# Programming Things Report

Jake Michael Courtman-Stringer

## Contents

## Overview

Our project focused on creating a home security system, interconnected over a central hosted server, using a mixture of MQTT and HTTP requests. My role in the project was to create the central API and MQTT functionality, which would be hosted on a server to allow the other team members to connect to it remotely.

## Research

Initially, we aimed to create a system that used 'mosquitto', an MQTT broker, and communicated on a local network. However, due to COVID restrictions, we were living in different households across the country and so needed to opt for something more distributed and available across the internet.

We found that a Node-RED package existed called 'Aedes' which, when added to a Node-RED flow would begin an MQTT broker instance and allow us to communicate through it.

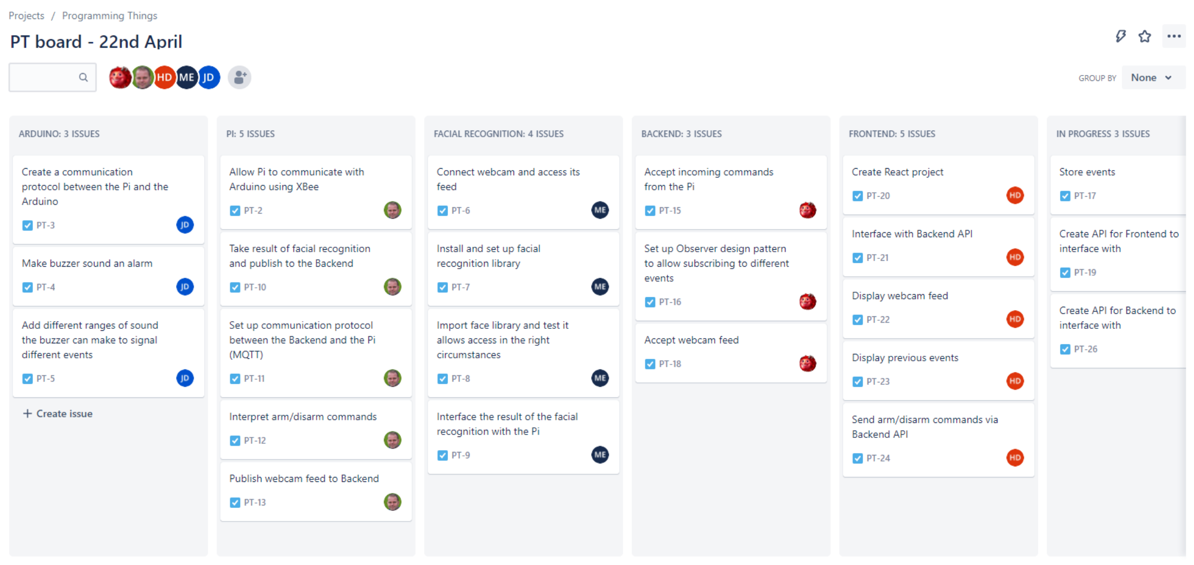
However, Node-RED caused a lot of problems to run on a server. Firstly, it wanted root (super user) access, and a lot of access to the file system. Both of these were difficult to do on a central server, and so in the end we decided to look into a different way to incorporate MQTT functionality on a server.

Node Express can have MQTT functionality enabled using the 'mqtt' package. This allows us to subscribe and listen to certain topics. As a final solution, we kept the MQTT broker as hosted using Aedes on Node-RED, but added the main functionality to a Node Express API that is running on a central server. Whilst web-socket functionality would allow us to communicate to the MQTT server in a more efficient way from the frontend to the backend, this ultimately caused server-side issues and crashing due to an issue with the way Node-RED was being hosted.

# Development

## Kanban Board and Task Assignment

To begin development, we created a Jira (Kanban) board to facilitate Agile development. We split the functionality of the project into different areas and assigned them to each other; I was assigned the backend API and MQTT functionality:



Projects / Programming Things

PT board - 22nd April

HD ME JD

GROUP BY None

ARDUINO: 3 ISSUES

Create a communication protocol between the Pi and the Arduino
PT-3 JD

Make buzzer sound an alarm
PT-4 JD

Add different ranges of sound the buzzer can make to signal different events
PT-5 JD

+ Create issue

PI: 5 ISSUES

Allow Pi to communicate with Arduino using XBee
PT-2

Take result of facial recognition and publish to the Backend
PT-10

Set up communication protocol between the Backend and the Pi (MQTT)
PT-11

Interpret arm/disarm commands
PT-12

Publish webcam feed to Backend
PT-13

FACIAL RECOGNITION: 4 ISSUES

Connect webcam and access its feed
PT-6 ME

Install and set up facial recognition library
PT-7 ME

Import face library and test it allows access in the right circumstances
PT-8 ME

Interface the result of the facial recognition with the Pi
PT-9 ME

BACKEND: 3 ISSUES

Accept incoming commands from the Pi
PT-15

Set up Observer design pattern to allow subscribing to different events
PT-16

Accept webcam feed
PT-18

FRONTEND: 5 ISSUES

Create React project
PT-20 HD

Interface with Backend API
PT-21 HD

Display webcam feed
PT-22 HD

Display previous events
PT-23 HD

Send arm/disarm commands via Backend API
PT-24 HD

IN PROGRESS 3 ISSUES

Store events
PT-17

Create API for Frontend to interface with
PT-19

Create API for Backend to interface with
PT-26

## Node Express Backend

To begin development, I set up a basic Node Express application using the standards outlined in the Applications and Frameworks (AAF) module. These state that:

- API routes must be split into ‘route’ and ‘controller’ files
- Promises should be used as much as possible (.then() and .catch()) instead of the ‘await’ keyword
- Models for the database should be kept in their own folders

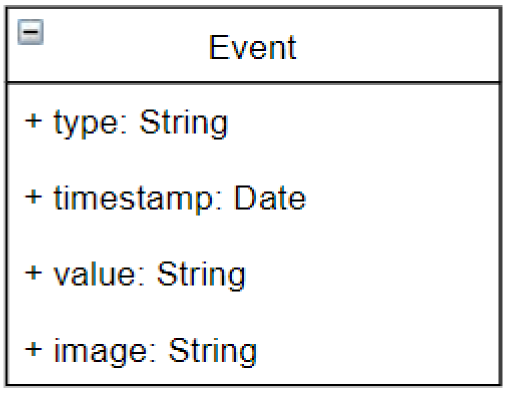
I implemented a **singleton** class to hold our MQTT object within the backend. This means that only one instance of the MQTT connection is initialised, and then kept in memory. All further references/imports of this class will return the same initialised instance:

```
var mqtt = require('mqtt')

class MQTTClient {
    constructor() {
        var options = {
            clientId: (process.env.LOCAL === "true" ? "LOCAL_INSTANCE_" : "S
ERVER_INSTANCE_") + 'file_api',
            username: process.env.BROKER_USERNAME,
            password: process.env.BROKER_PASSWORD,
            clean: true
        }
        this.client = mqtt.connect('mqtt://homesecurity.jakestringer.dev', o
ptions)

        this.client.removeAllListeners();
    }
}

module.exports = new MQTTClient();
```
Note the usage of ‘process.env’, which uses the ‘.env’ file included in the directory to story sensitive information like the MQTT broker’s username and password.

A series of listeners were then set up, namely to listen to MQTT **connection** events, and MQTT **message** events. Upon a client connecting, the backend subscribes them to the ‘PI_ONLINE’ and ‘ARM_SYSTEM’ topics.

## Saving to the database

If a message is received, a new event model is added to the database:



All fields except ‘image’ are mandatory.

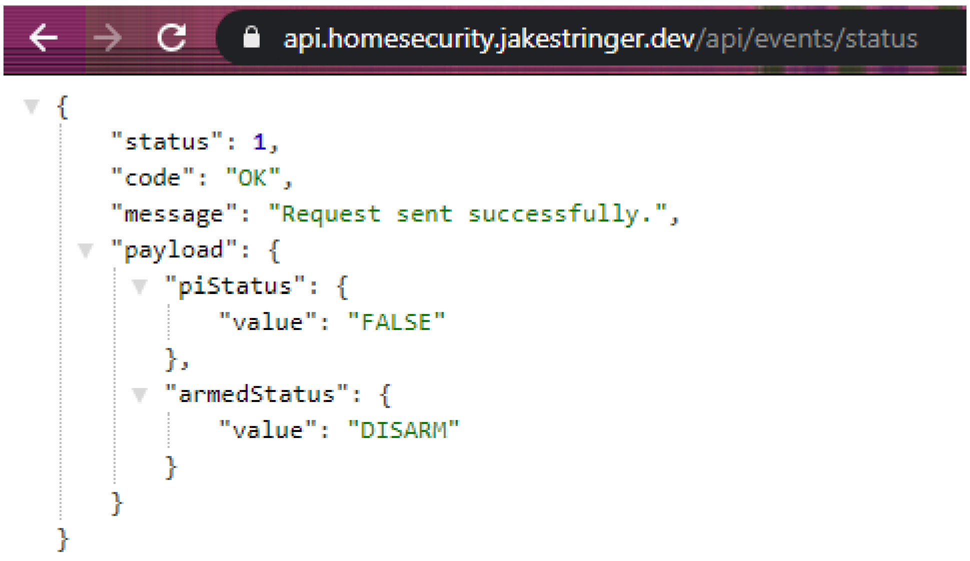
Saving the events to MongoDB not only implements persistence, but it also enables me to utilise the library **mongoose**, which has many powerful tools for making database queries. When a user wants to get the status of the Pi and the armed status of the house, they make a call to the **/status** endpoint:

```
await eventModel.find({ type: "PI_ONLINE" }, { value: 1, _id: 0 })
```
This will:

- Search the database for events that have a ‘type’ attribute equal to “PI_ONLINE” (only searching for the ‘Pi online/offline’ events)
- Apply a ‘projection’ of ‘value: 1’ and ‘_id: 0’. This means that we only retrieve the ‘value’ attribute of any results returned. Since the ‘_id’ field is usually returned by default but not needed in this case, we are explicitly excluding it from the result sets by setting it to ‘0’ in the projection object.
- The line:
  ```
  .sort("-timestamp").limit(1)
  ```

appears afterwards, which sorts the results by '-timestamp' – this signals to mongoose to look for an attribute called 'timestamp', and search it in descending order (using the minus sign). It uses the built-in date value parser to compare the values in the database to order them correctly.
- Finally, it is limited to only one result. This is because we only want the most recent entry to the database under this category.

In short, this reveals to us the latest "PI_ONLINE" event and its value. The same logic is applied to find the latest "ARM_SYSTEM" event too, and both are returned to the user:



```
{
    "status": 1,
    "code": "OK",
    "message": "Request sent successfully.",
    "payload": {
        "piStatus": {
            "value": "FALSE"
        },
        "armedStatus": {
            "value": "DISARM"
        }
    }
}
```

## Uploading images

As part of the system's functionality, the Pi should be able to upload images taken from its webcam and send them to the central API to be saved (and retrieved by the UI). The system is designed so that the facial recognition has already taken place on the Pi (if the system is armed), so once an image reaches the upload endpoint of the API, it already should have the facial recognition results (if the system is armed).

```
// Read the image //
const file = req.files.feed;

// Resize the image to a web-friendly size //
const sharp_image = sharp(file.data);
sharp_image.resize({height: 480, width: 640})
```

The image is taken directly from the request's 'files' attribute (as a result of sending the request as 'multipart/form-data') and then the **sharp** library for Node resizes it to 640x480.

If the request body indicates that a face has been detected using the facial recognition API, then it saves the uploaded image using the current timestamp and saves a new 'DETECTED_FACE' event. It then passes the newly saved image to an image manipulation function using the plugin **Jimp** for Node:

```
Jimp.read("./assets/rect_unrecognised.png", (rect_err, rect) => {
    if(rect_err) throw rect_err;

    let top = eval(coords[0]);
    let right = eval(coords[1]);
    let bottom = eval(coords[2]);
    let left = eval(coords[3]);

    let width = right - left;
    let height = bottom - top;

    rect.resize(width, height);
    img
     .blit(rect, left, top)
     .print(font, left + width / 2 - textWidth / 2, top - 40, name)
     .write(path);
})
```

Along with the image, this function receives the results of the facial recognition API, which contains:

- The name of the face it has recognised, and ‘Unknown’ if it does not recognise the face
- Co-ordinates of where on the image the face has been recognised, passed as the top-right and bottom-left positions of a rectangle.

These co-ordinates are used to work out the width and height of the rectangle needed, and then a transparent rectangle PNG is loaded, and applied (using a function called ‘blit’) to the image at the resulting co-ordinates.

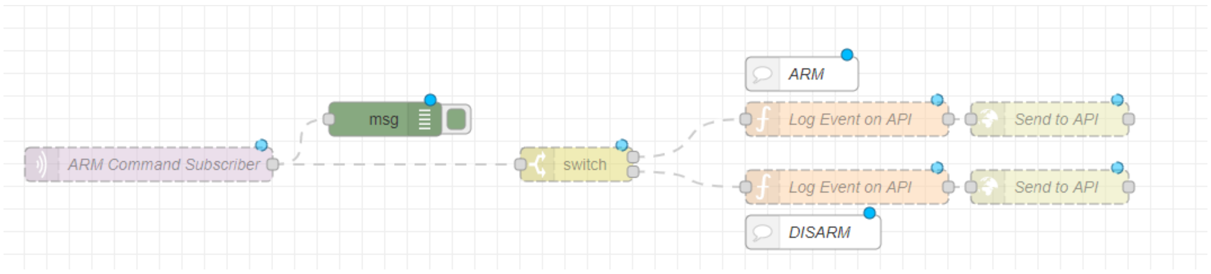
As well as this, text is printed onto the image reading the recognised name, and ‘Unknown’ if the facial recognition did not recognise the person.

If a face was not detected, but the system is armed, it instead blits a small label reading ‘scanning’ onto the image:

```
Jimp.read("./assets/scanning.png", (scan_err, scan) => {
    if(scan_err) throw scan_err;
    scan.scale(0.5);
    img.blit(scan, 0, 0).write(path);
})
```

## Successes and Failures

My initial plans aimed to include Node-RED in a much larger capacity, with the intent being that it would form the entirety of the backend system and completely handle all the API functionality. My initial Node-RED flows did have HTTP endpoint functionality, but these simply ‘forwarded’ requests to the Express API depending on the value of the request:



This was ultimately pointless and we decided to migrate the functionality entirely to the API.

As well as this, I planned to add user functionality to further secure the system. My intention was to add user-based authentication to the app so that the UI could be signed in as different users, and perhaps cater for arming/monitoring multiple households at a time.

On the other hand, the project's aim of creating an interconnected smart-home system using MQTT and an IoT philosophy was successfully met. Whilst not as involved with Node-RED as I intended, the system can fully and efficiently interface with both the Raspberry Pi and the frontend React UI, as well as handling image and database persistence and connecting to the Node-RED MQTT broker using an MQTT package.

Working on this project not only taught me about using the MQTT protocol to connect systems in an IoT network, but also about programmatic image manipulation and even about the importance of correctly handling exit codes within an application.

This is due to an issue we experienced where terminating our backend API code did not terminate the MQTT listeners that were instantiated during runtime. This meant we would have multiple listeners for the same events, leading to the same events being added to the database many times. To fix this, I had to add listeners for exit codes to correctly terminate all the running MQTT processes before letting the Node app close:

```
function exitHandler(code) {
    console.log(code);
    console.log("I am disconnecting");
    client.removeAllListeners();
    client.end();
}

process.on("SIGINT", exitHandler.bind(null, "SIGINT"))
process.on("exit", exitHandler.bind(null, "exit"))
process.on("beforeExit", exitHandler.bind(null, "beforeExit"))
process.on("SIGUSR1", exitHandler.bind(null, "SIGUSR1"));
process.on("SIGUSR2", exitHandler.bind(null, "SIGUSR2"));
process.on("uncaughtException", exitHandler.bind(null, "SIGINT"));
```