PLANTWATCH

An IoT Plant health monitoring system

Abstract

Plantwatch is an IoT plant health monitoring and care system that utilises electronic sensors and a Raspberry Pi to collect data on soil moisture, temperature, humidity, and light intensity. A worker service communicates with the Raspberry Pi via RabbitMQ, logs sensor data to a MongoDB database, and sends control data back to the Raspberry Pi, which can also operate actuators for environmental control. A hybrid web and mobile application allows the end user to add devices to their account, view sensor data, and set desired parameters. Flutter provides the frontend and interfaces with MongoDB via a REST API implemented in Dart.

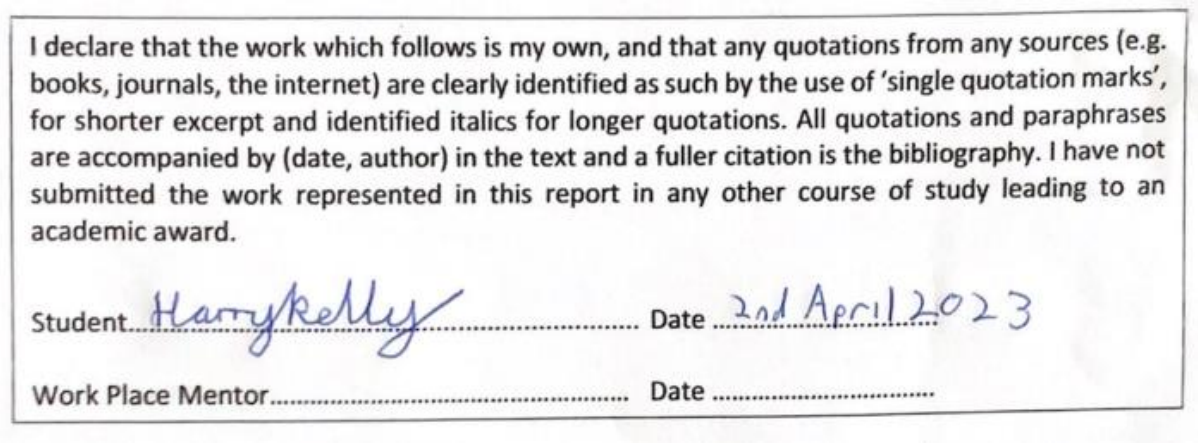
Harry Kelly (20095354)

April 4th 2023

CONTENTS

TABLE OF FIGURES

DECLARATION OF AUTHENTICITY



I declare that the work which follows is my own, and that any quotations from any sources (e.g. books, journals, the internet) are clearly identified as such by the use of 'single quotation marks', for shorter excerpt and identified italics for longer quotations. All quotations and paraphrases are accompanied by (date, author) in the text and a fuller citation is the bibliography. I have not submitted the work represented in this report in any other course of study leading to an academic award.

Student Harry Kelly Date 2nd April 2023

Work Place Mentor ..... Date .....

ACKNOWLEDGEMENTS

I would like to gratefully acknowledge the support and guidance of my project supervisor, Joe Daly. Additionally, I would like to thank all the teaching staff on the Higher Diploma in Science in Computer Science, without whom I would not have learned the skills needed to complete this project.

GLOSSARY OF TERMS

IoT: Internet of Things - physical objects (or groups of objects) with sensors, processing ability, software and other technologies that connect and exchange data with other devices and systems over the Internet.

Sensor: A device that detects or measures physical inputs such as temperature, humidity, light, pressure, or motion, and converts these into electrical signals.

Actuator: A device that converts electrical signals into mechanical actions to control or manipulate physical systems.

GPIO: General Purpose Input/Output - a type of digital pin found on microcontrollers and single board computers, such as the Raspberry Pi, that can work as an input or output, allowing for interface with external devices.

AMQP: Advanced Message Queuing Protocol - a protocol for sending and receiving messages between applications or devices.

Serialization: The process of translating data into a format that can be stored or transmitted.

JSON: JavaScript Object Notation - a lightweight data format used for serializing data, widely used in web applications.

UTF-8: The most popular character encoding standard employed on the internet.

REST API: Representational State Transfer Application Programming Interface - a web-based communication paradigm that allows for the transfer of data between applications. It uses HTTP requests to access and manipulate data stored on a server, using standard HTTP methods such as PUT, GET, and POST.

JWT: JSON web token – a JSON object containing a small amount of data and a digital cryptographic signature. Often used for authorization in the context of web applications and REST APIs.

LED: Light Emitting Diode - a semiconductor device that emits light when an electric current passes through it.

Worker service: A type of background process that runs continuously to perform a specific task, such as data processing or monitoring.

NoSQL: A category of database management systems that use non-relational data models, often for distributed or large-scale systems.

Entity-relationship diagram: A graphical representation of the relationships between different entities in a database. It shows the attributes, constraints, and relationships between these entities.

Logical data model: A high-level representation of the data objects, relationships, and constraints in a database. It defines the rules and requirements of a database, without specifying the implementation details.

Physical data implementation: The actual implementation of a database design.

1 PREFACE

1.1 PROJECT MATERIALS

The project git repository can be found at https://github.com/htkelly/plantwatch. A landing page hosted in the same repository can also be found at https://htkelly.github.io/plantwatch.

1.2 BACKGROUND

As part of a previous course module, I developed an IoT security system, Watchful Pi (Kelly, 2022). Through working on this previous project, I developed an interest in the IoT field, and although this project is a different application, it represents a continuation of that learning process.

Throughout several previous modules, I have also developed my skills in full-stack web and mobile application development, and I was interested in building a project that integrates those skills with my interest in IoT. I also wanted to build a project that would give me hands-on experience with specific tools and technologies I had not used before, such as Flutter, RabbitMQ, and Protocol Buffers.

As the scope of the project was to develop a complete solution, rather than focus solely on the physical computing aspect, I utilized some components and libraries from a hobby electronics vendor, Seeed Studio, such as the Grove Smart Plant Care Kit (Seeed Studio, 2022) and Grove Base Hat for Raspberry Pi Kit (Seeed Studio, 2023). Seeed Studio provides a wide range of components, together with a base hat for interfacing with the Raspberry Pi, and Python libraries for most of their components (Seeed Studio, 2023). Utilising these resources allowed me to quickly prototype the physical computing aspect of the project and focus on the broader system and overall technology stack, without getting too enmeshed in the low-level electronic specifications of the components.

1.3 OBJECTIVES

The original objective for the project was to develop an internet-connected plant health monitoring system that sends sensor data over the internet, where it is stored in a database and ultimately displayed to end-users in a front-end application. The goal was for the frontend application to allow users to specify ideal conditions for their plant and be notified when those conditions are not met. The user would also be able to remotely actuate a water pump for irrigation or specify conditions when the water pump will be automatically actuated.

The project objectives have been fulfilled by the finished work: the developed application uses a Raspberry Pi to read sensor data and control actuators, two-way AMQP messaging between the Raspberry Pi and a backend service that logs sensor data to a database and sends control data to the Raspberry Pi, and a REST API and Flutter frontend application that expose the sensor data to end-users and allow end-users to set desired parameters for soil moisture, temperature, and humidity. A relay-controlled water pump is triggered when soil moisture drops below the set parameter. Other actuators are represented by LEDs in the current prototype.

1.4 MOTIVATION

My primary personal goal in undertaking this project was to expand my knowledge of the IoT field by applying my previous learning in this area to a new project. My previous project, Watchful Pi, gave me my first exposure to essential IoT topics such as sensors and lightweight messaging. I wanted to expand on this by building a project that used actuators as well as sensors, a different messaging technology (RabbitMQ), and a different frontend technology (Flutter).

Another motivation was my interest in technological solutions to the climate and biodiversity crises. Contemporary environmental challenges are a matter of personal concern to me, and I wanted to develop skills that might be useful in that context. IoT technology has significant environmentally valuable applications, for example in the conservation and agritech fields. Charles McLellan writes in ZDNET, '*Technologies nearing maturity include IoT platforms, edge analytics, IoT-based streaming analytics, supervised and unsupervised machine learning, containers, low-power wide-area networks, and pub/sub messaging. Many of these technologies are already in use in biodiversity and climate change-related projects'* (McLellan, 2020). Developing a plant health monitoring system was an opportunity to apply my skills to an area of personal importance to me.

I also wanted to integrate aspects of as many course strands as possible into one project, as well as gain exposure to new technologies that weren't directly covered in the course content.

2 SYSTEM OVERVIEW

2.1 TECHNOLOGIES

2.1.1 HARDWARE

- Raspberry Pi 4 model B
- Seeed Studio Grove Base Hat for Raspberry Pi
- Seeed Studio Grove sensors including
    - Temperature and Humidity sensor
    - Moisture sensor
    - Sunlight sensor
- 12v water pump
- Seeed Studio Grove Relay
- LEDs and resistors

2.1.2 SOFTWARE

- Python 3.x and Seeed Studio grove.py libraries for implementation of the IoT component
- Additional Python libraries for working with RabbitMQ, MongoDB, and Protocol Buffers
- Dart and Flutter for implementation of the frontend web application
- Server-side Dart for implementation of a REST API
- Dart libraries for working with MongoDB and Google charts
- RabbitMQ for lightweight messaging
- MongoDB for data storage
- Firebase Authentication and associated Dart libraries for user account functionality
- Google Cloud Platform for deployment

2.2 FUNCTIONAL SPECIFICATION

2.2.1 USER ACCOUNTS

- User accounts with standard login and signup functionality
- User accounts may be associated with multiple devices

2.2.2 SENSOR DATA

- Data from all sensors is collated and logged as a single timestamped database entry – a 'reading'
- Every sensor reading is associated with the device it came from

2.2.3 INTERFACE

- User dashboard shows all devices associated with the logged-in user's account, with the latest reading for each device
- The user can click on any device to open the device view for that device
- The device view allows the user to see time-series charts of that device's historical temperature, humidity, moisture, and UV index data
- The device view also allows the user to set desired minimums and maximums for temperature, humidity, and moisture

2.2.4 CONTROL

- The same worker service that receives sensor data from the device also checks the database for parameters set by the user and sends these to the device over RabbitMQ
- On-device logic compares current reading values to parameters set by user and triggers the water pump and other actuators accordingly

2.3 MODELLING

2.3.1 SYSTEM DIAGRAM

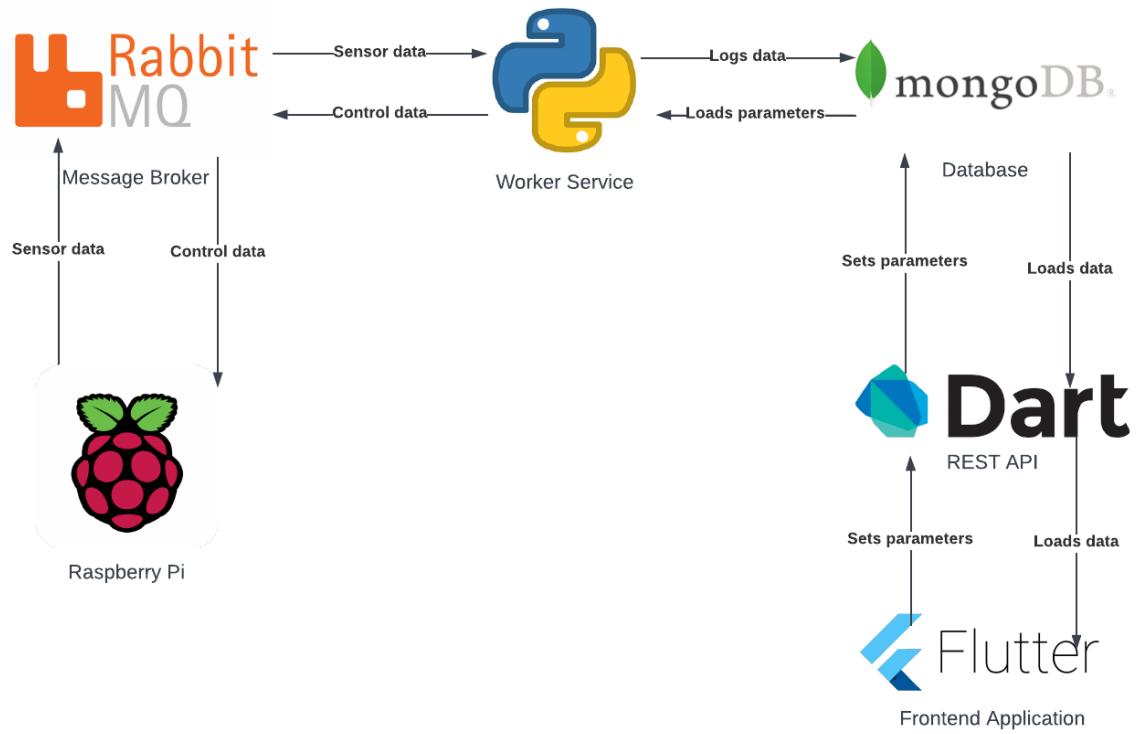


Figure 1: System diagram

Figure 1 is a high-level overview of the system, illustrating the bidirectional flow of data between the frontend application and the Plantwatch sensor device. It also highlights the core technologies used to build the project.

2.3.2 USER INTERFACE

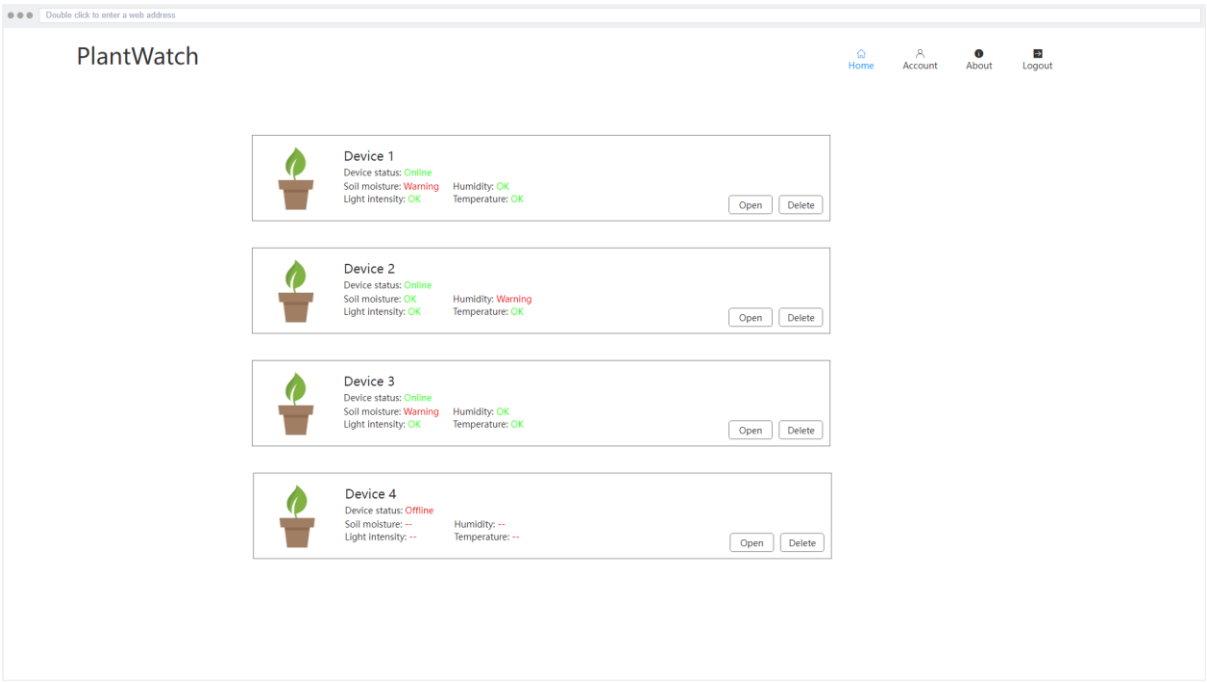


Figure 2: Dashboard UI wireframe

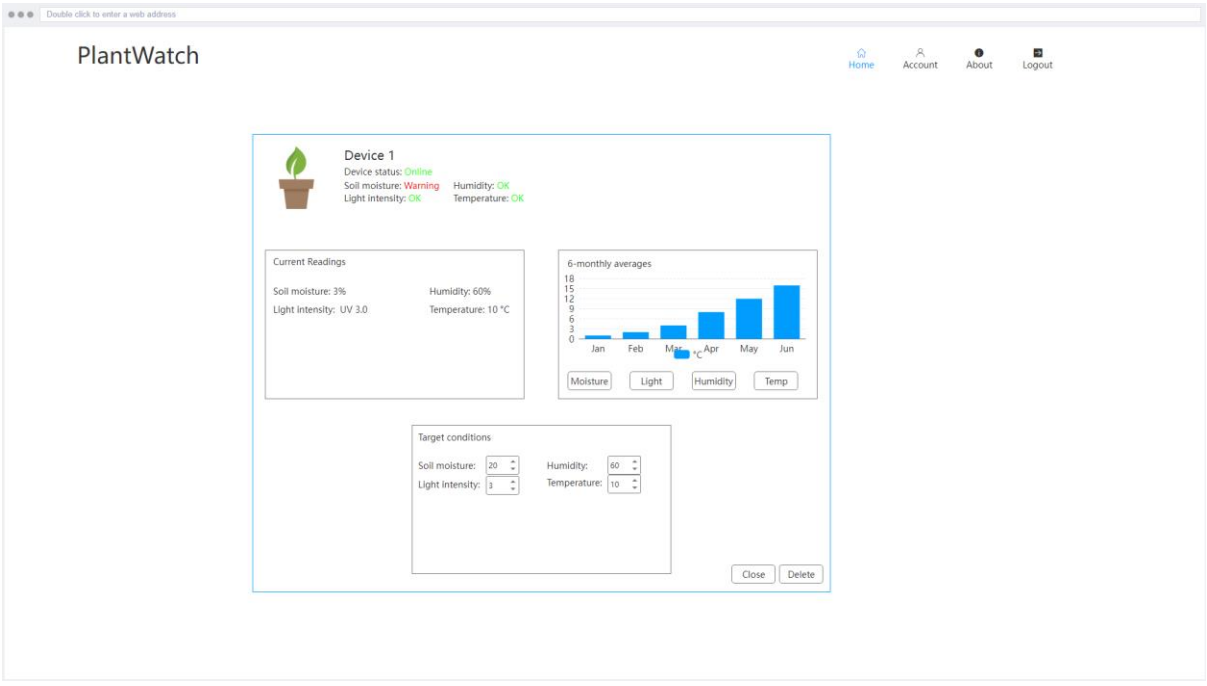


Figure 3: Device view UI wireframe

Figures 2 and 3 are UI wireframes showing how the UI was originally conceived. The current implementation of the UI differs from these, as discussed below, but they could still serve as a guideline for further development of the UI if development on the project were to continue.

2.3.3 DATA MODEL

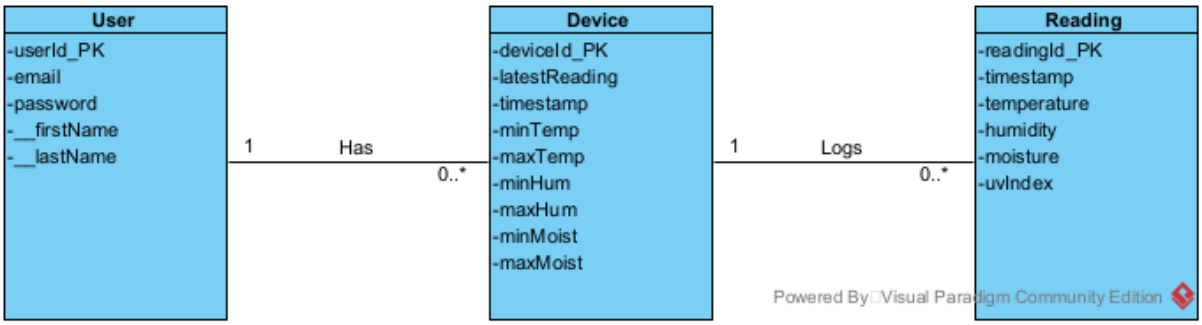


Figure 4: Entity–relationship diagram of data model

Figure 4 shows the data model described in terms of an entity–relationship model. Although I used MongoDB, which is a document-based, NoSQL database, and the entity–relationship modelling technique is typically associated with relational databases, this was still a useful way to conceptualise the data model.

It is important to note that the User entity does not exist in the MongoDB database, as this data is held in Firebase, which I used for authentication. As such, the attributes listed in the diagram represent only those attributes of the User entity which are relevant to the implementation, not all the attributes associated with the user in Firebase.

In translating the entity–relationship model to a physical implementation in MongoDB, I chose to record the relations between devices, readings, and users as references rather than by embedding documents in other documents. Choosing whether to record relations as references by id or to store related documents as embedded subdocuments is a key design decision when working with a document-based database. Rick Copeland writes that a "*factor that may weigh in favor of a more normalized model using document references is when you have one-to-many relationships with very high or unpredictable arity. For instance, a popular blog with a large amount of reader engagement may have hundreds or even thousands of comments for a given post. In this case, embedding carries significant penalties with it*", and cites MongoDB's hard document size limit of 16MB as one potential reason to choose referencing over embedding (Copeland, 2013). This is a key concern in this case, as the high frequency of readings being taken by the device means that this will be a high arity relation and thus embedding the reading documents in the device document is not practicable.

Copeland also mentions flexibility as a potential reason for choosing the referencing approach over the embedding approach (ibid.). This was also a relevant factor in this case, as referencing the user id in the device document allows us to store our user entities in another database (Firebase).

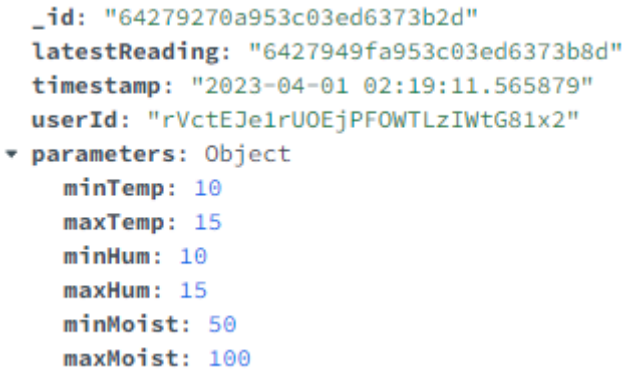


Figure 5: A device document as displayed in MongoDB Compass

```
_id: "64279270a953c03ed6373b2e"
deviceId: "64279270a953c03ed6373b2d"
timestamp: "2023-04-01 02:09:52.857746"
temperature: 24.936676
humidity: 52.20747
moisture: 0
uvIndex: 0.1
```


Figure 6: A reading document as displayed in MongoDB Compass

Figure 5 and Figure 6 show examples of a device and a reading document, respectively, illustrating how the entity–relationship model has been physically implemented in MongoDB. Drawing on what I learned in the Database Design and Implementation module, I understood that physical implementations do not always correspond directly to entity–relationship models or logical data models, hence some differences between the entity–relationship model and the physical implementation are apparent: the primary keys for each entity are named _id, as required by MongoDB, and the device parameters are embedded in a subdocument of the device document, which is not something that would be modelled in the entity–relationship model, because the parameters subdocument does not represent an attribute in its own right, but rather a collection of attributes.

2.3.4 WATER PUMP LOGIC

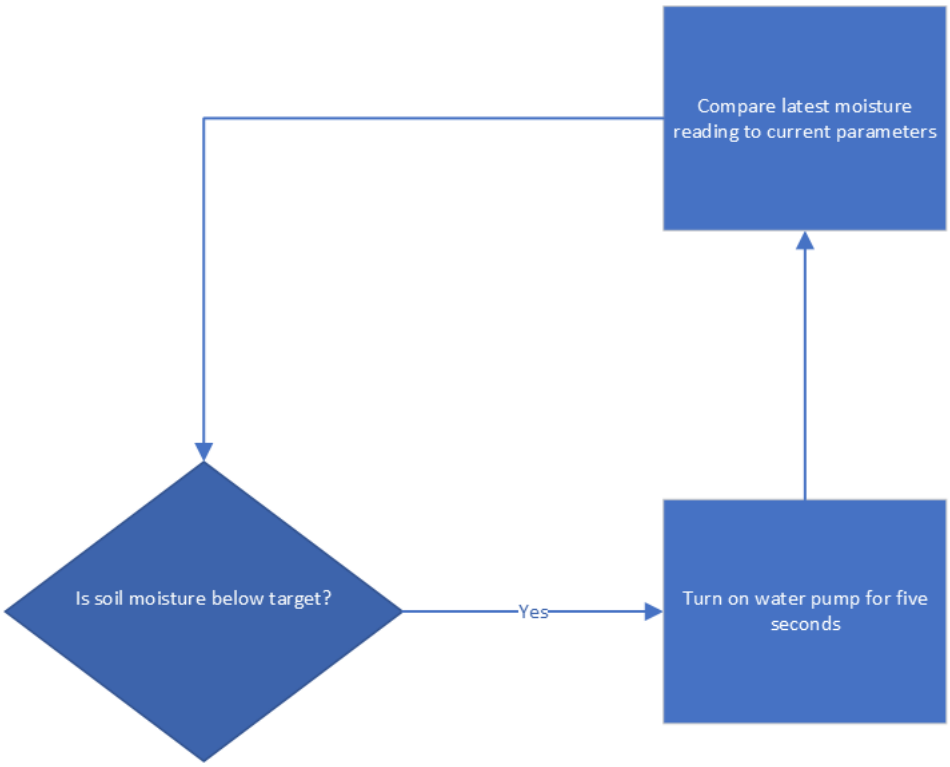


Figure 7: Water Pump Logic

Figure 7 shows the logic used to turn the water pump on and off. Rather than being directly controlled by the end user, it is automatically actuated by comparing the most recent sensor reading to the current parameters set by the user. The verbiage of the flowchart has been subtly changed from the original version to better reflect the logic used by the device to determine whether the pump should be turned on – see Appendix A for the original.

3 PROJECT MANAGEMENT

3.1 PROJECT PHASES

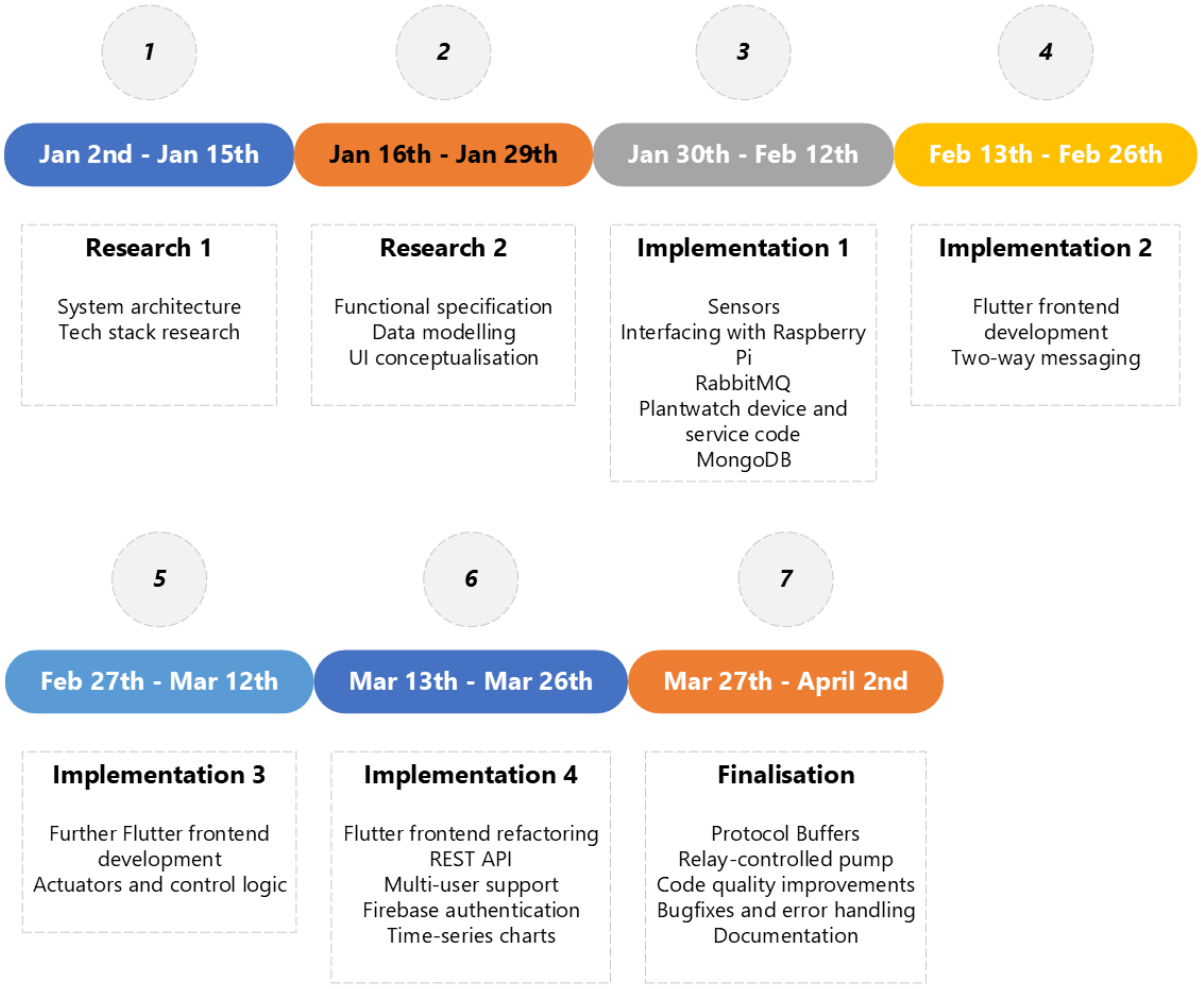


Figure 8: Project phases diagram

3.1.1 SPRINT ONE

During the first sprint, I conceptualized how the system would be architected and conducted research on the technology stack that I would use for the project. It was during this phase that I researched and decided on which sensors and actuators I would use, as well as deciding on MongoDB for data storage, RabbitMQ for lightweight messaging and Flutter for the frontend.

See Appendix A for graphics indicating how the system architecture was conceived at this stage of development.

3.1.2 SPRINT TWO

During the second sprint, I developed the functional specification of how the system would work and what functionality it would expose to the end user. I also conceptualised how the data would be modelled and how the UI would look to the end user.

See Appendix A for graphics indicating how the functional specification and data modelling were conceived at this stage.

3.1.3 SPRINT THREE

During the third sprint, project implementation began. I interfaced the sensors with the Raspberry Pi and using the Grove Python libraries to read data from them.

As I had some difficulty identifying the correct library to use for the temperature and humidity sensor, I referred to a Stack Overflow answer that directly implemented the necessary functionality (ultracold, 2021). With my project supervisor's guidance, this was an opportunity to learn something about the low-level details of the I2C protocol. I was later able to identify the correct library to use, but this was a valuable learning experience regardless.

During this phase I also deployed a RabbitMQ server for message brokering, a MongoDB database server, and wrote the first implementations of the sensor device code and the worker service code.

3.1.4 SPRINT FOUR

During the fourth sprint, I began work on the frontend application. This involved learning the basics of Flutter and developing a proof-of-concept that the user would be able to set parameters in the frontend application and have those passed as control data to the device.

3.1.5 SPRINT FIVE

During the fifth sprint, I further developed the frontend application, working to create a UI based on the conceptualisation developed during sprint two. At this point, I also introduced logic on the device to control actuators based on parameters set by the user and passed to the device as control data. At this point the actuators were all represented as LEDs.

3.1.6 SPRINT SIX

During the sixth sprint, significant refactoring of the frontend application took place, as well as changes to the overall architecture of the system.

I began by introducing multi-user support using Firebase Authentication. Up until now, the frontend application had been interacting directly with the MongoDB database to read and write data. Once multi-user support was implemented, it occurred to me that exposing the database directly to the frontend application was not secure or best practice.

On this basis, and on the advice of my supervisor, I set about implementing a REST API in Dart to facilitate secure database access in line with best practices. The frontend application was refactored to read and write data via HTTP requests to the REST API. Using Firebase Authentication and associated libraries, I secured the API routes by passing a JSON web token with the API calls, and having the API validate this token before responding.

During this sprint, I also implemented time-series charts of device readings, using the Google charts library for Flutter.

3.1.7 FINALISATION

During the finalisation phase, I swapped out the LED representing the water pump with an actual relay-controlled water pump and updated the control logic accordingly. I had left this task until late in development as it involved working with a higher voltage device as well as running water, and I was conscious of the need not to damage my equipment mid-development.

I also composed Protocol Buffer schemas, compiled them, and used them in the device and service code to serialize sensor and control data that were previously serialized as JSON.

This phase also involved a significant amount of time spent on improving the readability and robustness of the project code, fixing bugs, and writing documentation. Finally, during this phase I deployed the project infrastructure to Google Cloud Platform.

3.2 DEVELOPMENT TOOLS

3.2.1 PROXMOX AND LINUX CONTAINERS

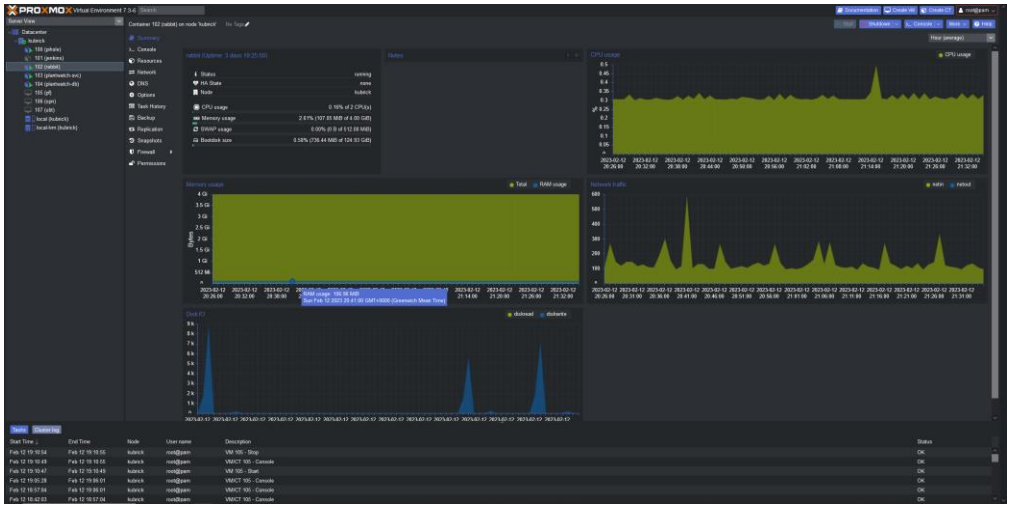


Figure 9: Proxmox Virtual Environment web UI

As the system was designed as a decoupled system with multiple components, I needed a way to run multiple servers during development. I used a self-hosted instance of Proxmox Virtual Environment to host the various components of the project during development.

Proxmox Virtual Environment is "*a powerful open-source server virtualization platform to manage two virtualization technologies - KVM (Kernel-based Virtual Machine) for virtual machines and LXC for containers - with a single web-based interface*" (Proxmox Server Solutions, 2023).

During development, most of the project components were hosted in LXC containers on Proxmox. LXC containers "*are often considered as something in the middle between a chroot and a full-fledged virtual machine. The goal of LXC is to create an environment as close as possible to a standard Linux installation but without the need for a separate kernel*" (Linux Containers, 2023).

Running the application components in their own LXC containers allowed me to simulate a networked, distributed cloud deployment on my own local server. Furthermore, Proxmox is easily configured to deploy LXC containers using templates from the Turnkey Linux library. Turnkey Linux is "*a free Debian based library of system images that pre-integrates and polishes the best free software components into secure, easy to use solutions*" (Turnkey Linux, 2023). I deployed my MongoDB server using this method, which saved time and allowed me to focus on development.

The ability to quickly stand-up virtual servers on Proxmox was enormously helpful during the development process. For example, when late in development I realised the need for a REST API to interface between the frontend and the database server, I was able to quickly stand-up a Debian server in an LXC container, install the Dart runtime, and begin developing the API. Running my own development server also meant that I did not need to consume expensive cloud resources prior to deploying the project.

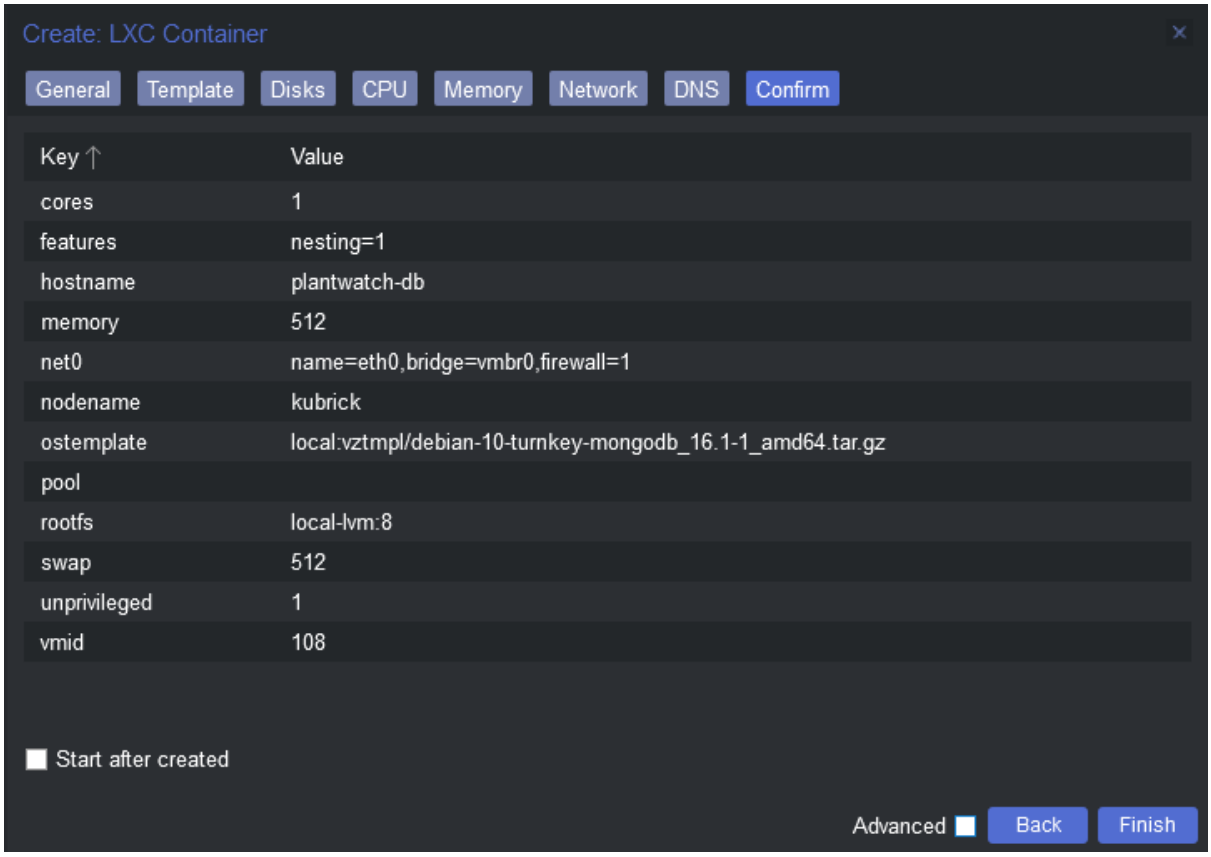


Figure 10: Deploying MongoDB in an LXC container on Proxmox

3.2.2 VS CODE

Visual Studio Code was the natural choice for developing this project. Its Visual Studio Code Remote - SSH extension facilitates remote development on both the Raspberry Pi and the LXC Containers (Visual Studio Code, 2023). As well as having support for developing in Python, it also has full support for developing and debugging Flutter applications (Flutter, 2023). This meant that development of both the front-end and back-end components could be carried out in the same editor.

3.2.3 MODELLING TOOLS

I used various diagramming and modelling tools to model the project. The system diagram was drawn using Lucidchart, the entity–relationship model was drawn using Visual Paradigm, the pump logic flowchart was drawn using Microsoft Visio, and the early UI wireframes were drawn using Mockplus. Each tool was chosen for its suitability to the task I needed to complete.

4 IMPLEMENTATION

4.1 SENSOR DEVICE

The Raspberry Pi is connected to a DHT20 temperature and humidity sensor (Seeed Studio, 2023) and an SI1151 sunlight sensor (Seeed Studio, 2023). These are both I2C devices that communicate over the Raspberry Pi's GPIO pins. Third-party code has been used to read data from these, but I have endeavoured to understand the fundamentals of the I2C protocol. I2C communicates data in 8-bit data frames (Circuit Basics, 2016), which require bitwise masking operations to convert to familiar data types such as integers and floats. A soil moisture sensor is also connected, which measures voltage to indicate soil resistivity, and hence moisture content (Seeed Studio, 2023).

The software running on the Raspberry Pi is a Python program that at regular intervals reads data from these sensors, stores the temperature, humidity, soil moisture, and UV index in a Python dictionary together with a reading id, a device id, and a timestamp, and uses the Pika Python library (Read the Docs, 2023) to publish that data to RabbitMQ as an AMQP message. It also reads AMQP messages on another channel to receive target values for the environmental conditions, which it uses to set the state of its actuators.

The device also publishes a heartbeat message to RabbitMQ at regular intervals. This allows for the fact that it may be desirable to reduce the frequency of readings being taken from the device sensors; by implementing the heartbeat as a separate message, it becomes possible to reduce the frequency of sensor readings while still receiving a heartbeat as confirmation that the device is online and functional.

During development, the various actuators were represented as LEDs. During the finalisation phase, I implemented support for real water pump, controlled by a normally open relay (Seeed Studio, 2023). The other actuators are still represented as LEDs.

The code for the sensor device can be found in *plantwatch_sensor.py* in the *plantwatch_python/* directory of the project repository.

4.2 MESSAGING

4.2.1 RABBITMQ

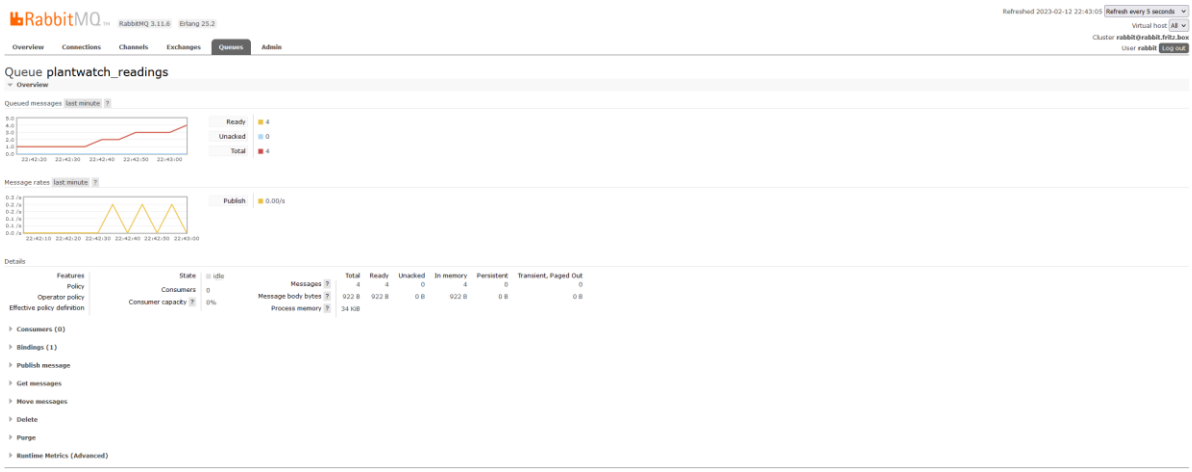


Figure 11: RabbitMQ web UI showing activity on our message queue for sensor readings

Rabbit MQ is a message broker, and its role in this project is to facilitate a decoupled design by passing messages between the Raspberry Pi sensor device and a worker service that interfaces with the database. This decoupling is a common use case for Rabbit MQ (RabbitMQ, 2023).

Messages containing sensor readings are published by the sensor device and consumed by the worker service, which logs the readings to the database. The worker service also checks the database to determine if parameters have been set for the device, and if so, publishes a message containing those parameters, which is consumed by the sensor device.

Using messaging to decouple the sensor device from the database in this way makes the design more scalable and allows for more efficient serialization of data being transmitted to and from the sensor device, as discussed in the next subsection.

To deploy RabbitMQ, I followed the RabbitMQ's guide on Installing on Debian and Ubuntu (RabbitMQ, 2023).

4.2.2 PROTOCOL BUFFERS

During the initial stages of development, the messages being sent over RabbitMQ contained sensor data and control data that was serialised simply by encoding JSON objects as UTF-8 strings. This was convenient for development purposes as it was easily implemented and the data was human-readable, which was helpful for debugging.

Later in development, I decided to serialize the data using protocol buffers ('protobuf') instead of JSON. As per the documentation, *"Protocol buffers provide a language-neutral, platform-neutral, extensible mechanism for serializing structured data in a forward-compatible and backward-compatible way. It's like JSON, except it's smaller and faster, and it generates native language bindings. You define how you want your data to be structured once, then you can use special generated source code to easily write and read your structured data to and from a variety of data streams and using a variety of languages"* (Google, 2023).

Implementing serialization using protobuf involved defining message schemas in *.proto* files and using these to generate Python classes which are imported in *plantwatch_sensor.py* and *plantwatch_service.py* and used to serialize and deserialize the data. The protoc compiler used to generate the Python classes and documentation on how to do so are provided by Google (Google, 2023).

Efficient serialization of data can be an important concern for IoT applications, as IoT devices are often deployed to remote locations, and bandwidth may be constrained, for example if using a low-power wide area network (LPWAN) such as Sigfox, which limits payload sizes to 12 bytes for uplink and 8 bytes for downlink (Sigfox, 2023).

```
INFO:root:Took a reading: {'_id': '642a252d02bdb9342c587105', 'deviceId': '642a252802bdb9342c587103', 'timestamp': '2023
-04-03 01:00:29.614435', 'temperature': 19.614982604980 47, 'humidity': 59.59157943725586, 'moisture': 0, 'uvIndex': 0.2}
```

Figure 12: Sensor device log entry showing a reading in JSON format, prior to serialization using protobuf

```
INFO:root:Received a reading: b'\n\x18642a252d02bdb9342c587105\x12\x18642a252802bdb9342c587103\x1a\x1a2023-04-03 01:00:2
9.614435%|\xeb\x9cA~\xc7]nB5\x00\x00\x00\x00=\xcd\xccL>'
```

Figure 13: Worker service log entry showing the same reading, received in protobuf-serialized format

Figure 12 shows a log entry from the sensor device, illustrating the length of a reading message in JSON format, prior to being serialized using protobuf. Figure 13 shows a log entry from the worker service after receiving the same reading, which was protobuf-serialized before being sent over RabbitMQ. This illustrates the shorter length of the protobuf-serialized message.

The protobuf schemas are found in the *plantwatch_protobuf/* directory of the project repository, and the compiled Python classes are found in the *plantwatch_python/* directory.

4.3 PLANTWATCH SERVICE

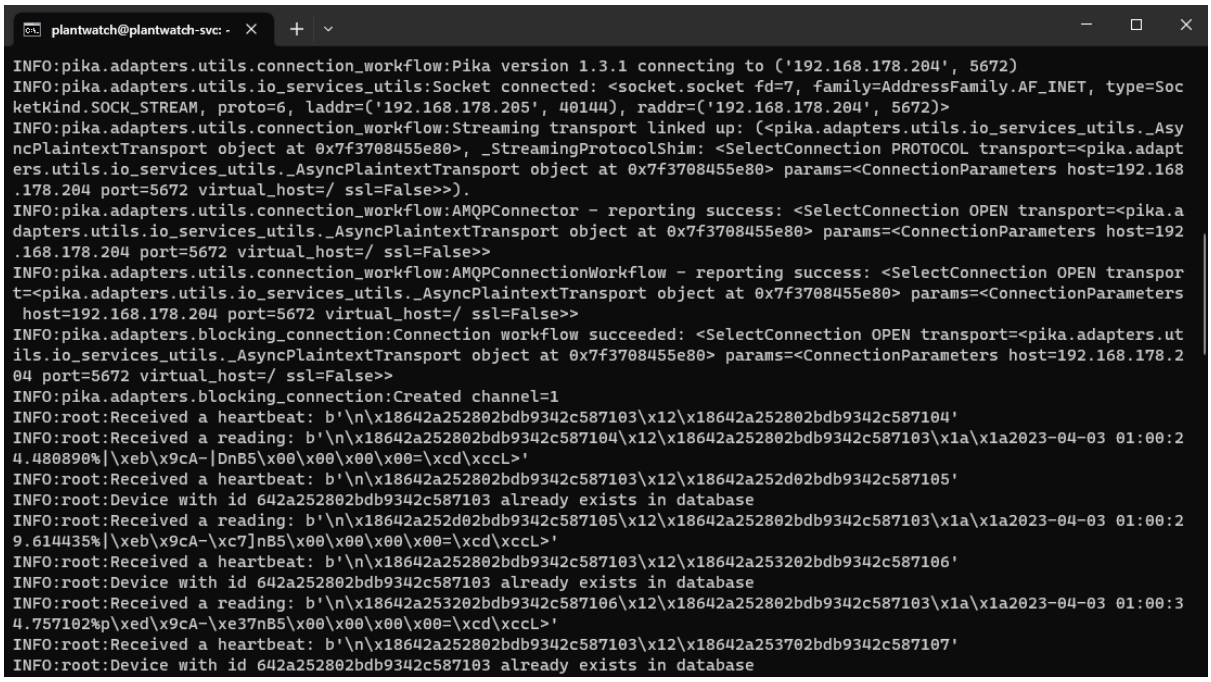


Figure 14: Worker service log output

The worker service is a Python program that reads and writes data to and from the MongoDB database and consumes and publishes messages to and from RabbitMQ.

When a message is consumed containing a reading, that reading is logged to the database. When a message containing a heartbeat from a device is received, that device is updated in the database with the id of its latest reading, and if user-specified parameters are found in the database for that device, a command message is published containing those parameters, which will be consumed by the sensor device.

The code for the worker service can be found in *plantwatch_service.py* in the *plantwatch_python/* directory of the project repository.

4.4 MONGODB

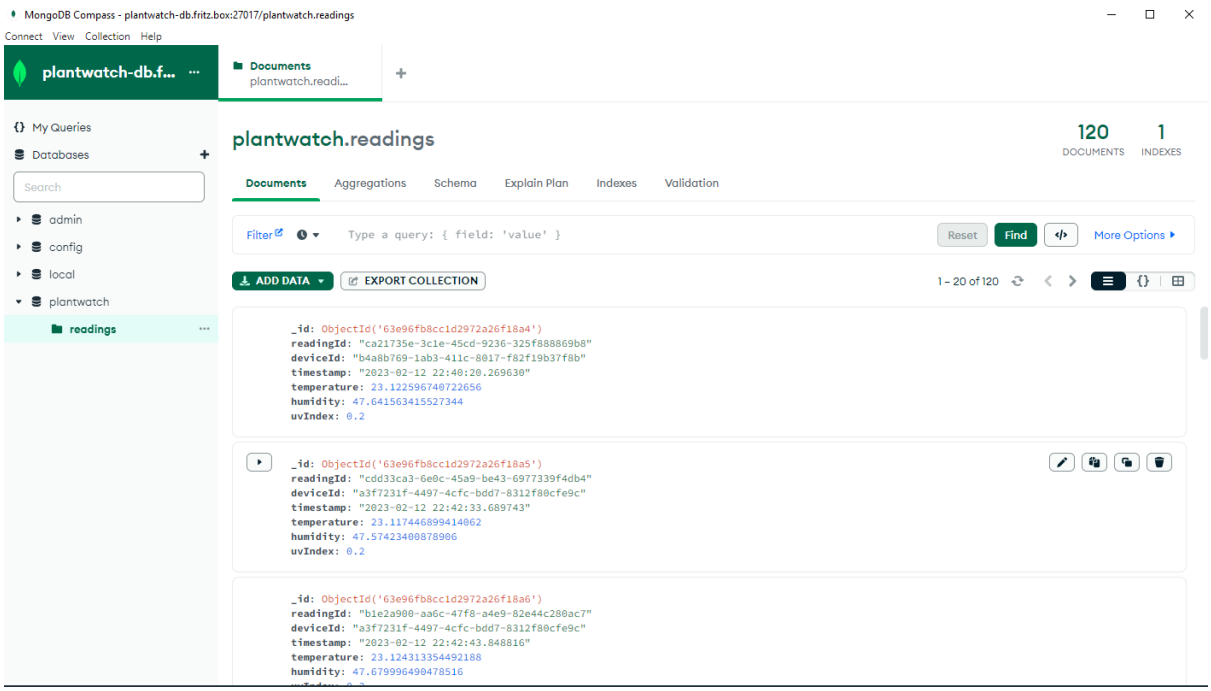


Figure 15: MongoDB Compass displaying reading data as stored in the database

The MongoDB database is read and updated by the worker service and the REST API. It contains two collections, one for readings and one for devices. The structure of the data is described in section 2.3.3 of this report.

MongoDB was chosen for its flexible nature as a document-oriented database, which was helpful in using a separate service (Firebase) for storing user entities, as well as being able to change the structure of the data as the project was developed.

4.5 PLANTWATCH API

I have implemented a REST API in Dart to allow safe communication between the frontend application and the database.

The API endpoints allow the frontend application to retrieve reading and device information, set device parameters, and associate a device with a user account. Security of the API is managed by using the Firebase admin SDK to validate JSON web tokens sent by the frontend application.

In developing the API, I referred to existing repositories for guidance. The MongoDB-Flutter repo by Harshvardhan Shinde (Shinde, 2021) was helpful in the early design stages when the frontend application was interacting directly with the database, and the dart_mongo repo by Jermaine Oppong (Oppong, 2019) was referred to when reimplementing this functionality to use a REST API.

The ability to validate JWTs with Firebase relies on a Dart implementation of the Firebase admin SDK by Rik Bellens (Bellens, 2022).

The code for the API can be found in the *plantwatch_api/* directory of the project repository.

4.6 FLUTTER

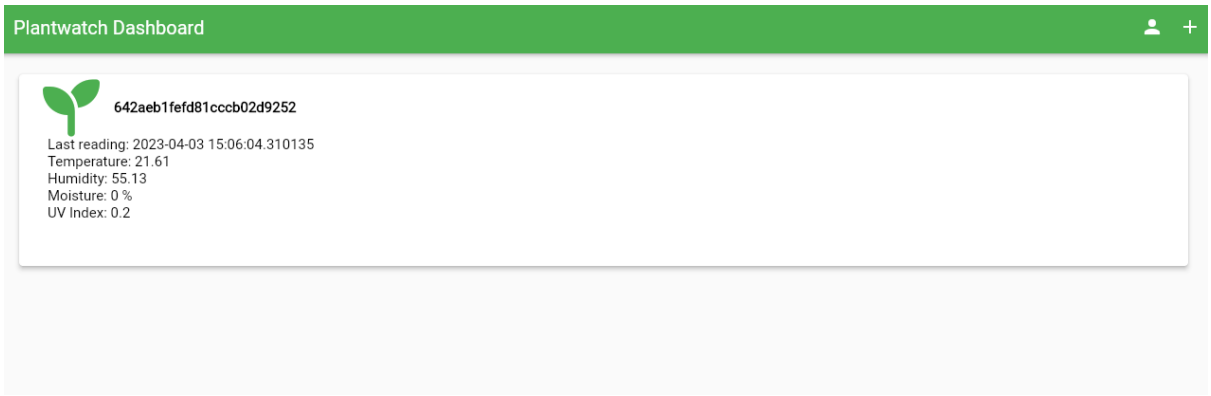


Figure 16: Dashboard UI on web

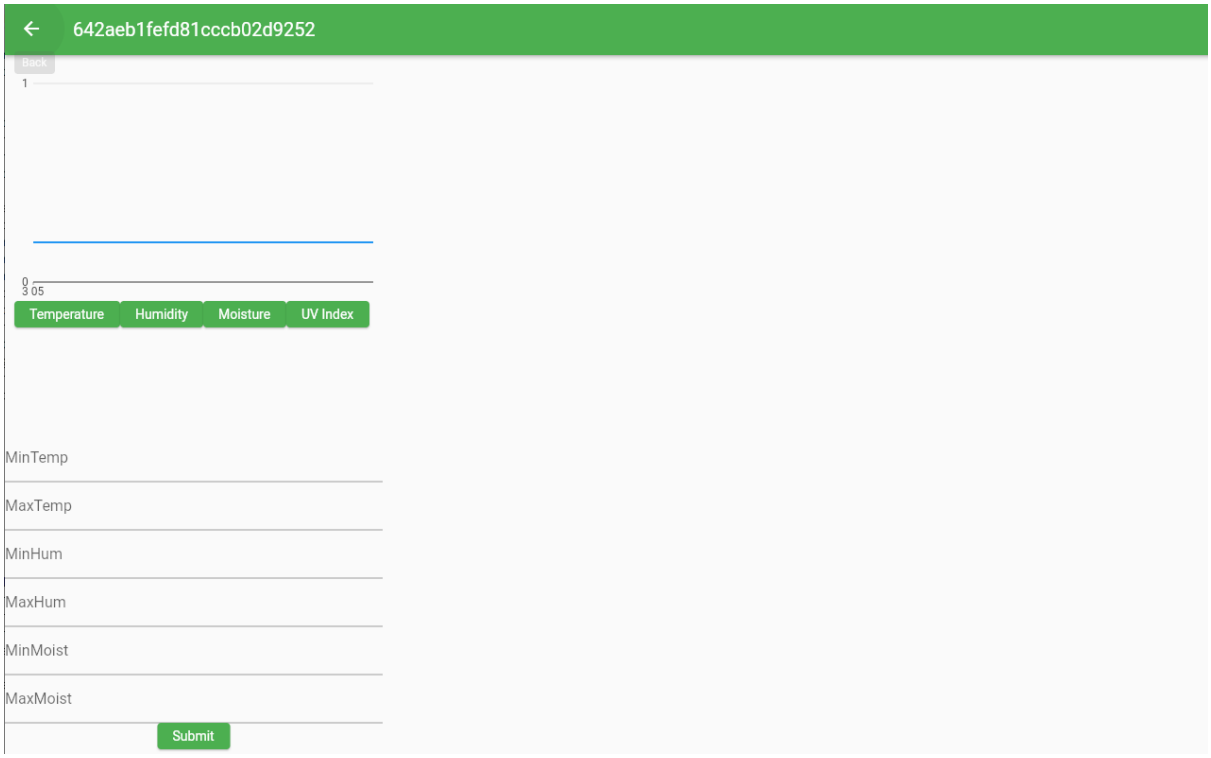


Figure 17: Device view UI on web

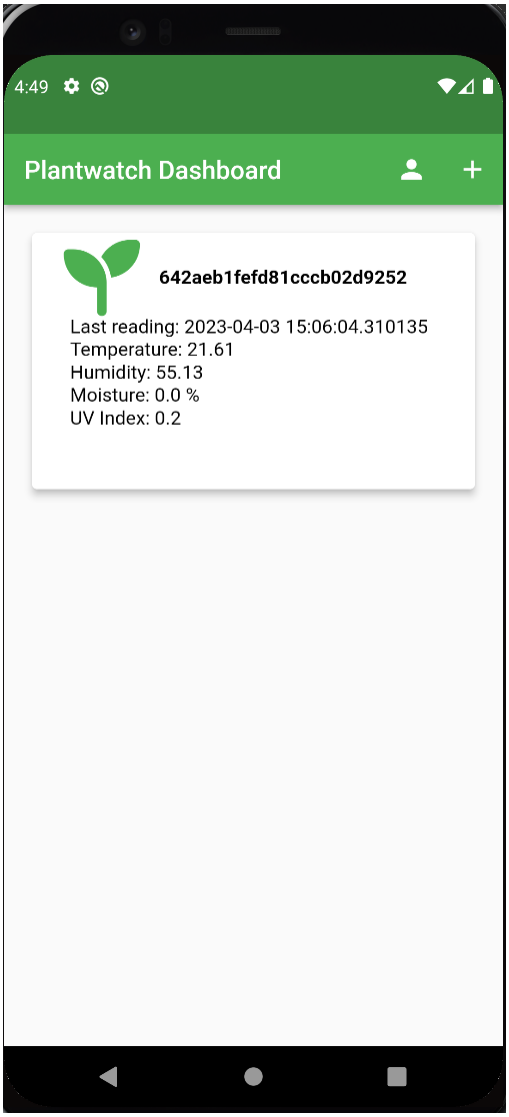


Figure 18: Dashboard UI on mobile

For the frontend of the project, I have built a hybrid web and mobile app in Flutter. It uses Firebase UI components to facilitate user registration and login. Logged in users are presented with a dashboard of devices associated with their account, which they can open to view charts of historical reading data and set device parameters. Users can also enter a device's object ID to associate it with their account, which they must do before being able to interact with it.

HTTP requests to the REST API are used to send and receive data. This is implemented using the *http* dart package (dart.dev, 2022). The application can be run as a web application or an Android application. The Google charts library is used to draw time-series charts (Google, 2022).

When first learning Flutter, I referred to the tutorial *How to Build a Flutter Card List in Less Than 10 Minutes* from DLT Labs (DLT Labs, 2019). This provided a useful reference for a basic card list interface.

Figures 16-18 illustrate how the UI currently looks on web and mobile. This differs somewhat from the UI as originally conceived in the modelling phase, largely because working with Flutter and developing a hybrid application naturally leads to the adoption of UI styles familiar to mobile applications. With more development time, the web UI could be improved and made to look more like the original wireframes.

The code for the frontend application can be found in the *plantwatch_flutter/* directory of the project repository. The parts implemented by can be found in *plantwatch_flutter/lib/*, with separate subdirectories for the API requests, data models, UI components, pages, and utilities.

4.7 FIREBASE AUTHENTICATION

Firebase authentication using Firebase UI was implemented according to the tutorial *Add a user authentication flow to a Flutter app using FirebaseUI* by Eric Windmill (Windmill, 2022). Firebase UI provides out-of-the-box support for typical user account functions, such as signup, login, logout, and delete account.

Firebase authentication is integrated with the rest of the application using Firebase's ability to get and verify a JSON web token for the logged currently logged-in user. Once the user is logged in, the frontend can pass a JSON web token in the Authorization header of its API requests. The REST API is designed to verify this token using the Firebase admin SDK before responding, and to return an appropriate 403 "Not authorised" response status code if the token is missing or invalid.

4.8 FIREBASE AND GCP DEPLOYMENT

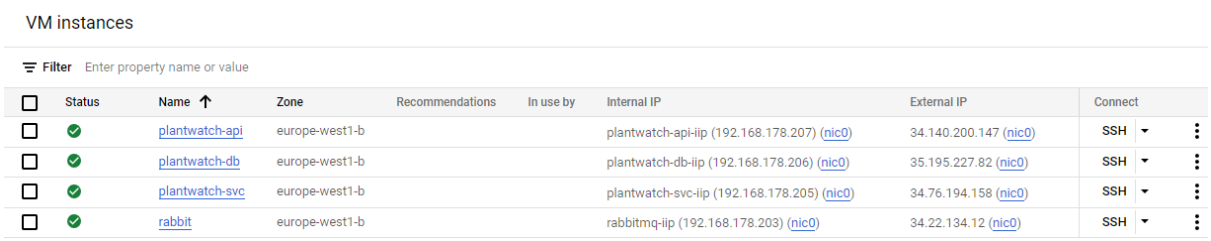


Figure 19: Project infrastructure running on GCP

Name	Type	Targets	Filters	Protocols/ports	Action	Priority	Network
plantwatch-allow-8085	Ingress	plantwatch-allow-8085	IP ranges: 0.0.0.0/0	tcp:8085	Allow	1000	plantwatch
plantwatch-allow-all-internal-tcp-udp-icmp	Ingress	Apply to all	IP ranges: 192.168.178.0/24	all	Allow	1000	plantwatch
plantwatch-allow-amqp	Ingress	plantwatch-allow-amqp	IP ranges: 0.0.0.0/0	tcp:5672	Allow	1000	plantwatch
plantwatch-allow-ssh	Ingress	plantwatch-ssh	IP ranges: 0.0.0.0/0	tcp:22	Allow	1000	plantwatch

Figure 20: Firewall rules configured in GCP VPC

As a final step prior to demonstration, I deployed the project infrastructure to Google Cloud Platform. This was a case of replicating the infrastructure that existed on my local development server in a GCP Virtual Private Cloud. I created Debian virtual machines on GCP Compute Engine, applied appropriate firewall rules in GCP VPC, and configured the project infrastructure as I had done on my local server.

In the early stages of the project, I had planned to use Microsoft Azure for cloud deployment, but as I was already using Google's platform for Firebase Authentication, it made sense to pivot to GCP for deployment.

It is also possible to deploy the web version of the Flutter application to Firebase Hosting using the Firebase CLI tools, meaning that all the deployed resources are neatly housed in the same Firebase/GCP project.

```
=== Deploying to 'plantwatch-46cfe'...

i  deploying hosting
i  hosting[plantwatch-46cfe]: beginning deploy...
i  hosting[plantwatch-46cfe]: found 31 files in build/web
+  hosting[plantwatch-46cfe]: file upload complete
i  hosting[plantwatch-46cfe]: finalizing version...
+  hosting[plantwatch-46cfe]: version finalized
i  hosting[plantwatch-46cfe]: releasing new version...
+  hosting[plantwatch-46cfe]: release complete

+  Deploy complete!

Project Console: https://console.firebase.google.com/project/plantwatch-46cfe/overview
Hosting URL: https://plantwatch-46cfe.web.app
```

Figure 21: Deploying to Firebase Hosting using the Firebase CLI tools

5 CRITICAL SELF-REVIEW

5.1 LEARNING OUTCOMES

While working on this project, I gained exposure to several new technologies, of which I had previously held no or very limited knowledge. This included the specific physical sensors and actuators used in the project, RabbitMQ, protocol buffers, Dart, and Flutter. Not only did I learn to use these technologies, but I also learned to integrate them into a single IoT solution.

I am pleased to say that I was able to draw on and expand skills learned in all the previous course modules while working on this project. Programming fundamentals skills were used at all levels of the technology stack; full-stack web development, software security, and reactive frameworks skills were used to implement the REST API and frontend application; physical computing, IoT, computer systems and networks skills were used to build and implement the sensor device; database design and implementation skills were used to model and store the application data; developer operations skills were helpful in deploying to Google Cloud Platform and in configuring and running my own local development server with Proxmox; mobile app development skills were helpful in implementing Firebase authentication and debugging the Android build of the frontend application.

Although I drew directly on the skills learned in previous course modules, I also went beyond them by using the new technologies mentioned above. This was an opportunity to critically apply the skills learned in the course modules to independently learning new technologies. General skills such as project management, modelling, testing, and debugging were essential to successfully building a solution using these new technologies.

In terms of project management, this project was significantly greater in scope and duration than any of module assignments. Learning to manage a project over the course of several months, and to handle changes in direction while working to a timeline, was another significant learning outcome of working on this project.

Working with my project supervisor was essential not only in managing the project from start to finish, but also in developing my understanding of some of the new technologies I was being exposed to. For example, although I used the grove.py libraries from Seeed Studio interface the sensors, I nonetheless needed to learn the basics of the I2C serial communication protocol to understand how the grove.py libraries were working and to troubleshoot issues with them. My supervisor directed me towards helpful resources in learning about the I2C protocol and gave me a detailed explanation during one of our meetings. The outcome of this was that I was able to develop my understanding of the low-level technical details of I2C, and I was even able to assist another GitHub user in troubleshooting an issue with the grove.py libraries – see my comments on issue #2 on the Seeed_Python_SI114X GitHub repo (Kelly, 2023).

5.2 PROJECT ACHIEVEMENTS

I am pleased to say that the project has achieved the objectives set out in the project proposal. The proposal was for an IoT plant health monitoring system using Raspberry Pi, sensors, a water pump, and Python for the IoT aspect, MongoDB for data storage, and Flutter and Dart for the frontend application. This is what has been implemented.

This is not to say that the project does not in some ways differ from what I initially envisioned. RabbitMQ, protocol buffers and Firebase authentication were technologies that I settled on during the development process. I regard it as a significant achievement that I was able to integrate new technologies into the project when the project goals called for it.

A significant change occurred late in development when I realised that the approach I had taken of having the frontend application interact directly with the database was not secure. At this point, I decided to implement a REST API using JSON web tokens for authorization. This occurred during the sixth sprint and in part was

successful because, on my supervisor's advice, I chose to implement the REST API in Dart and thus was able to refactor some of my database code from the frontend application to a REST API solution. I regard this as a major success because I was successfully able to change course late in development to address a critical security concern.

Further development of the project would focus on meeting the necessary requirements to commission the project as a consumer product. To release the product to market, improvements to user experience would be required. Users would need to be able to add a device to their account without entering a lengthy Object ID, for example. A simple way to improve user friendliness in this respect would be to add a screen to the device which would display a shorter code, or a scannable QR code that would allow the user to associate the device with their account. A more complex, but more robust solution, would be to use Bluetooth pairing with a mobile device for this.

Another requirement that would need to be met before the project would be suitable for the consumer market would be improved security. For example, the sensor device currently authenticates to RabbitMQ with credentials stored on the device. This would be a security risk in a consumer product. A possible solution I have considered for this would be to automate the creation of a unique RabbitMQ account associated with every end-user of the application. Behind the scenes, the application would authenticate to RabbitMQ with this account, and the sensor device would not commence communication with RabbitMQ until associated with a user account. This would also require implementation of the Bluetooth pairing method of associating sensor device with a user account, as there would need to be a way to set unique RabbitMQ credentials on the device without requiring the user to do this manually.

Applications of a more developed version of this system could include home gardening, or more commercial applications such as greenkeeping or industrial agriculture. As the system allows multiple devices to be controlled from one dashboard, it could potentially be used in quite large-scale applications.

5.3 PROBLEMS ENCOUNTERED

Problems encountered early in development mostly related to interfacing the Grove sensors with the Raspberry Pi. Although libraries are provided by Seeed Studio, the documentation is limited and there are multiple versions of some of the sensors, leading to a lack of clarity on which library to use in some cases. As mentioned in section 5.1, once I learned the basics of the I2C protocol it was easier to identify the correct library and I was even able to answer another GitHub user's question on the topic.

A minor but concerning problem that occurred early in development was that, after interfacing multiple sensors with the Raspberry Pi, I noticed that the power LED was intermittently flickering. Checking the system logs (in /var/log/syslog) confirmed that this was because the system was going undervoltage at times. This was concerning because of the potential that device could be damaged. Although unlikely, if this were to occur it would have major implications for the viability of the project, given the current scarcity of Raspberry Pi devices. After discussing with my project supervisor, I replaced the 5V 3A power supply I had been using with a 5.1V 3A power supply, and this resolved the problem.

As mentioned above, there was a security issue with the original architecture of the system, which had the frontend application communicating directly with the database. This was resolved by introducing a REST API and JWT-based authorization as described in section 4.5.

Other problems encountered late in development mostly related to developing and debugging the frontend application. I found Flutter challenging to develop with. Although I had previous experience of similar reactive frameworks from the ICT Skills Studio 2 and Mobile App Development modules, developing the frontend application in Flutter was nonetheless challenging and time-consuming. I think this was largely because issues with this kind of framework are often difficult to debug. Their complexity can mean that build error messages are not always instructive as to the root cause of a problem, and when working with a REST API it's not always clear whether the root cause is a backend or a frontend issue. These problems were addressed by using the available debugging tools in VS Code and Chrome, by developing the frontend application and the REST API in parallel, and by approaching issues with patient and methodical troubleshooting.

A particular problem encountered during development of the frontend application was the implementation of time-series charts showing historical sensor reading data. I had initially thought that this would be straightforward to implement by passing reading data to components provided by the Google charts library, but that was not the case. Successfully implementing these charts involved developing a set of helper functions that would take a list of device readings and return a "Series" object that could be passed to the "TimeSeriesChart" component from the charts library. This in turn required developing some new data classes, each of which stores a single sensor metric (temperature, humidity, moisture, or uvIndex) together with a timestamp, and can be used to create the "Series" object. The mentioned helper functions can be found in the *plantwatch_flutter/lib/utils/seriesHelper.dart* file in the project repository, and the mentioned data classes can be found in the *plantwatch_flutter/lib/models/* directory. This approach was based on the example provided by Google in the documentation for the charts library (Google, 2023).

With respect to the above problems, and all of the more minor problems encountered during development, regular contact with my project supervisor was essential to resolving them effectively and efficiently. When I encountered problems, I would relay them to my supervisor and seek his thoughts on the best way to resolve them. Even in cases where I was confident in my own ability to find a solution, it was always helpful to have a discussion with my supervisor, as it provided additional perspective and direction.

5.4 CONCLUSION

In summary, I am satisfied that this project achieved its goals and was a highly valuable learning experience. I wanted to learn new technologies, integrate elements of previous course modules, and develop an IoT solution that was relevant to an area of interest to me, and I succeeded in doing all those things.

6 BIBLIOGRAPHY

Kelly, H. [htkelly] (2022) Watchful Pi [GitHub]. Available at: https://htkelly.github.io/watchfulpi (Accessed: 6 November 2022).

Seeed Studio (2022) Grove Smart Plant Care Kit. Available at: https://wiki.seeedstudio.com/Grove_Smart_Plant_Care_Kit (Accessed: 6 November 2022).

Seeed Studio (2023) Grove Base Hat for Raspberry Pi. Available at: https://wiki.seeedstudio.com/Grove_Base_Hat_for_Raspberry_Pi (Accessed: 17 January 2023).

Seeed Studio [Seeed-Studio] grove.py [GitHub]. Available at: https://github.com/Seeed-Studio/grove.py (Accessed: 3 April 2023).

McLellan, Charles (2020) The Internet of Wild Things: How the IoT joined the battle against climate change. Available at: https://www.zdnet.com/article/the-internet-of-wild-things-technology-and-the-battle-against-biodiversity-loss-and-climate-change (Accessed: 2 April 2023).

Copeland, Rick (2013) MongoDB Applied Design Patterns, Ch. 1. Available at: https://www.oreilly.com/library/view/mongodb-applied-design/9781449340056/ch01.html (Accessed: 2 April 2023).

[ultracold] (2021) Stack Overflow Answer #133487 [StackOverflow]. Available at: https://raspberrypi.stackexchange.com/a/133487 (Accessed: 3 April 2023).

Proxmox Server Solutions (2023) Features. Available at: https://proxmox.com/en/proxmox-ve/features (Accessed: 12 February 2023).

Linux Containers (2023) Introduction. Available at: https://linuxcontainers.org/lxc/introduction/ (Accessed: 12 February 2023).

Turnkey Linux (2023) About Turnkey GNU/Linux. Available at: https://www.turnkeylinux.org/about (Accessed: 12 February 2023).

Visual Studio Code (2023) Remote Development using SSH. Available at: https://code.visualstudio.com/docs/remote/ssh (Accessed: 12 February 2023).

Flutter (2023) Set up an editor. Available at: https://docs.flutter.dev/get-started/editor?tab=vscode (Accessed: 12 February 2023).

Seeed Studio (2023) Grove – Temperature & Humidity Sensor (DHT20). Available at https://wiki.seeedstudio.com/Grove-Temperature-Humidity-Sensor-DH20 (Accessed: 12 February 2023).

Seeed Studio (2023) Grove – Sunlight Sensor. Available at: https://wiki.seeedstudio.com/Grove-Sunlight_Sensor (Accessed: 12 February 2023).

Circuit Basics (2016) Basics of the I2C Communication Protocol. Available at: https://www.circuitbasics.com/basics-of-the-i2c-communication-protocol (Accessed: 12 February 2023).

Seeed Studio (2023) Grove – Moisture Sensor. Available at https://wiki.seeedstudio.com/Grove-Moisture_Sensor (Accessed: 28 March 2023).

Seeed Studio (2023) Grove – Relay. Available at https://wiki.seeedstudio.com/Grove-Relay (Accessed: 2 April 2023).

Read the Docs (2023) Introduction to Pika. Available at: https://pika.readthedocs.io/en/stable/intro.html (Accessed: 12 February 2023).

RabbitMQ (2023) What can RabbitMQ do for you? Available at: https://www.rabbitmq.com/features.html (Accessed: 12 February 2023).

RabbitMQ (2023) Installing on Debian and Ubuntu. Available at: https://www.rabbitmq.com/install-debian.html#manual-installation (Accessed: 2 April 2023).

Google (2023) Protocol Buffers Documentation: Overview. Available at: https://protobuf.dev/overview (Accessed: 3 April 2023).

Google (2023) Protocol Buffer Basics: Python. Available at: https://protobuf.dev/getting-started/pythontutorial (Accessed: 2 April 2023).

Sigfox (2023) Payload. Available at: https://build.sigfox.com/payload (Accessed: 2 April 2023).

Shinde, Harshvardhan [harshshinde07] (2021) MongoDB-Flutter [GitHub]. Available at: https://github.com/harshshinde07/MongoDB-Flutter (Accessed: 28 March 2023).

Oppong, Jermaine [graphicbeacon] (2019) dart_mongo [GitHub]. Available at: https://github.com/graphicbeacon/dart_mongo (Accessed: 28 March 2023).

Bellens, Rik [rbellens] (2022) firebase_admin. Available at: https://github.com/appsup-dart/firebase_admin (Accessed: 28 March 2023).

Dart.dev (2022) http documentation. Available at: https://pub.dev/packages/http (Accessed 3 April 2023).

Google (2022) charts. Available at: https://github.com/google/charts (Accessed: 28 March 2023).

DLT Labs (2019) How to Build a Flutter Card List In Less Than 10 Minutes. Available at https://medium.com/dlt-labs-publication/how-to-build-a-flutter-card-list-in-less-than-10-minutes-9839f79a6c08 (Accessed: 28 March 2023).

Windmill, Eric (2022) Add a user authentication flow to a Flutter app using FirebaseUI. Available at: https://firebase.google.com/codelabs/firebase-auth-in-flutter-apps#0 (Accessed: 28 March 2023).

Kelly, Harry [htkelly] (2023) response to issue I2C error (Please check if the I2C device insert in I2C of Base Hat) [GitHub]. Available at: https://github.com/Seeed-Studio/Seeed_Python_SI114X/issues/2 (Accessed: 3 April 2023).

Google (2023) Simple Time Series Charts Example [GitHub]. Available at: https://google.github.io/charts/flutter/example/time_series_charts/simple (Accessed: 3 April 2023).

APPENDICES

APPENDIX A – EARLY-STAGE DESIGN AND MODELLING

ORIGINAL SYSTEM DIAGRAM

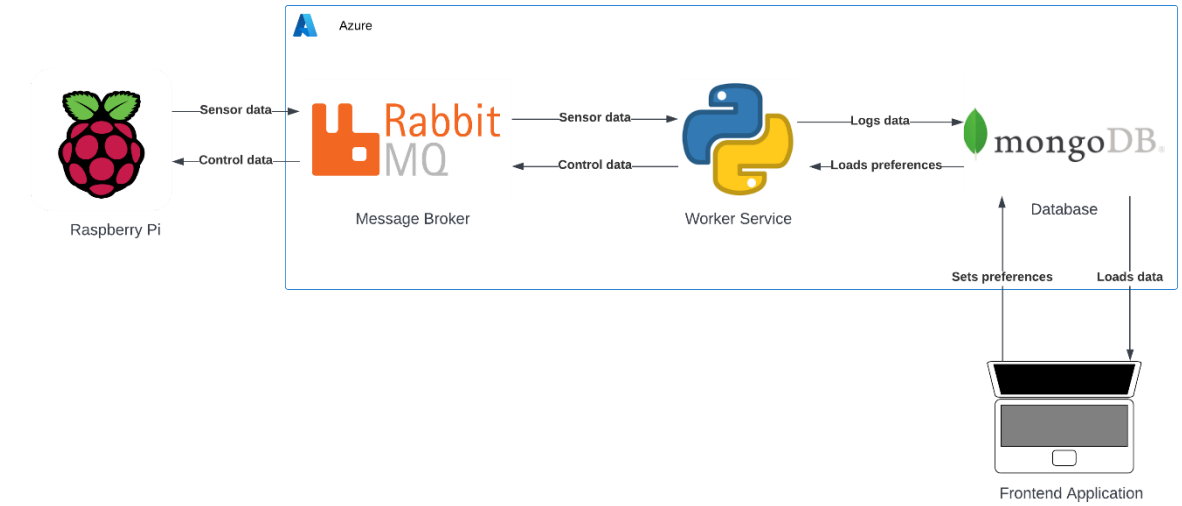


Figure 22: Original system diagram

ORIGINAL DATA MODEL

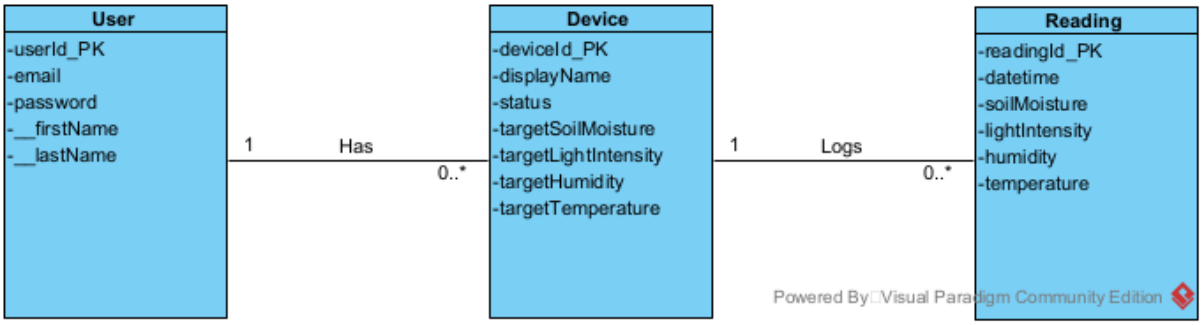


Figure 23: Original data model depicted as an entity–relationship diagram

ORIGINAL PUMP LOGIC FLOWCHART

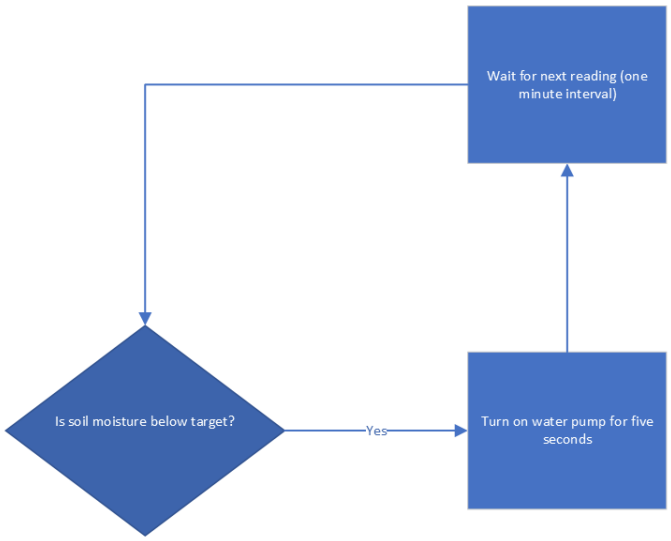


Figure 24: Original pump logic flowchart

APPENDIX B – ETHICAL APPROVAL CHECKLIST

See appended pages for ethical approval checklist.

School of Science and Computing Research Ethics Committee

Your responses

Summary | Individual responses | All your responses

Your 2 response(s)

1 | 2 | Next

Respondent: **Harry Kelly** (Group: CM-HDIPCS) Submitted on: Wednesday, 7 December 2022, 12:53 PM

Ethics Checklist for Undergraduate, Taught Postgraduate and Research Projects in the School of Science and Computing

All students in the School of Science and Computing who are either (1) in the final year of an undergraduate/BSc degree, or (2) on a taught postgraduate/MSc programme **must complete this Ethics Checklist before conducting their project** regardless of the project type or discipline. The Checklist should also be completed by anyone (whether staff member or student) conducting a **research project** (whether programmatic or not) within the School.

The purpose of this Ethics Checklist is to **identify projects that will require formal ethical approval** from the School Research Ethics Committee, or the SETU Research Ethics Committee, before they can proceed.

Students/applicants should note that this Ethics Checklist is a **formal declaration**, and great care must be taken to **answer all questions accurately**. Students should consult with their project supervisors/advisors regarding any aspects or questions that they are unsure of before completing and submitting the Ethics Checklist.

Students/applicants must **answer all questions** presented to them until the Checklist questionnaire is completed.

Feedback Report

No human experimentation issues (TPG/RP).

No animal experimentation issues (N/A).

No issues regarding the use of human tissues.

No animal tissue or biological fluids issues.

No ionising radiation issues.

No primary data collection issues (N/A).

No underage/vulnerable people issues (TPG/RP).

No issues regarding existing/secondary data use (N/A).

No controversial data issues.

No issues related to the collection of rare or protected plants.

No issues regarding the use of genetically modified (GM) plant material.

Instructions:

1. If the above feedback is **entirely** green then, based on your answers, there is **no need to apply for ethical approval** for your project.
2. If **any** part of the above feedback is yellow/amber, then there is at least one issue with your project that needs to be reviewed and **you must apply for ethical approval** to continue your project.
3. If **any** part of the above feedback is red then there is a serious ethical issue and **you cannot continue your project** as currently planned.

It is recommended that you print this Feedback Report to a PDF file for your records. You should also forward and discuss this Feedback Report PDF with your project supervisor. They will be able to advise if you have any further questions or if you need to apply for ethical approval.

1 * Are you a student on a **final year undergraduate** programme, a **taught postgraduate** programme, or are you conducting a **research project**?

- ○ Final Year Undergraduate
- ◉ Taught Postgraduate
- ○ Postgraduate Research Project
- ○ Other Research Project

2 * What is the **working title** of your project?

IoT Plant Health Monitoring System

3 * Who are the project **supervisors/advisors/principal investigators**?

Colm Dunphy, Peter Windle, TBC

4 * Does your project involve **human experimentation**? (e.g. clinical trials)

○ Yes ◉ No

5 * Does your project involve **animal experimentation**?

Yes No

(6) * Is the planned animal experimentation limited to **non-invasive procedures only** (such as feeding, weighing, or taking hair samples), and does **not** involve any invasive procedures (such as taking blood) from live animals?

Yes No

7 * Does your project involve the use of **human** remains/cadavers/tissues/cells/biological fluids/embryos/foetuses?

Yes No

(8) * Do you intend to only use established **commercial human cell lines**, and no other **human** remains/cadavers/tissues/cells/biological fluids/embryos/foetuses in your project?

Yes No

9 * Does your project involve the use of **animal tissues** or **biological fluids**?

Yes No

(10) * Do you intend to only use (1) **established commercial animal cell lines**, or (2) **slaughterhouse-derived tissues/fluids**, or (3) **fluids collected as part of routine animal husbandry** (e.g. milk) and no other animal tissues or biological fluids in your project?

Yes No

11 * Does your project involve the **collection of rare or protected plants**?

Yes No

12 * Does your project involve the generation or use of **genetically modified (GM) plant material**?

Yes No

(13) * Do you agree to (1) only use **established genetically modified (GM) plant cell lines, seeds, or plant products** in your project, (2) **not generate new plant mutations** using chemical or other means, and (3) follow specified SETU **containment and use protocols** for GM plant materials at all times?

Yes No

14 * Does your project involve the use of **ionising radiation**? (e.g. use of gamma ray spectrometry)

Yes No

(15) * Do you agree to carefully **follow the instructions** of the SETU designated **Radiation Protection Officer (RPO)**, and **adhere to all legal requirements** as set out in the Radiological Protection Act 1991 (Ionising Radiation) Regulations (2019), regarding the use of ionising radiation materials and equipment?

Yes No

16 * Does your project involve the **collection of any new (or primary) data** from **individual people or groups**?

Yes No

(17) * Does your project involve the **collection of any new (or primary) individual or group data** that is **personally or uniquely identifying**? (e.g. data about people or organisations/companies/groups that could be used to identify those individuals or groups; data collection might take any form, including internet and social media data, etc.)

Yes No

(18) * Will you ensure that participants who you are collecting data from are provided with **fair warning** and must provide **explicit informed consent** for any data collected?

Yes No

(19) * Will you ensure that any project-related data collection, data storage, and data use is in **full compliance** with the **EU General Data Protection Regulation (GDPR)** and the **Data Protection Act (2018)**?

Yes No

(20) * Does any of the data that you intend to collect include **sensitive or private personal information** about individuals, or **commercially sensitive information** about organisations/companies/groups?

Yes No

21 * Does your project involve **persons under the age of 18 years** (i.e. minors), or **any vulnerable groups**? (e.g. prisoners, refugees, those in care, addiction service users, etc.)

Yes No

22 * Does your project involve the use of **existing (or secondary) data**? (i.e. data originally collected for another purpose)

Yes No

(23) * Is the existing or secondary data you intend to use either (1) **anonymous/non-personally identifying** and in the **public domain**, or (2) available with **explicit and specific informed consent or permission** for the data to be **legally** reused in the way you intend?

Yes No

(24) * Are any aspects of the primary/secondary data you intend to use for the project **controversial** in nature?

Yes No

25 * Before you submit the Ethics Checklist, you must **confirm all of the following**:

- I understand that the Ethics Checklist is a formal declaration.
- I have answered all questions on the Ethics Checklist carefully and truthfully.
- The supervisor/advisor (or principal investigator) for the project is present as the Ethics Checklist is being submitted, or they have given me explicit permission to submit it in their absence.
- I have had adequate ethics training and/or instruction prior to completing the Ethics Checklist.
- I understand, and agree to abide by, the general ethical principle of "do no harm" for this project.
- I will follow the instructions given in the Feedback Report.

26 * Authentication Code (ask your project supervisor/advisor for this code)

Enter Student Number:

Enter the Authentication Code below and click "Verify Code" Verify Code

Note: If an INVALID authentication code is used then this submission is NULL and VOID

7494

1 | 2 | Next

◄ Announcements Jump to... Ethics Checklist Feedback Questionnaire (***optional***) ►