**IOT CA2**

**The Smart Coaster (Alpha Release)**



Jamie Lawlor

Caitlin Maguire

Luke Hilliard

Shahzad Shabeer

**Version 3**

**23 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 -**

**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.

**2.2 Web Server – Jamie**

**Server with FlaskApp**

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 cloude 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 and 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

**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**

**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

* 1. **AWS – Jamie**

**4. Pubnub**

* 1. **Communication – Jamie**

**5. Security**

* 1. **IoT Device –**
  2. **Access to communication channels – Jamie**
  3. **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

* 1. **Webserver – Jamie**
  2. **Data in transit -**