resistor ensures that a signal line connected to a sensor does not float when the sensors output is inactive. This can cause inaccurate readings which is not ideal for our project.

While developing this prototype I came across some challenges which I will improve upon for the next release. For starters the height of the coaster must be increased to accommodate the jumper wires. If you look closely at the photo I have bent the connectors on the breadboard so that the lid will close resulting is some of them snapping.

Another issue I ran into was developing this with a Raspberry Pi 5. This model is much newer and has a more advanced System on Chip (SoC) compared to older models such as the Raspberry Pi 2 Zero which we intend to use for the final release. With newer hardware the Pi 5 interacts with the GPIO pins in a different way resulting in the RPi library not working, which is what was used to write code for each IoT element. There is an alternative high-level library called gpiozero which I could use but I would prefer to stick with RPi.

**2.2 Web Server – Jamie**

**Server with Flask App**

In the app.py I controlled the app.routes() to ensure the receiving and sending of data. To receive the user’s input on the drink selections page I used a POST request which calls a request from the form on the webpage to get the selected drink and temperature. From this I call the MongoDB method for adding a drink to the database from the MongoDB.py file and redirect the user to the barista page. This is all run in the background hidden rom the user. (WORK IN PROGRESS)

**2.3 Hosted on AWS - Jamie**

**EC2 Instance**

Using Amazon Web Services I created an ec2 ubuntu cloud server instance to host the project. I begun with deploying the example Buzzer project we created in class to see if the deployment would be successful. I utilised this project to test connections to pubnub before our project was created. Once we had a functioning alpha I substituted the Buzzer project with the Alpha project and ensured connections between the database to the server and from the server to the hardware were still successful and operating.

**2.4 Pubnub communication – Jamie**

**Pubnub with Hardware**

I utilised Pubnub to create communication, primarily between the hardware and server. Using Pubnub's publish/subscribe messaging API I could control the data being retrieved from the pi to be used for various purposes. For this project, Pubnub served a great purpose for controlling the Light Dependent Resistor to detect whether or not a cup has been placed onto the coaster, the LED’s to trigger based on the drink’s current temperature and temperature sensor to collect the sensor data.

**2.5 Database – Caitlin**

**Gathering Data**

For the database aspect of the project, I started off with creating a local database using MongoDB Compass. I created a database called Sipify, with three collections inside. I gathered coffee temperature data and created a dataset based off my research for the project.

To track coffee temperatures and issue notifications based on the current temperature, we firstly needed some data to compare the temperature against. I carried out some research to find out what exactly are the ideal coffee drinking temperatures, and most importantly discovering the minimum and maximum temperatures. The maximum temperature being the hottest temperature the coffee can be served at, and the minimum being the coldest. I also got some data about tea, as we would like the Smart Coaster to be used by everyone not only coffee drinkers. I created a csv file based on the drinks data and imported this into MongoDB Compass into my Sipify database. My aim was to find the temperature ranges based on the different coffee types and get the average drinking temperatures. I found the following websites helpful at figuring out the temperature ranges for each drink type:

<https://thedrinksproject.com/how-hot-is-coffee-served/>

<https://weareliferuiner.com/perfect-temperature-hot-chocolate-the-ultimate-guide/>

<https://letsdrinktea.com/what-is-the-right-temperature-for-drinking-tea/>

**Drinks Data:**

A screenshot of a computer

Description automatically generated

**Notifications Data:**

A screenshot of a computer

Description automatically generated

**Drink Status Data Example:**

**A screenshot of a spreadsheet

Description automatically generated**

**Connecting to the Database:**

I created the drinks, notifications and drink status collection in the local database using MongoDB Compass. For the drink status I first used sample data to test the displaying of data, the idea of the drink status collection is to hold all data related to the drink the user has set on the coaster. Starting with the database I used the local database to get data displaying and functions working, after this I began working on the MongoDB Atlas.

**MongoDB Atlas:**

I created a cluster on MongoDB Atlas called sipify-mongodb, within the cluster I have three collections similarly to MongoDB Compass, I added the data for each collection and inserted each of the drinks and the notifications into the collection. For the drink status this will be empty until a drink has been added to the collection.

**Flask App:**

I defined each of the routes in the app.py file within the flask app, for this release we have a loading screen, drink selection page, barista mode page and a view temperatures page. Starting with the drink selection page, I created a view all drinks function to display the list of drinks for the user on the front end.

By calling this function in the drinks selection route it will display a list of all the drink names as radio buttons, allowing the user to select a drink. Before the project was fully connected, I first used an input box for the user to input the current temperature to show how the database will deal with the temperature. Once the user inputs the temperature and selects a drink, this data is sent to the database using a POST call on Jamie's server side, I then included a function to add the inputted data into the drink status collection.

On the barista mode page, the selected drink, current temperature and the notification matching the temperature are displayed.  To get the notification I have compared the inputted temperature with the minimum and maximum temperatures. Within the function I am checking whether the selected drink matches any of the drinks that is in the drinks collection, the current temperature is then compared against the drinks minimum and maximum temperature ranges. If the drink is in the colder range, it will output the notification message associated with the cold status, it is the same for the hotter range the hot status notification will be outputted. The notification is passed in to the add drink status to be displayed to the user.

* 1. **Security – Jamie/Shahzad**

**Overall Security - Jamie**

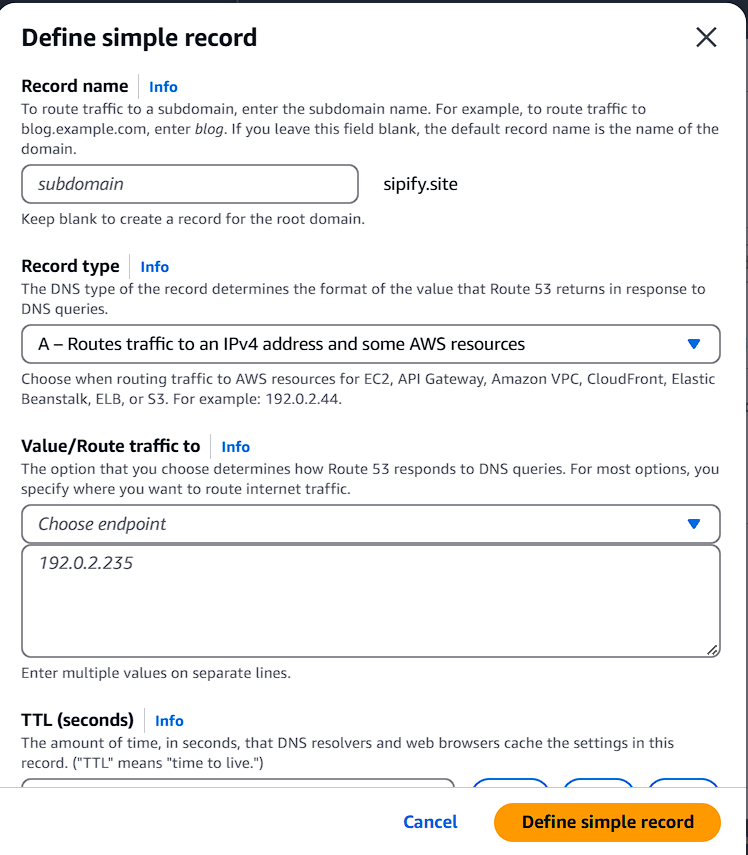
I ensured security in a number of ways throughout the project. I created the security for connecting to the AWS instance, ensuring secure https access to the project online and I am in charge of the pubnub security as well. I will go into depth on each of these parts further down in the document.

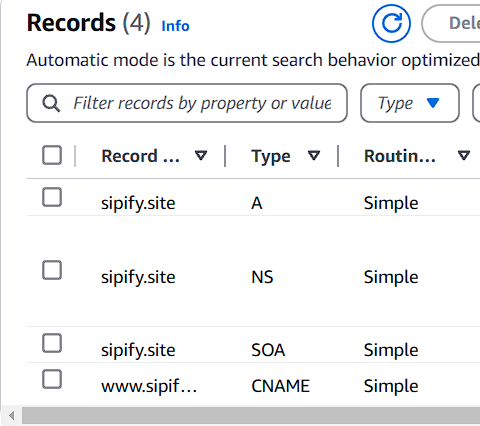
**3. Deployed to AWS**

* 1. **Custom Domain – Jamie**

**Godaddy.com**

Using godaddy.com I was able to purchase a custom domain for the project. I had been given a number of different options for the domain name and after a long discussion with the team, we decided on sipify.site. After gaining access to the domain I created a hosted zone for the domain on Route53. Using this I was given access to AWS name servers to route traffic to my domain. I created a record for my domain and assigned the IPv4 address from my instance to route the traffic to connect my project with the domain.





I then changed the godaddy nameservers for my domain to be the custom nameservers that route53 provided to successfully route the domain to the instance. I then created a record to allow www. to be used to access the website as well. This record allowed the website to be accessed using sipify.site or [www.sipify.site](http://www.sipify.site).

A screen shot of a computer

Description automatically generated

Figure Accessing website without www

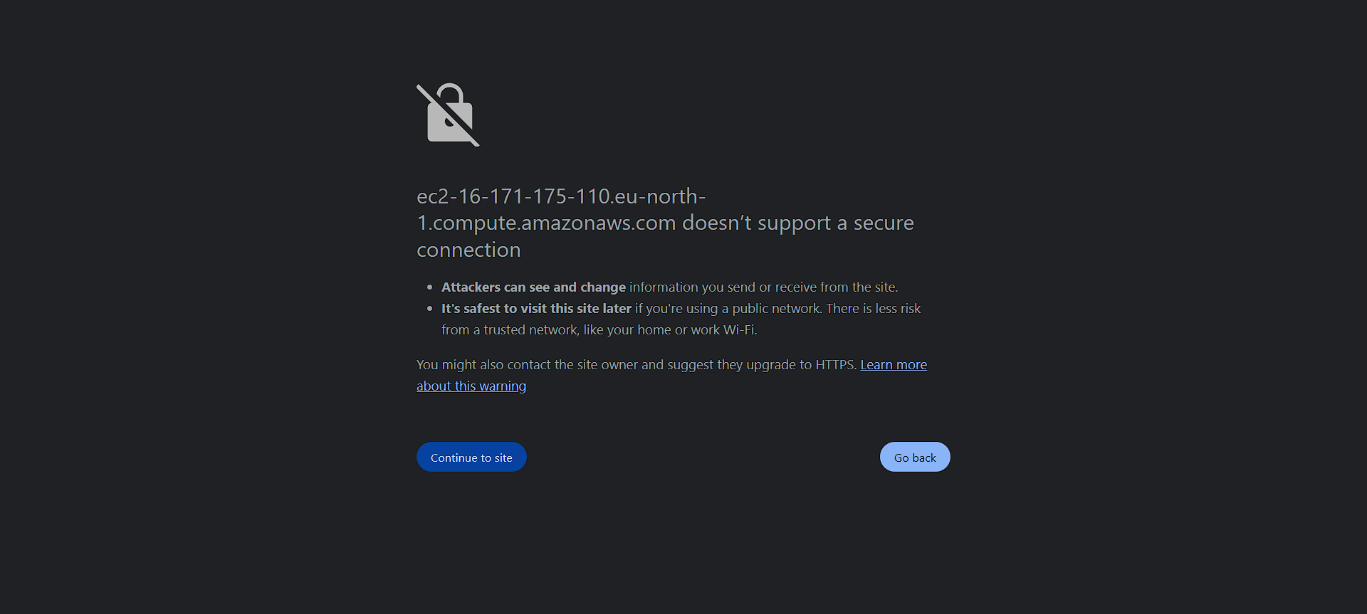
A screen shot of a computer

Description automatically generated

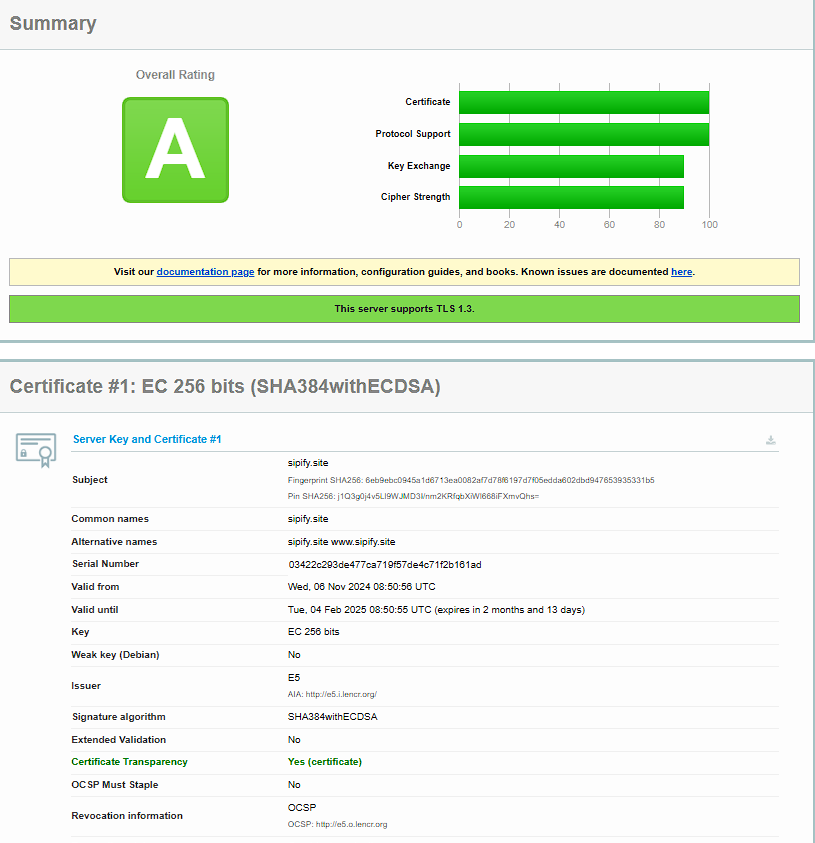
Figure Accessing website using www

**Securing the domain**

When first creating the AWS server and deploying the project, the website only had unsecure http access. In reality most users avoid accessing http websites as they are unsecure and can provide risk to being attacked. When accessing the website originally, Google Chrome provided this message before continuing to the page.



To secure the website and provide a https connection, I used letsencrypt.org to generate an SSL cert for the domain. In the AWS instance I added the inbound rule to allow https requests to the server and redirect all http traffic to https. After using certbot to generate the certificate I ran a test on ssllabs.com/ssltest to ensure that the certificate was successfully generated and that my domain was now secure.

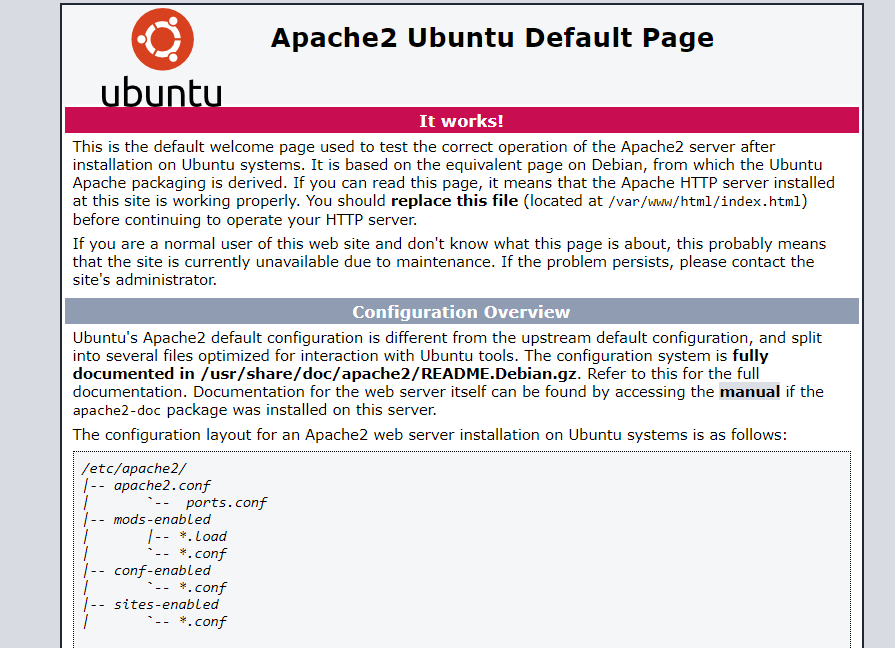


This provides an extra layer of security both to the website and the user as the SSL certificate provides a secure connection just like all the other websites you would use on a daily basis.

* 1. **AWS – Jamie**

**Project Deployment**

As apart of my backend role I was responsible for deploying the project onto AWS. After creating my amazon web services account I created a ec2 t3.micro ubuntu instance for the server. Using PuTTY I was able to access the server using SSH to begin development. Once apache2 was installed I accessed the .conf file and included the link to the instance so that it could display the default Ubuntu page. On the instance I added an inbound rule to allow for http traffic so that the default Ubuntu page would display when using the instance’s IPv4 address.



To deploy the project onto Apache I need to use a web server gateway interface (wsgi) to allow Flask to talk to Apache on the Ubuntu server. After installing the library and installing flask onto the instance I created the directory to store our IoT project.

A black background with white text

Description automatically generated

**WinSCP**

I used WinSCP to transfer the project files onto the AWS instance. The instance denied my access so I had to permit login for root in the sshd\_config file and copy the authorisation keys from my ubuntu user over to the root user. After rebooting the instance I was to copy the files over to the AWS instance. The server cannot recognise the app.py so I had to rename it to \_\_init\_\_.py to be recognised and run.

A screenshot of a computer

Description automatically generated

**Configure Virtual Host**

I created a FlaskApp.conf file to prepare for displaying the project when accessing the instances IPv4 address or using the custom domain. I created a virtual host on port 80 for http access and ensured the project would display on the domain, that the wsgi would be run, and that an error log would be created for debugging when there were issues to be solved. Later on I created another conf file called FlaskApp-le-ssl.conf to have a virtual host on port 443 to allow https access once the domain was secured.

After this I created the wsgi file to allow the Apache Server to serve the project. Once this was finished I restarted and reloaded the server. Finally I disabled the default ubuntu page .conf file and enabled my FlaskApp.conf file and restarted the server once more to see that the project was now displayed successfully.

**4. Pubnub**

* 1. **Communication – Jamie**

**Connection to hardware**

As I mentioned previously, Pubnub played a major role in the communication between hardware and server. Using Pubnub I was able to send a message from the light dependent resistor and the LED’s to be received on the server and to execute the appropriate methods attached to those messages. For this release we had some difficulty with the temperature sensor connecting due to our raspberry pi 5 being too up to date for the libraries. For the final release I will use pubnub to send the temperature data to the server and then store it on the database. After creating my pubnub account, I created a Sipify App with a keyset to be used for the project.

A screenshot of a computer

Description automatically generated

Using the publish and subscribe key from the keyset it allowed me to send messages to this keyset from the pi and listen for the messages and requests on the server. Using the debug console I set the channel to Sipify-channel to check for messages being received from the hardware.

A screenshot of a computer

Description automatically generated

I used the SipifyTestApp in the server to test the connection originally by getting a message to display on the front-end depending on the type of message sent from the hardware. For the LDR I had the front-end display whether or not there was a coffee cup placed on the coaster and for the LED’s I displayed which LED should be activated. For final release I will send the data with the LED message to trigger the appropriate LED. As well as this I will use the data from the LDR to automatically bring the users to the drink selection page once a drink has been registered to have been placed on the coaster. The temperature sensor will be our most crucial sensor which will have its data stored on the database and get displayed for the user on the front-end. Using cron I plan to automatically run the gathering of temperature sensor data every minute to ensure that it gathers the current temperature for users to view in real time.

**5. Security**

* 1. **IoT Device –**

### **Prevent direct access to Iot device (Shahzad)**

Disabling unused gpiopins

To enhance the security , I disabled unused GPIO pins in the system. This step minimizes vulnerabilities by preventing unauthorized access or manipulation of the hardware. By limiting access to only essential components, the device's overall security and stability will be significantly improved.

### **Box to place device**

I attempted to find a suitable metal box to house the Pi for improved durability and to prevent direct access. However, I was unable to get a metal enclosure in time. Instead, I sourced a plastic box as an alternative. After discussion with the team, we decided not to proceed with the plastic option

A plastic container on a counter

Description automatically generated

### **Adding Security Tests**

I implemented and executed security tests for the project to ensure the application is robust against potential threats like SQL injection and XSS attacks. These tests involved simulating harmful inputs to confirm that the app handles them safely. As a result, the app is now more secure and reliable

* 1. **Access to communication channels – Jamie**

**Access Manager**

For this release I was unfortunately unable to complete the security for pubnub however I will be able to achieve the security using access manager by the final release. Access manager is used for granting access to pubnub channels as a form of security. A user will need a cipher key if they wish to send or receive a message from a channel. Each time a user logins there will be a request sent to generate a token. This token will be used to determine what permissions to give the user, whether they can have read or write access or both. As the admin you will have full control over the permissions for users. These keys are also a secure way of ensuring that a user cannot give themselves unauthorised permissions such as admin access to the project. For our project these keys will be an excellent way of ensuring that no malicious data is sent to the hardware or to the database. It will also allow us to control the add, edit and remove drink functionalities for the barista’s to ensure that only they have access to changing the database for their companies. A regular user wouldn’t require access to these features for general use.

* 1. **Database – Caitlin**

**MongoDB Atlas**

I enabled the network access to allow any IP Address to connect this will allow my team members to connect, and I also added the IP address of our server. Within the database access, I created an admin user for myself and added in my team members as users with read and write access to the database. A username and password is required to gain connection to the database on MongoDB Atlas.

**Database Access on MongoDB Atlas:**

A screenshot of a computer

Description automatically generated

**Connection to MongoDB Atlas**

I added in the database connection string into the .env file, and called the database URI in the app.py file.

**Issues**

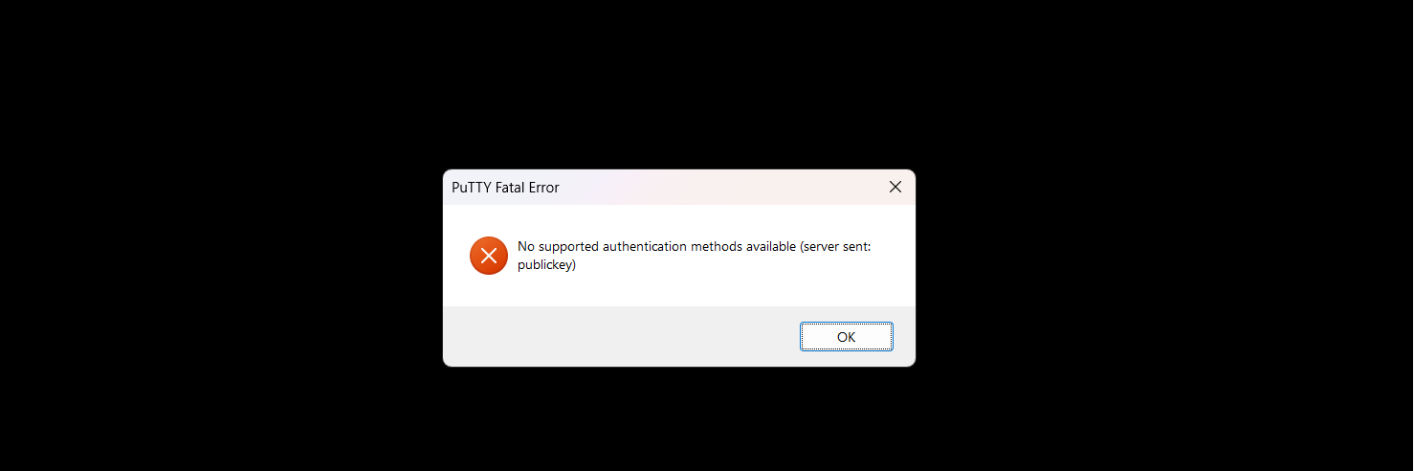
I begun working with the database on MongoDB Compass, I decided to get functions working like displaying all drinks and allowing the user to select a drink and input temperature, before moving on to MongoDB Atlas. Some issues I had within using compass was getting used to the code for Mongoengine, between the MongoDB documentation and the videos on Moodle it was difficult to figure out whether to use pymongo or Mongoengine. It took some time to get used to working with Mongoengine and after a while I realised Mongoengine was far easier to work with and made the most sense to me, so I decided to stick with it for bringing in and displaying the data on the Flask App.

I was having a few problems with the authentication on MongoDB Atlas, first of all I was getting a “bad auth” error. To fix this I created a new admin user with a new password, the next issue was once the error had gone there were no drinks data coming in however there were no errors. After a while, I realised the connection string did not have the database name only the cluster name and this was the issue, even though I had copied in the connection string from MongoDB Atlas itself.

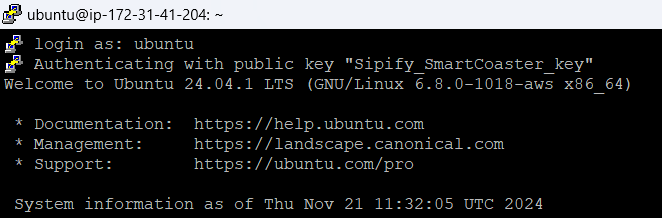
* 1. **Web server – Jamie**

**Public and private keys**

It was essential that security was in place for the AWS instance so that an unauthorised user does not gain access and destroy the server. As I was creating the instance I created a key pair so that it removed the need of a username and password and provides a much tighter security as a user would need the private key to access the server. To test this I attempted to login as ubuntu to access the server without the private key and my access was immediately denied.



To test the correct connection I attached the .ppk file that allowed PuTTY to recognise the private key and was able to successfully SSH into the instance.



This security allows applied to WinSCP which prevents an attacker from transferring unauthorised files into the server. Without access to the private key, the instance remains secure and rejects all other attempted access.

* 1. **Data in transit -**