SYSC3010 |
Computer Systems Development Project

FLORALYFE

Plant Monitoring System Detailed Design

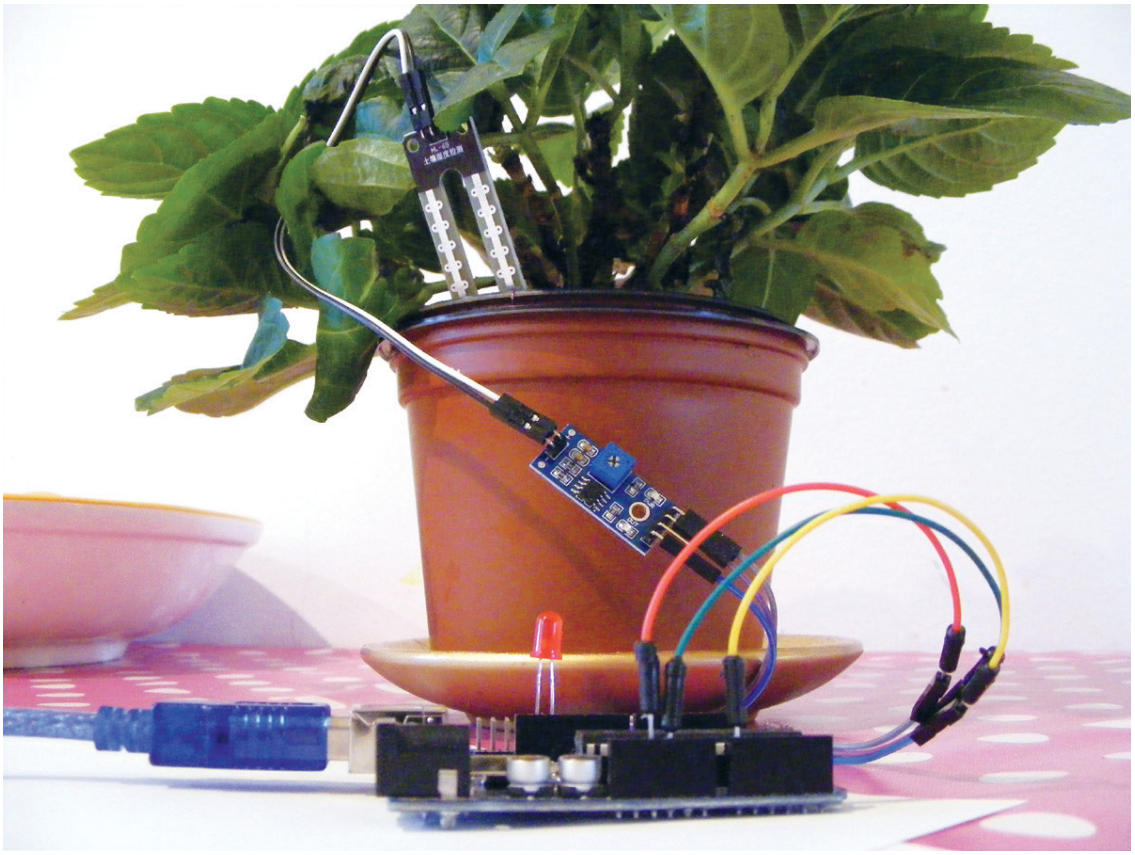


Figure 1. ***Mock*** *representation of physical Floralyfe system [1].*

Group L3W - G5

Abdalla Abdelhadi, 101142768
Zakariyya Almalki, 101152124
Yousef Yassin, 101143052

TA: Roger Selzler

March 13th, 2022

Table of Contents

1 Problem Statement

Houseplant demand has surged by 20% during the Covid-19 pandemic, with 66% of American households owning at least one houseplant [2]. The desire to become a plant parent has become particularly popular to the degree that sales in the U.S. have increased by 50 percent to $1.7 billion as a result [3]. Studies have shown that the average houseplant owner will kill around seven plants. It follows that the leading fear of owning a houseplant is the assurance that the plant receives sufficient water [2]. To broaden the plant parent demographic, *Floralyfe* aims to make the plant-growing hobby simpler and thus more attractive. The system intends to remove the attached upkeep of plant care. *Floralyfe* will ensure stress-free and healthy plant growth, allowing individuals to spend more time enjoying their plants rather than worrying about their health.

1.1 Functional Requirements

To facilitate the care of houseplants, *Floralyfe* will satisfy the following functional requirements:

Remote Plant Vital Monitoring: The system shall allow users to register up to 2 plants and periodically measure the temperature, humidity, and soil moisture for these plants every hour. These measurements will be visualized graphically on a web interface.

Automatic Irrigation: The system shall water a user's plants. When moisture is low, a selected plant shall be watered until soil moisture is restored to above the optimal value. Vital optima will be specified through user input and recommendations will be offered through a third-party API based on automatic recognition of the plant species.

Vital Notifications: The system shall notify users of their plants' health. When a plant's vital measurements drop below the set optimum, the plant's owner will be notified of the event detailing the critical vital(s). A corresponding icon will be displayed on the system's Sense Hat to communicate the notification, removing the need to check the user interface.

Plant Live Camera Feed: The system shall frequently send image frames of the selected plant to the frontend to create a live video feed. The front end will allow users to alternate between selected plants, upon which the camera will rotate to face the new plant. Users shall also be able to rotate the camera yaw from the client, allowing them to observe their plants' environment.

Plant-specific Notes: The system shall allow users to track the growth progress of their plants. Users shall be able to create note posts for each of their plants to list important information, akin to a plant diary, to track plant growth.

2 Design Overview

2.1 System Overview Diagram

Floralyfe is structured as a real-time distributed system. A detailed structural view of the system is illustrated by the deployment diagram in Figure 2.1.1. below. As conveyed by the network of clusters in the figure, the project shall consist of several deployed nodes, each with a specific task. These nodes will intercommunicate over the internet. Namely, the design consists of three *Plant Monitoring Stations* that are

depicted in Figure 2.1.2. below. Each station is centered around a Raspberry Pi (RPi) device running in headless mode. The stations will host a local database to store identifying information about their registered user to tag messages and allow multiplexed broadcasting. The database will also store a daily image of each registered plant to form a growth slideshow. The RPi stations additionally rely on three interconnected subsystems to achieve more complex functionality: the Vital Monitoring Subsystem, the Automatic Irrigation Subsystem, and the Camera Monitoring Subsystem.

Further detail regarding the roles of these three core subsystems, among others, is provided in various sections below. In short, the vital monitoring system is responsible for sensor data collection, some of which is processed by the automatic irrigation system to determine when to water a plant. Along with these two systems, the camera monitoring system periodically captures images of a selected plant. All of this information — the sensor data, watering events, and camera images — are sent to a remotely deployed frontend client interface to be visually displayed to the user.

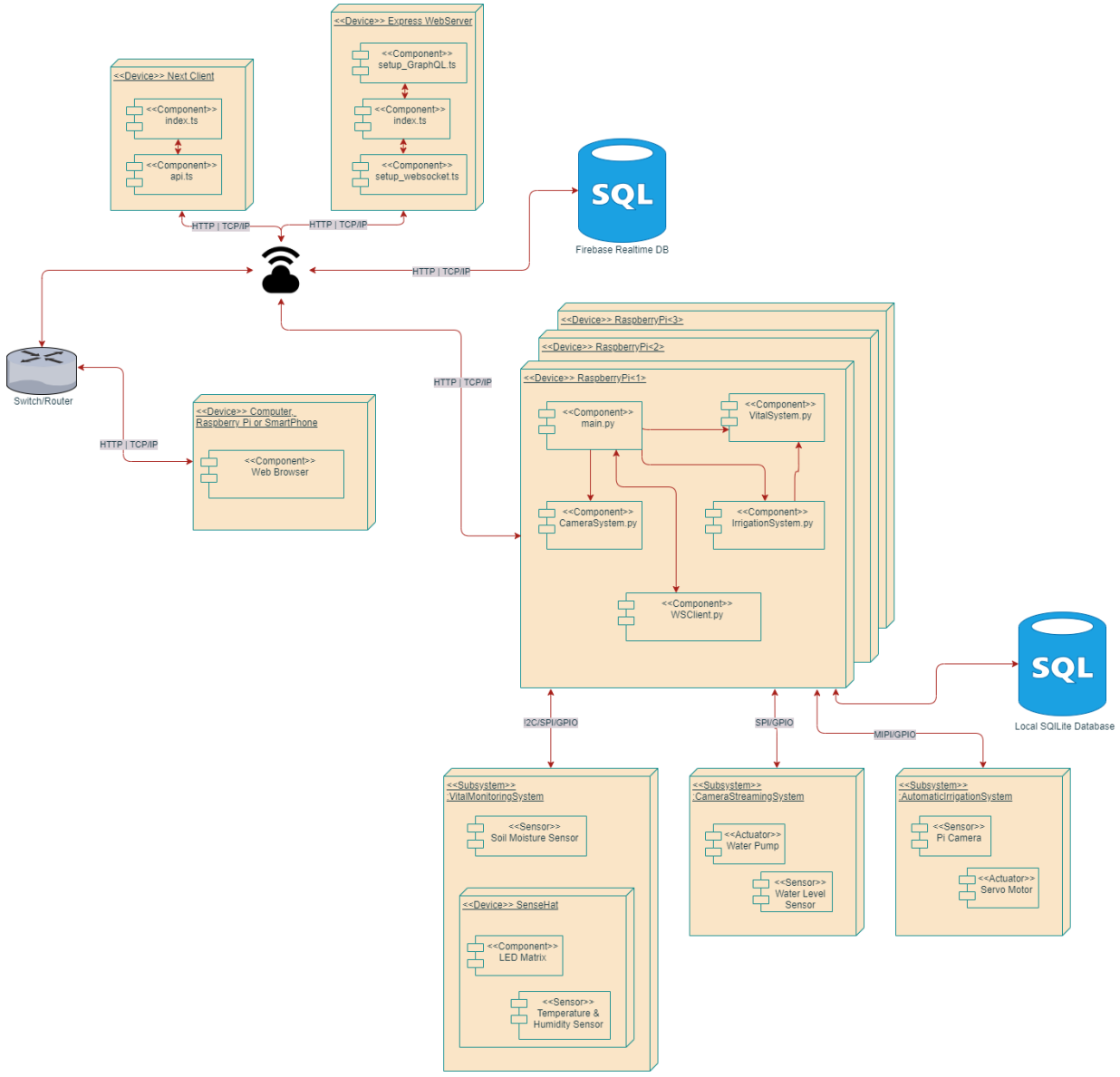


Figure 2.1.1 Deployment diagram

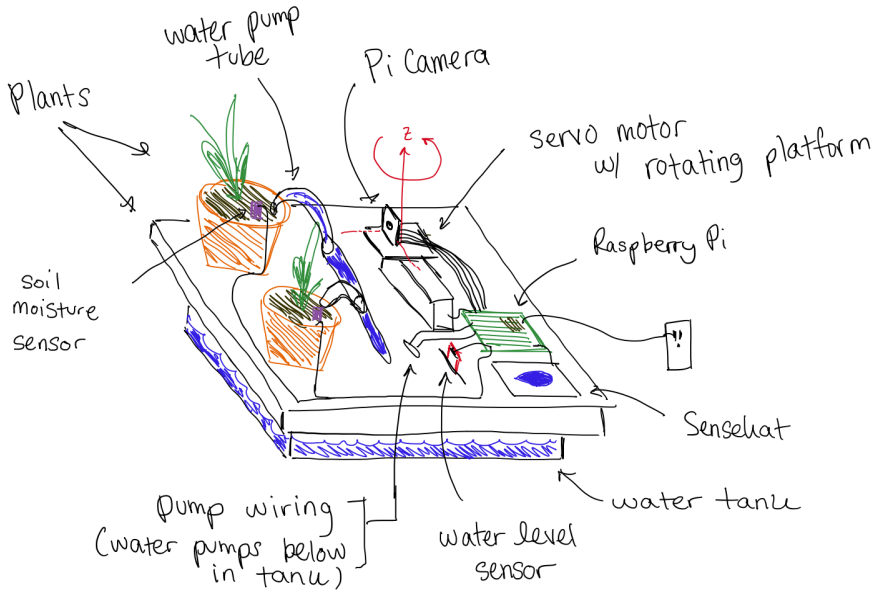


Figure 2.1.2. Structural Representation of Floralyte Plant Monitoring Station

**Wiring in Figure 2.1.2. is simplified for brevity. In reality, the wiring will go through a breadboard circuit to support organization and to route sensor data through an analog to digital conversion (ADC) circuit. Refer to 2.2 Component Details for detailed circuit diagrams.*

2.2 Communication Protocols

Two communication contexts exist for the Floralyfe system; communication between hardware components at the physical level and between the RPi system and cloud clients at the software level. Both contexts differ in their use of communication protocols.

At the physical level, the Floralyfe system collects data from several sensors. The Sense Hat module reads the room temperature and humidity. The acquired data is communicated to the Raspberry Pi using the I2C Serial Protocol. The remaining set of analog sensors— the soil moisture sensor and the water level sensor —are each connected to a channel on an MCP3008 chip to convert the sensors' analog output into a digital one. The chip then communicates with the Raspberry Pi using the SPI Serial Protocol. The actuators in the system— a servo motor and DC water pump —both have DC inputs and are thus activated using the Raspberry Pi's GPIO interface. To accurately control the servo motor's heading, a Pulse Width Modulated (PWM) signal will be used. The final physical component is the PiCamera which periodically captures images of plants and sends them back to the frontend in real-time. The camera interfaces with the Raspberry Pi using the MIPI Camera Serial Interface Protocol [4].

While the physical system is central to the Floralyfe system - the frontend interface is equally significant in processing and communicating the acquired data to our end users. In some instances, the frontend will query third-party REST APIs using the HTTP Protocol to acquire plant recognition information

synchronously. In all other cases, data must be communicated from the physical nodes to be rendered on the web client. This communication is achieved using two internal backend servers: a Node-Express Server with a GraphQL API and a dedicated WebSocket server.

The GraphQL server is responsible for wrapping our Firestore database. Wrapping helps to expose a common API to both the frontend client and the physical RPi system. To communicate with Firestore, the server will use synchronous HTTP (POST, GET) requests. Clients communicating with the server API can do so with three actions; queries, mutations, and subscriptions. Queries retrieve data while mutations update data. Both these actions will operate using the HTTP Protocol. Subscriptions subscribe to a specific collection and allow the subscriber to be notified of the collection's mutations in real-time. Due to the real-time, asynchronous nature of this communication, WebSockets are used operating under the TCP/IP protocol.

2.2.1 Communication Protocol Tables

Four distinct messaging edges can be identified in the Floralyfe communication graph. Both the frontend client and physical Raspberry Pi systems communicate with the backend GraphQL server to interface with the Firestore database. These queries use the GraphQL query language, but the queries themselves are embedded into a JSON in the HTTP requests. In addition to the GraphQL API, the backend server also exposes a minimal amount of REST routes to interface with third-party APIs such as Nodemailer to send email notifications.

The client and physical systems also communicate directly through the WebSocket server. The socket is used by the Camera Monitoring Subsystem to send images to the client and to accept messages to turn the servo motor supporting the camera. The Vital Monitoring Subsystem uses the socket to push live plant vitals of a selected plant to the client (which are not persisted in the database) along with accepting messages to change the informative icon on the Sense Hat. The Irrigation Subsystem uses the socket to listen to messages indicating a manual plant watering request.

2.2.1.1 GraphQL Server

The GraphQL Server wraps the Firestore database. As such, the server's primary responsibility is to manage and publish create, read, update and delete (CRUD) operations. The GraphQL querying language achieves reading operations through *queries*, while all other operations are handled by *mutations*. Subscriptions are also handled by the GraphQL API to publish real-time notifications for every mutation. These messages are multiplexed through node ids and can be bound to clients, thus allowing real-time data synchronization between the client and physical system nodes. The messages handled by the GraphQL server are summarized in Table 2.2.1.1. below. Note that GraphQL specific types have been indicated for formatting and that each query will be wrapped by a JSON. Specifically, each query consists of an HTTP POST request to the server with the query embedded under a "query" JSON field.

Table 2.2.1.1 *Communication table for backend GraphQL Server*
** An exclamation point (!) indicates a mandatory field.*

Sender	Receiver	Message	Data Format
Frontend	GraphQLServer	*getVitals*	`query {`

			<pre>vitals(plantID: ID!) {
			id: ID!
			soilMoisture: Float! [%]
			Temperature: Float! [°C]
			airHumidity: Float! [%]
			light: Float! [%]
			greenGrowth: Float!
			[unitless]
			plantID: ID!
			createdAt: ISODate!
			}
			}</pre>

RPiDevice	GraphQLServer	*createVital*	<pre>mutation {
			createVital(
			soilMoisture: Float! [%]
			temperature: Float! [°C]
			airHumidity: Float! [%]
			light: Float! [%]
			greenGrowth: Float!
			[unitless]
			plantID: ID!
			){
			id
			}
			}</pre>

Frontend	GraphQLServer	*subscribeVitals*	<pre>subscribe {
			vitals(deviceID: ID!) {
			mutation: String!
			data: {
			id: ID!
			soilMoisture: Float! [%]
			temperature: Float! [°C]
			airHumidity: Float! [%]
			light: Float! [%]
			greenGrowth: Float!
			[unitless]
			plantID: ID!
			createdAt: ISODate!
			}
			}
			}</pre>

Frontend / RPiDevice	GraphQLServer	*getUser*	`query { user(userID: ID!) { id: ID! firstName: String! lastName: String! username: String! email: String! avatar: String! deviceID: ID } }`
Frontend	GraphQLServer	*createUser*	`mutation { createUser( firstName: String! lastName: String! username: String! email: String! password: String! avatar: String! subscribedNotifications: Boolean! ){ id } }`
Frontend	GraphQLServer	*updateUser*	`mutation { updateUser( userID: ID! firstName: String lastName: String username: String email: String password: String avatar: String subscribedNotifications: Boolean deviceID: ID ){ id } }`
Frontend	GraphQLServer	*getNotifications*	`query { notifications(userID: ID!) { id: ID!`

			Label: String!
			type: *NotificationType!
			plantID: ID!
			createdAt: ISODate!
			}
			}
			
			enum NotificationType {
			VITAL, REFILLED, LOW_WATER
			}

RPiDevice	GraphQLServer	*createNotification*	mutation {
			createNotification(
			label: String!
			type: NotificationType!
			){
			id
			}
			}

Frontend	GraphQLServer	*Subscribe Notification*	subscribe {
			notification(deviceID: ID!)
			{
			mutation: String!
			data: {
			id: String!
			label: String!
			type: NotificationType!
			createdAt: ISODate!
			}
			}
			}

Frontend	GraphQLServer	*getNotes*	query{
			notes(
			userID: ID!
			){
			id: ID!
			title: String!
			text: String!
			plantID: ID!
			updatedAt: ISODate!
			}
			}

Frontend	GraphQLServer	*createNote*	`mutation { createNote( title: String! text: String! plantID: ID! ){ id } }`
Frontend	GraphQLServer	*subscribeNotes*	`subscribe { note(userID: ID!){ mutation: String! data: { id: ID! title: String! text: String! plantID: ID! updatedAt: ISODate! } } }`

2.2.1.2 WebSocket Server

The WebSocket Server handles real-time communication between nodes that do not need to be persisted in the Firestore database. The server accepts two types of JSON messages that enable client subscriptions and serves as a middleman to route the remaining messages between subscribed clients. To scale the *Floralyfe* system, all inter-node communication is multiplexed by the sender's id and a topic such that multiple clients can be supported. At the WebSocket level, nodes must subscribe to the clients they'd like to receive messages from. The routed message then consists of a topic string and a payload, containing the transmitted data. The topic string allows interpreting the data on the receiver's end. Table 2.2.1.2. below highlights the "subscribe" and "unsubscribe" message format for the WebSocket server. Note that TypeScript data types have been used to indicate formatting below.

Table 2.2.1.2. *Communication table for WebSocket Server*

Sender	**Receiver**	**Message**	**Data Format**
Frontend / RPiDevice	WebSocketServer	*subscribe*	`JSON { topic: "subscribe", payload: { subscriptionID: string } }`
Frontend /	WebSocketServer	*unsubscribe*	`JSON {`

RPiDevice			`topic: “unsubscribe”,`
			`payload: {`
			`subscriptionID: string`
			`}`
			`}`

2.2.1.3 Camera Monitoring Subsystem

The Camera Monitoring Subsystem’s communication responsibilities consist of sending base64 encoded image frames to the client and accepting commands to turn the camera. The system also allows users to register a plant’s heading relative to the system from the web interface. Table 2.2.1.3. below displays the messages relating to these operations, all of which are routed through the WebSocket Server and are tagged with the “camera-topic” topic.

Table 2.2.1.3. *Communication table for Camera Monitoring Subsystem*

Sender	Receiver	Message	Data Format

Frontend	RPiDevice::CameraMonitor	*turnCamera*	`JSON {`
			`topic: “camera-topic”,`
			`id: string,`
			`payload: {`
			`yaw: number [radians]`
			`}`
			`}`

Frontend	RPiDevice::CameraMonitor	*registerPlant*	`JSON {`
			`topic: “camera-topic”,`
			`id: string,`
			`payload: {`
			`yaw: number [radians],`
			`plantID: string`
			`}`
			`}`

RPiDevice::CameraMonitor	Frontend	*sendFrame*	`JSON {`
			`topic: “camera-topic”,`
			`id: string,`
			`payload: {`
			`encodedFrame: string{base64}`
			`}`
			`}`

2.2.1.4 Vital Monitoring Subsystem

The Vital Monitoring Subsystem's communication consists of accepting messages to change the informative icon displayed on the SenseHat and sending messages containing real-time plant vital data. These vitals are sent periodically every five seconds for a plant being inspected and are thus not persisted in the database like the ones taken every hour. The system also publishes persisted vitals to the GraphQL server every hour as documented in Table 2.2.1.1. Table 2.2.1.4. summarizes the messages related to the real-time operations, which are routed through the WebSocket and tagged with the "vitals-topic" topic.

Table 2.2.1.4. *Communication table for Vital Monitoring Subsystem*

Sender	Receiver	Message	Data Format
Frontend	RPiDevice ::VitalsMonitor	*setSenseIcon*	`JSON { topic: "vitals-topic", id: string, payload: { icon: *SenseHatIconType } } enum SenseHatIconType { CLEAR, MOISTURE, SUN, WATER_TANK }`
RPiDevice ::VitalsMonitor	Frontend	*sendVital*	`JSON { topic: "vitals-topic", id: string, payload: { vital: { soilMoisture: float [%], temperature: float [°C], airHumidity: float [%], light: float [%], greenGrowth: float [relative/unitless], plantID: string, createdAt: ISODate } } }`

2.2.1.5 Irrigation Subsystem Communication Table

The Irrigation Subsystem does not need to send direct messages to any other node except for handling notifications of watering events. These notifications are handled by the GraphQL server subscriptions and third-party nodemailer API for sending emails. The Subsystem does still need to handle incoming messages to override the internal control system and turn on the watering pump for a specified timeout in seconds. Table 2.2.1.5. highlights the format of this watering message.

Table 2.2.1.5. *Communication table for Irrigation Subsystem*

Sender	Receiver	Message	Data Format

Frontend	RPiDevice ::IrrigationSubsystem	*waterPlant*	JSON {
			topic: “irrigation-topic”,
			id: string,
			payload: {
			wateringTimeout: number [seconds]
			}
			}

2.3 Message Sequence Diagrams

To implement the functional requirements described in *Section 1.1: Functional Requirements*, the *Floralyfe* system will satisfy six core use cases highlighted and illustrated in the message sequence diagrams below.

2.3.1 Message Sequence Diagram 1: RegisterSystem UseCase

The RegisterSystem use case is the first use case that a user encounters when using the Floralyfe system. Figure 2.3.1 below depicts a message sequence diagram of the RegisterSystem use case. The use case allows a user to register their account along with their physical device. To register their account, a user fills a form on the frontend with their identifying information such as a username, password, and email. This information is sent to the GraphQL server to register the user in Firestore. Afterward, the user specifies their login information to their physical RPi device. The RPi uses the information to query the GraphQL server, register the device and associate it with the user’s account. At the end of this transaction, the frontend can query the server to obtain the user’s device identifier (id). Likewise, the device already has the user's id through the login credentials. The device subscribes to all WebSocket messages originating from the user’s client session and the client subscribes to all messages coming from the device by specifying the corresponding ids, hence inter-node communication is established. All the message sequence diagrams below assume the RegisterSystem UseCase has been satisfied.

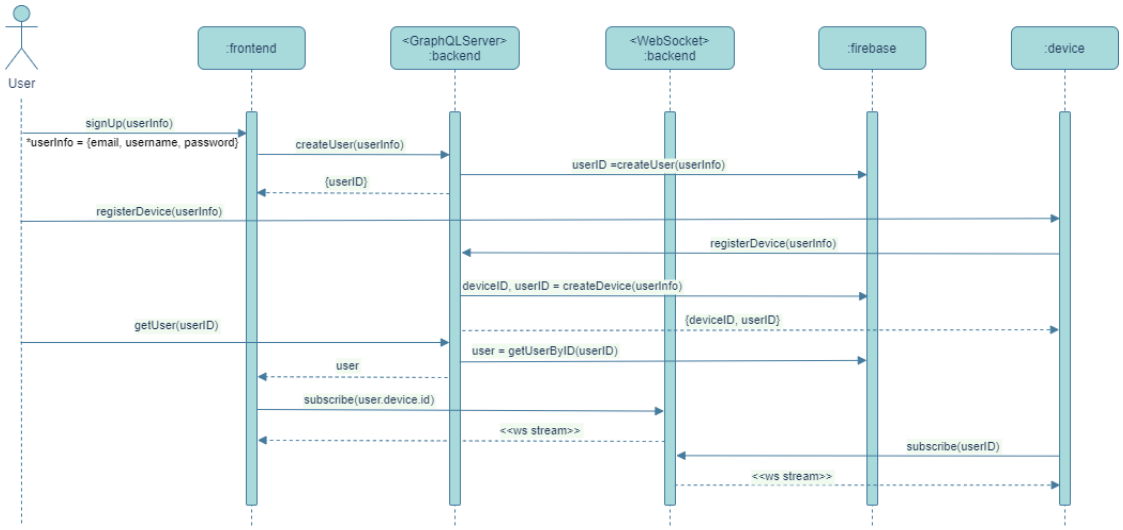


Figure 2.3.1. Sequence diagram for use case 1: RegisterSystem

2.3.2 Message Sequence Diagram 2: RegisterPlant UseCase

The RegisterPlant use case allows users to register a maximum of two plants to their account. The use case is depicted in Figure 2.3.2 below with the registration of a single plant. To initiate the use case, the user selects the “register” option on the frontend which triggers an image stream to begin from the RPi. The stream also includes all the sensor information visible on all pin channels. With the according options on the frontend interface, the user is then able to turn the RPi camera to face the plant they’d like to register. Once satisfied with the camera heading, the user confirms their selection. The current camera frame is then sent to the PlantId third-party API to identify a list of the most likely plant species. The most likely species is used to query an external plant database (WUCOLS IV) to obtain information regarding optimal watering (if available). Both pieces of information— the species list and optimal watering —are sent back to the client. The user is prompted to enter the species and optimal living environment with the option of using the predicted values or selecting their own. The user is also told to enter a name for their plant along with the correct sensor channel connected to the plant. Once confirmed, the information is sent to both the GraphQL Server and RPi system to register the plant on the cloud and locally on the physical system. The same sequence applies for re-registering or updating a plant's information.

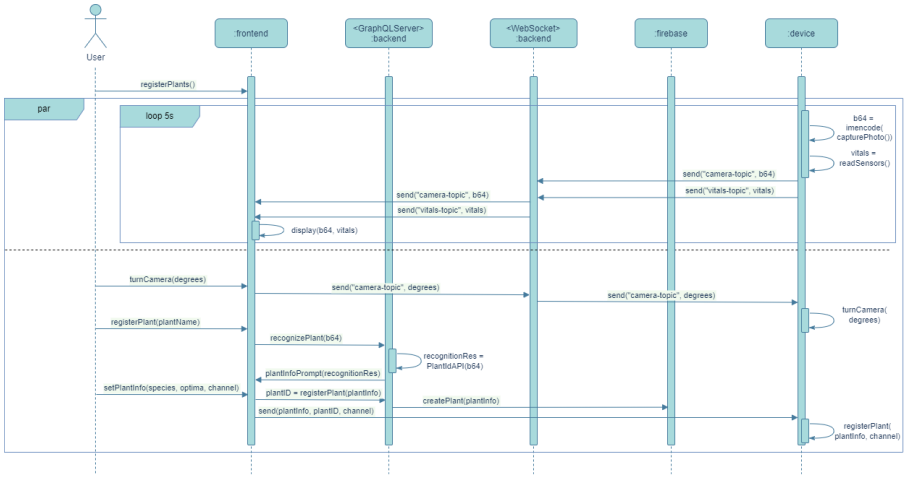


Figure 2.3.2. Sequence diagram for use case 2: RegisterPlant

2.3.3 Message Sequence Diagram 3: InspectPlantStream UseCase

The InspectPlantStream use case allows users to select a plant through the frontend web interface and to view a live stream of its data. The data stream includes a live image stream and real-time vital sensor values both as numerical measures and graphed plots. Figure 2.3.3 below depicts the message sequence diagram of the InspectPlantStream use case. Once a plant is selected, the RPi system is notified and turns to face the selected plant. Two separate threads are activated; one that captures and sends an encoded image every second to create the live image stream. The second reads various sensor values - soil moisture, temperature, humidity - and sends them to the client every five seconds.

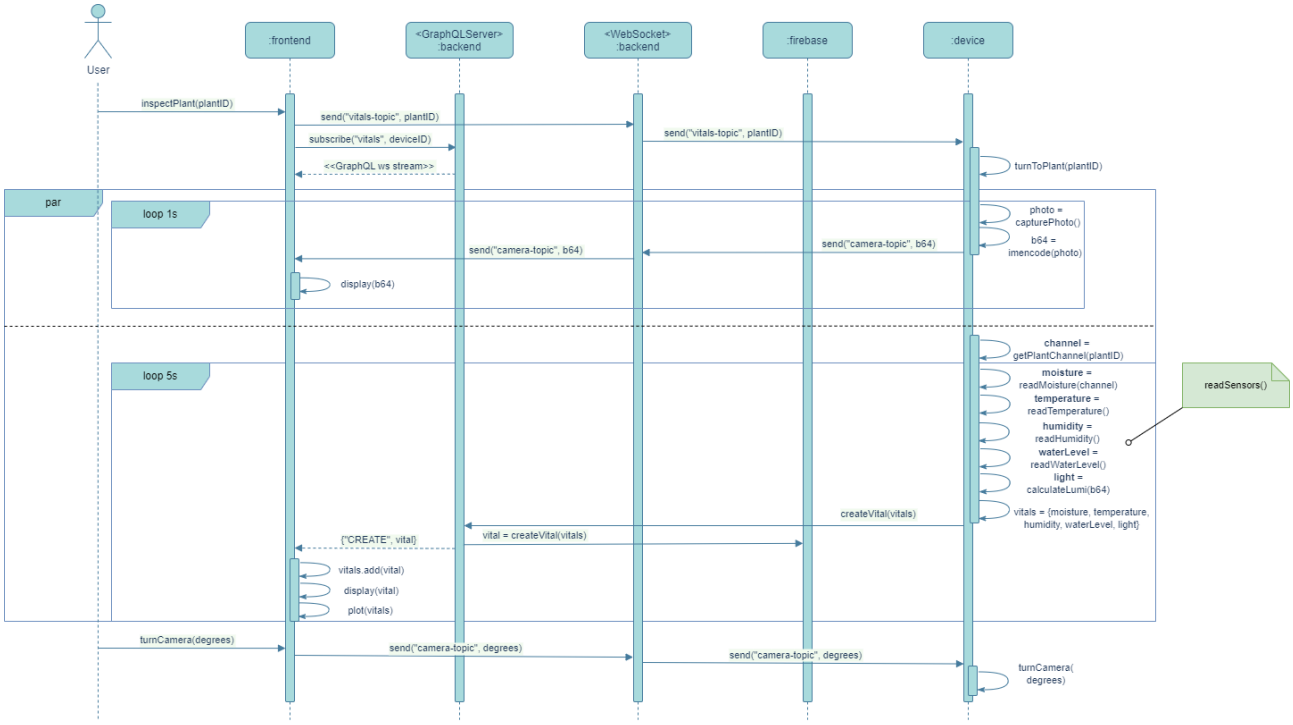


Figure 2.3.3. Sequence diagram for use case 3: InspectPlantStream

2.3.4 Message Sequence Diagram 4: NotifyPlantOwner UseCase

The NotifyPlantOwner use case involves running a check on every measured vital and ensuring that it is above the user's specified optimal values for each selected plant. Figure 2.3.4 depicts the sequence diagram of the NotifyPlantOwner use case. When the Vital Monitoring Subsystem performs its periodic sensor readings, they will be compared on the backend to the stored optima for each registered plant. In the event that a measured value falls below the specified optimum, a notification will be created and stored in the database for future reference. The same notification is sent to the third-party nodemailer API, along with the plant owner's email to send an email notification to the user informing them of the event.

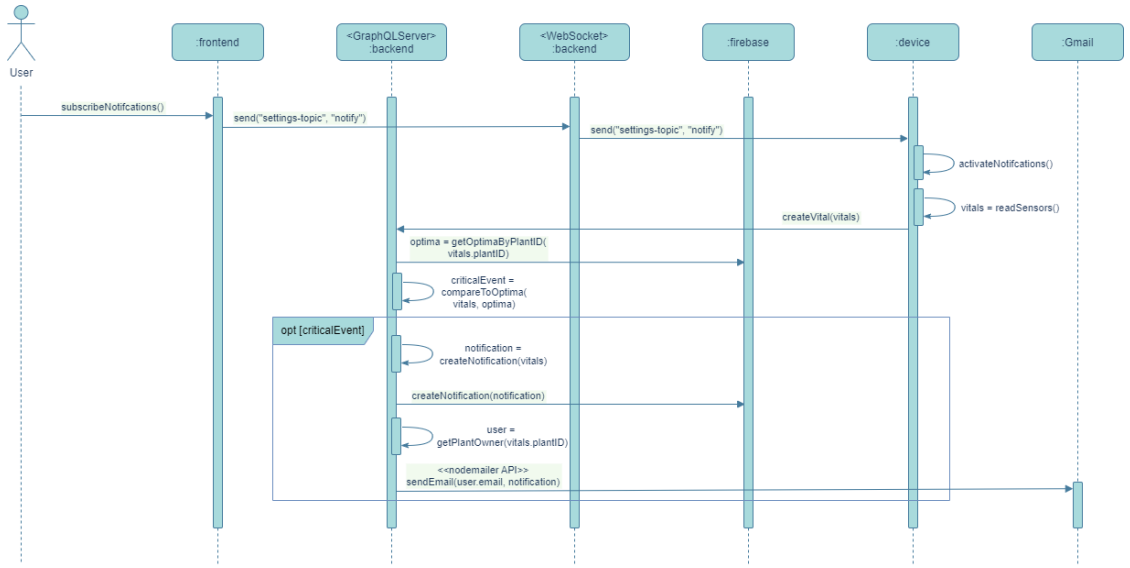


Figure 2.3.4. Sequence diagram for use case 4: NotifyPlantOwner

2.3.5 Message Sequence Diagram 5: IrrigatePlant UseCase

The IrrigatePlant use case will periodically measure the soil moisture levels of each registered plant every hour through the moisture sensor (through the usage of the Vital Monitoring System's interface). Figure 2.3.5 below depicts a sequence diagram of the IrrigatePlant use case. In the event of a critical event, if gathered data indicates dry soil conditions, then the plant irrigation system will be activated. The system will water the soil for 5 seconds then sleep for two minutes. The soil moisture level will then be checked again, if it is still below the optimal value, then the watering process will be repeated. Otherwise, the system will sleep until the next hour cycle. Water tank levels and their associated notifications are already handled by the NotifyPlantOwner use case.

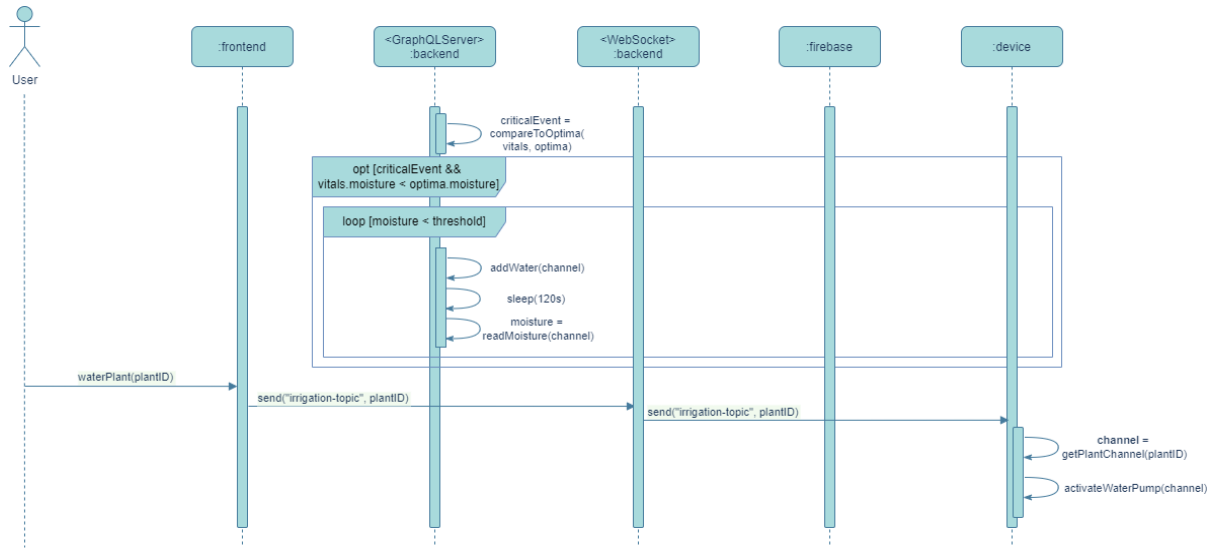


Figure 2.3.5. Sequence diagram for use case 5: IrrigatePlant

2.3.6 Message Sequence Diagram 6: CreatePlantNote UseCase

The IrrigatePlant use case allows users to create notes tied to their registered plants through the web interface. Figure 2.3.6 below depicts a sequence diagram of the CreatePlantNote use case. Note creation consists of sending a mutation to the GraphQL server containing the note information. The mutation creates and persists the note in the Firestore database. Upon logging in, all a user's notes are retrieved by the frontend with a GraphQL query. The interface will use an optimistic UI in automatically updating the client's note state under the assumption that GraphQL queries are successful. The stored notes are then displayed near associated plants on the frontend dashboard, allowing users to create a plant diary of sorts.

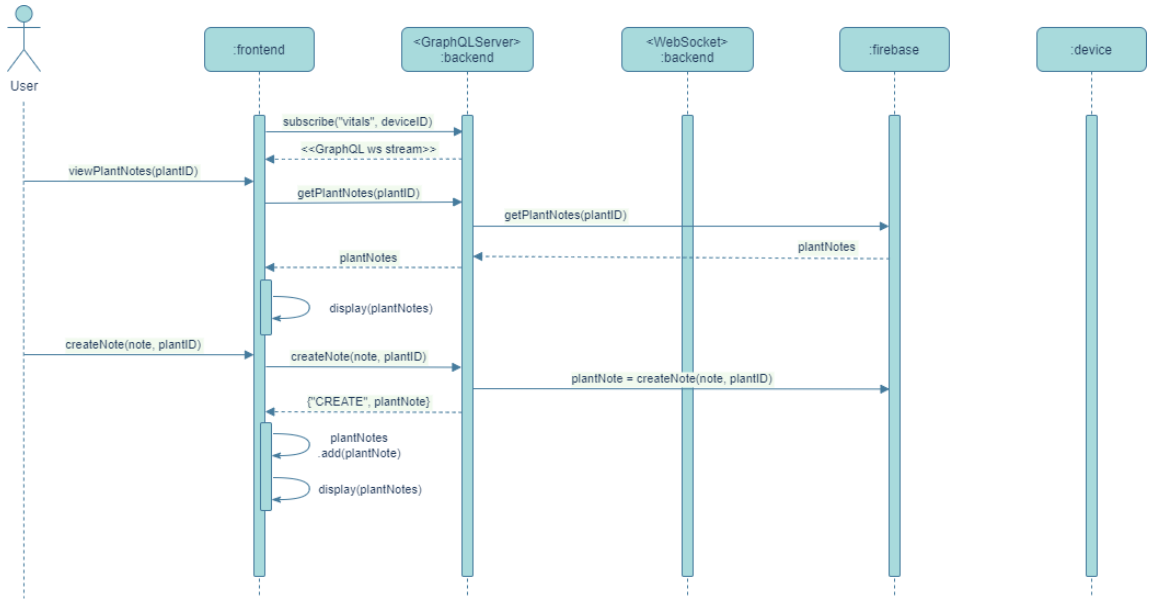


Figure 2.3.6. Sequence diagram for use case 6: CreatePlantNote

2.4 Database Table Design/Schema

Floralyfe will rely on two databases; a cloud Firestore database and a local SQLite database. Firestore will store five relational tables that link a user to their system device and their plants. Plants are further associated with sensor measurements (vitals), notifications, and notes. While Firestore is a NoSQL database, *Floralyfe* will treat the database as if it were a SQL one. As a system still in development, NoSQL offers more fluidity in changing schemas in the future whereas SQL is more strict and rigid. All data collections will consist of a single entity with a well-defined schema and JSON nesting will not be used. This treatment enhances the maintainability of the system and facilitates the creation of simpler queries. The local SQLite database on each system will store unique user identifying information, device state-related information, and a daily photo taken of each of a user's plants to create a local growth time lapse. The schemas for both databases have been merged in Figure 2.4 below.

Each database entry is identified by a universally unique identifier to support the scalability of the system and associations between entries. All database fields are unitless except those found in the Vital collection. The units for soilMoisture, airHumidity and light are all percentages. Temperature is measured in celsius and greenGrowth is unitless but used as a relative value (more green results in a larger value).

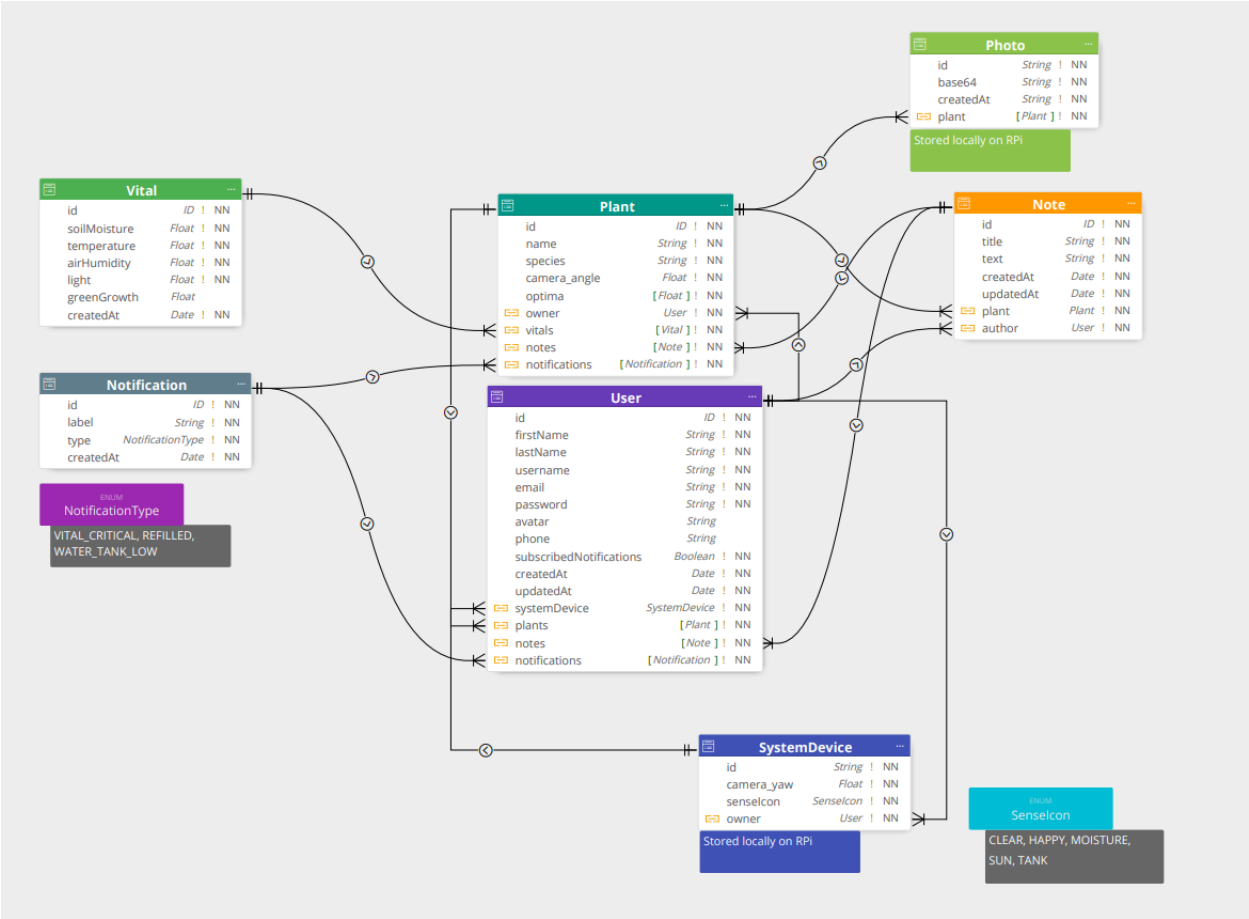


Figure 2.4. Database schema diagram

3 Software Design

The *Floralyfe* system will be implemented using the Python Programming Language at the physical Raspberry Pi nodes, and TypeScript for all cloud and client-side applications. The majority of Python modules will be developed using an object-oriented approach while some helper and utility functions will be defined as standalone functions. The backend server will be developed using TypeScript in a Node JS environment and using the Express Framework. The server will be written using a functional paradigm. Likewise, the frontend client will also be developed using TypeScript in a Node JS environment but using the Next JS framework, a derivative of the React framework.

Disclaimer: various periodic schedules are described below in the order of hours or days. In development and demo contexts, these periods will be significantly reduced (max ~10 minutes) to accurately test the system.

3.1 Software Design for Raspberry Pi Physical Node

Each FloraNode will have a task queue to receive WebSocket messages from the client, a name for registration and logging purposes, and a scheduler to schedule periodic tasks. Moreover, each node will have a shared reference to the currently selected plant channel which is updated externally. Each

FloraNode also has access to a database connection reference to make queries or publish data to the local database along with a WebSocket Client reference to send real-time messages to the client. Helper methods will be exposed using the *requests* library to interface with the GraphQL server. The overall operation of a FloraNode consists of two operating threads: a main and worker thread. The main thread is periodically scheduled to accomplish the core tasks of the subsystem - those that always occur and are not triggered by an asynchronous message. The worker thread is responsible for watching the node's task queue and blocks until a message is received. Upon message receival, the worker parses the message and executes the appropriate action to respond to the message. The source of all messages is the WebSocket Client.

The *Floralyfe* system has two WebSocket Clients, one at the RPi level and another at the frontend client level. Both operate in a similar manner which is detailed in *Section 3.4: Software Design for WebSocket Client Below.* In short, the RPi WebSocket client connects to the backend WebSocket and subscribes to messages coming from the user's frontend client (tagged with the user's id). These messages are then routed to the appropriate subsystem according to their topic field.

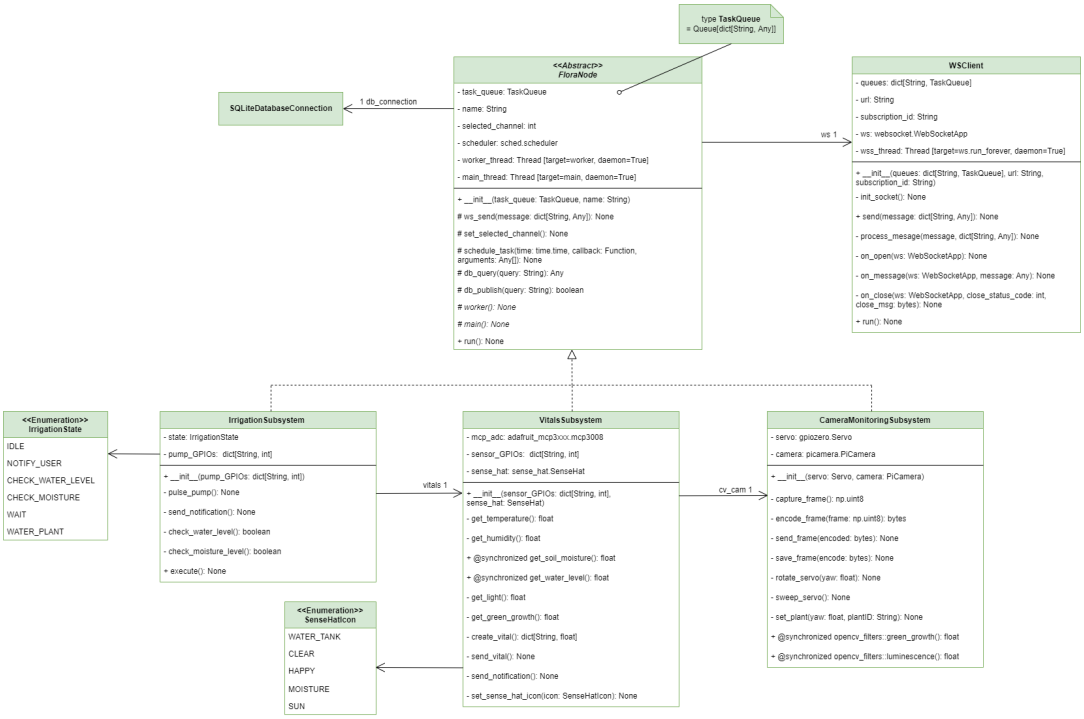


Figure 3.1.1. UML Class Diagram of RPi Physical Node

3.1.1 Software Design for Irrigation Subsystem

The Irrigation subsystem monitors and moderates the soil moisture of a user's plants. The subsystem will be implemented through a state machine that is illustrated in Figure 3.1.1 below. The subsystem starts in an idle state where it simply waits for the next monitor cycle. Every hour, the state is scheduled to change to check the moisture. The subsystem reads the soil moisture level by invoking an exposed method from the Vital Monitor Subsystem that interfaces the soil moisture sensor. If the moisture level is above the optimal threshold for the currently selected plant, the state switches back to idle and the cycle repeats. However, if the moisture is below the threshold then the subsystem will attempt to water the plant.

First, the Irrigation system ensures that there is enough water in the water tank to water the plant again by interfacing with the appropriate method in the Vital Monitor system. If the measured water level indicates that there isn't enough water to irrigate the plant then the subsystem switches states to notify the user by sending them a notification to fill up the tank. A notification email is sent from the backend using the Nodemailer API. The Irrigation Subsystem simply requests the appropriate backend route to trigger the API. After sending the notification, the system returns to idle - if the water tank is not filled by the next hour cycle, another notification is sent (and so on, until it is filled).

If there is sufficient water in the tank to water the plant, the system switches states to water the plant. Once the state changes, the appropriate channel's water pump is activated to pump water for five seconds then turns back off. The state then changes to wait for the water to disperse through the soil for two minutes. Once the wait is over, the subsystem needs to ensure that the water supplied was enough to sufficiently moisturize the soil so the state is switched to check the moisture level again. Once it switches, this process keeps repeating until the moisture level reaches an optimal level. The system will also respond to client messages by directly transitioning to the water level checking state to attempt watering the plant.

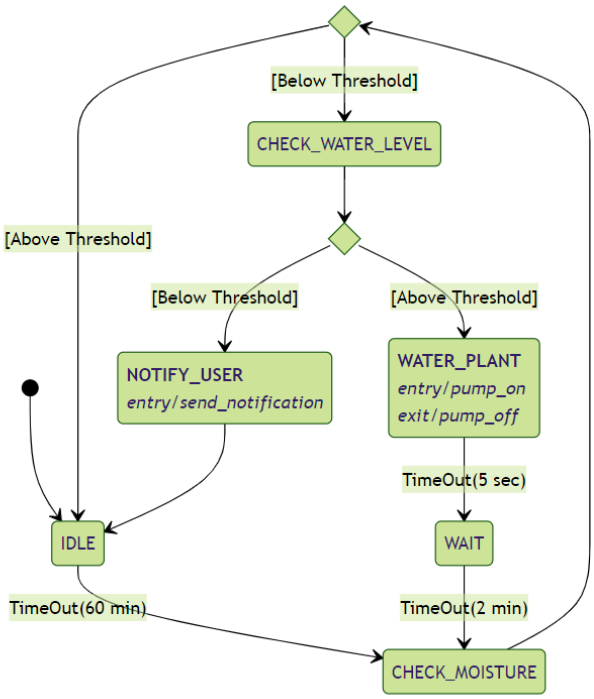


Figure 3.1.1. UML State Machine Diagram of Irrigation Subsystem

3.1.2 Software Design for Camera Monitoring Subsystem

The camera system allows users to inspect the environment of their plants, analyze these images and save them locally to create a growth time lapse. The camera monitoring subsystem will also implement a state machine that is displayed in Figure 3.1.2 below. The system's state is initialized as idle. While idle, the system can be triggered in two ways. It is listening to the WebSocket client for a command coming from the frontend and will respond to these asynchronous messages as they arrive. The system will also be scheduled to take a photo of each registered plant in the system every 24 hours. The picture is stored in

the local database. These can be retrieved from the physical device later by the user to create a growth time lapse.

When the camera monitoring subsystem receives a WebSocket message originating from the frontend, the message will either indicate a control message to turn the camera to a certain heading or request a live stream of a newly selected plant. If the user chooses to view a certain plant, the system queries the local database with the specified plant ID to obtain the heading of the selected plant. A control pulse is then sent to the servo motor to face this plant and the camera stream is initiated. If a camera rotation command is provided, then the desired heading is simply routed to the servo.

The system achieves live streaming through a combination of libraries. Namely Numpy, OpenCV, and Pybase. The system captures a BGR color-coded image in an empty NumPy uint8 array such that the image is compatible with the OpenCV interface. The image frame is then compressed using a JPEG scheme and encoded into a base64 byte string. The byte string is wrapped in a JSON payload, along with the appropriate multiplexing information, and sent to the frontend. This endeavor is repeated every second while the live stream is requested and occupies a dedicated thread in the Camera Monitoring Subsystem.

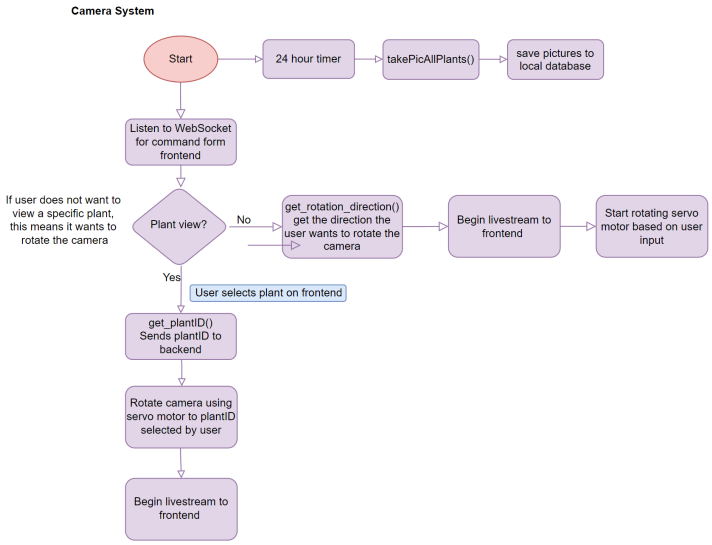


Figure 3.1.2. UML State Machine Diagram of Camera Monitoring Subsystem

3.1.3 Software Design for Vital Monitoring Subsystem

The Vital Monitoring Subsystem allows users to measure the properties of their registered plants' environment. Like the other two subsystems, the Vital Monitoring Subsystem will also implement a state machine and be initialized at an idle state. The complete state machine is described in Figure 3.1.3 below. Every hour, the system will be scheduled to measure the values of all the sensors on the selected plant's channel. This measurement consists of reading the room temperature and humidity from the SenseHat, the soil moisture from a soil moisture sensor, and the tank water level from the water level sensor. An estimated light measurement will be calculated using an OpenCV-based filter that measures image luminescence from a CIELAB colour encoded channel. Moreover, the application of a green mask and the sum of the resulting pixels' values will be used to estimate plant growth.

The measurements will be aggregated into a “vital” message and select parameters, namely soil moisture, will be compared to the optimal thresholds designated by the plant owner. In the event that a measured quantity lies below its optimal threshold, the subsystem will make a POST request to the backend server with the critical information. The information will be embedded in an email and sent to notify the user through the Nodemailer API. The created notification is also published to the Firestore database, through a GraphQL mutation, for future reference from the web interface.

In the case of optimal vitals, or following the sending of notification, the system publishes the measured vital to the Firestore database. Subscribed clients will be notified of the new vital and updated accordingly. Before returning to the idle state, the Vital Subsystem updates the Sense Hat informative icon based on the most recent vital. Note that the frontend may also override this icon through a message. It is also worth noting that clients requesting a stream of live plant vitals will simply result in a repeated measurement of sensor values that are sent through the WebSocket; none of the streamed vitals are validated or persisted in the database.

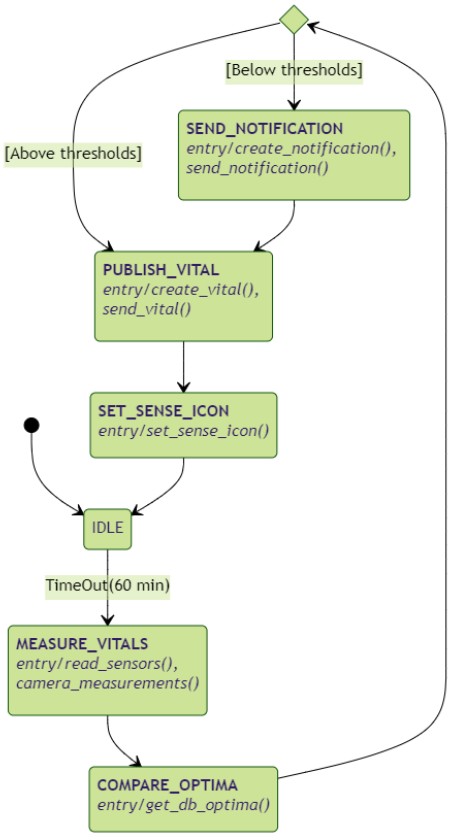


Figure 3.1.3. UML State Machine Diagram of Vital Monitoring Subsystem

3.2 Software Design for Backend Server

The backend server is the first among two cloud components in the Floralyfe system, the second being the frontend client. The server is written using the TypeScript language in a Node JS environment using the Express Framework. The server’s root task is to facilitate inter-node communication within the Floralyfe ecosystem. To do so, it has been subdivided into two smaller dedicated servers: a GraphQL server that wraps the Firestore database and a WebSocket server that routes messages from physical nodes to frontend clients and back. The GraphQL server also abstracts a data-subscription system that is built on top of WebSockets. Both servers are initialized from the Express server to operate on two distinct ports which are exposed on two separate public domains.

3.2.1 Software Design for GraphQL Server

The GraphQL server is directly connected to the Firestore database through the Firebase API. The server wraps the database and exposes a custom GraphQL API that is built using the Apollo package. The Apollo library abstracts the majority of the server's implementation. Apollo parses and handles the routing of GraphQL queries to their respective handler methods on the backend, hence all that remains is defining these queries. A list of all supported queries is supplied in Table 2.2.1.1. All queries are provided to the server as POST requests with JSON messages containing the query in a "query" field. In addition to calling the appropriate handler, Apollo also automatically destructures the supplied query variables. The handlers defined in the server simply interface the Firestore database to accomplish the desired task and return the queried data as an HTTP response.

Apollo also supports the creation of a subscription server using WebSockets and an AsyncIterator. An Asynchronous Iterator defines an iterator that yields values as future promises that have yet to be resolved. The process by which these subscriptions operate is highlighted in Figure 3.2.1 below. In short, AsyncIterators allow clients to iterate over values as they are published on the server. The GraphQL server then supports subscription queries that bind a client to a WebSocket channel listening to a specific "publish" tag to be placed on the AsyncIterator. Any payload tagged with the tag the client is subscribed to will be sent to them through the WebSocket connection. Moreover, each subscription consists of a mutation type highlighting how the data was updated (create, update or delete) along with a payload containing the updated item.

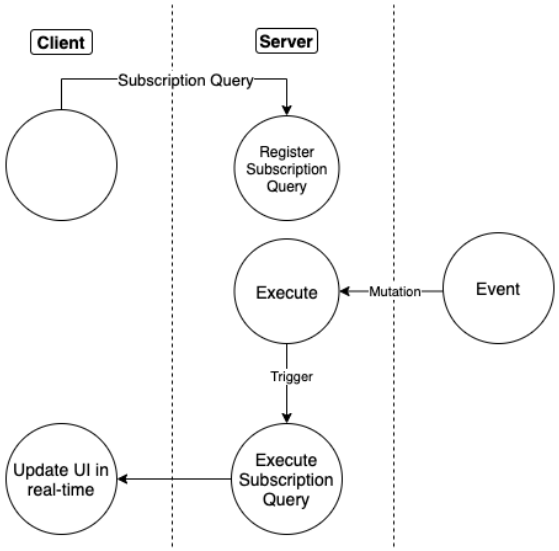


Figure 3.2.1 Graphql Subscription [5]

3.2.2 Software Design for WebSocket Server

The WebSocket Server enables real-time, multiplexed communication to occur between remote nodes. The native WebSocket library for Node JS will be used to create the socket and it will be developed to multiplex/demultiplex data according to a custom specification which is highlighted in the flowchart displayed in Figure 3.2.2. The socket will have a TypeScript object that acts as a record of client subscriptions, mapping client and device ids to a set of all the connection instances that are subscribed to them.

For clients to connect to the WebSocket, they must first request a connection to the socket's URL. This connection request is recognized and automatically upgraded to a virtual WebSocket connection following a TCP handshake. Every client will then send a JSON message with a "subscribe" topic for each id they'd like to subscribe to. The server will then store a reference of the client's connection in the set associated with the ids they are subscribed to. Likewise, the client can send a message with an "unsubscribe" topic and a specified id to be removed from that id's subscription set. The ids are specified as arguments under a "payload" field in the supplied message.

All messages will be parsed on arrival to ensure the correct format. A non-JSON message or one lacking a required field will be replied to with an error. Any message containing a topic other than "subscribe" or "unsubscribe" must also contain the sender's id and a payload that they are sending (which is effectively a nested JSON message). Using the sender's id, all subscribed clients will be forwarded with the message containing the topic and associated payload.

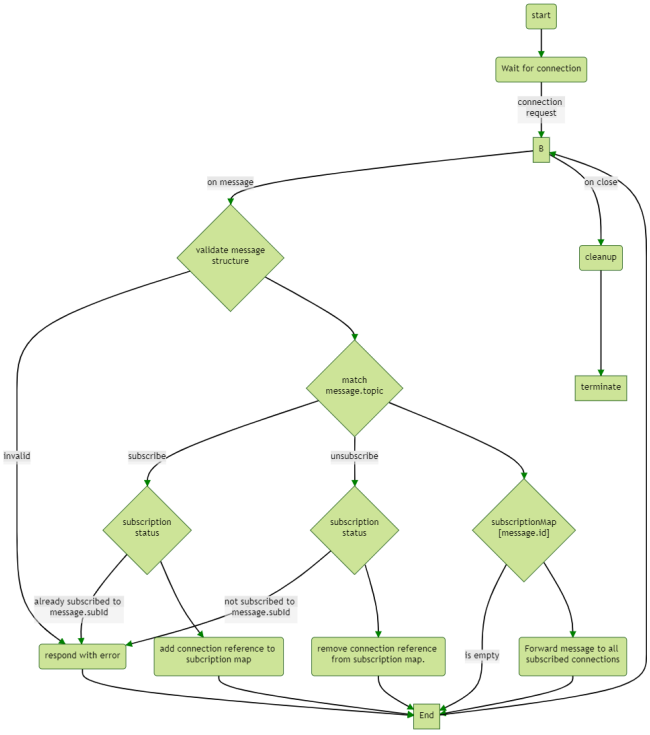


Figure 3.2.2 WebSocket Server Flowchart

3.3 Software Design for WebSocket Client

The Floralyfe system has two WebSocket Client instances: one operating at the Raspberry Pi level and another on the Web Interface. Both operate similarly hence they have been grouped into one discussion. The RPi socket client is written using Python and using an object-oriented paradigm while the frontend socket client is written in TypeScript using a functional approach. The interfaces exposed by both clients and their behaviors are nonetheless similar. Their operation is summarized by the flowchart in Figure 3.3

below. The purpose of the client is to initiate a WebSocket subscription then listen to it, routing all messages that are received to the appropriate node or component that must handle it.

On the Raspberry Pi, each subsystem will have a task queue that its worker thread is monitoring. The websocket client will also have a reference to each task queue along with the topic it is associated with. When a message is received, the client places it in the appropriate queue based on the message topic. If the topic is invalid or not being listened to by any subsystem, then the message is simply dropped.

The Web Interface adopts a similar scheme. Again the WebSocket client will contain a map of message topics but, on the frontend, each topic will map to a set of callback functions. Through the usage of React Hooks, the WebSocket client will expose an interface that allows UI components to add their own callback functions to topic associations. When a message is received, the client will invoke all the callbacks attached to the received message's topic, passing in the message payload as an argument. These callbacks will primarily be used to update a state store shared among the UI components that contains key information such as vitals, plant notes, notifications and other central entity models.

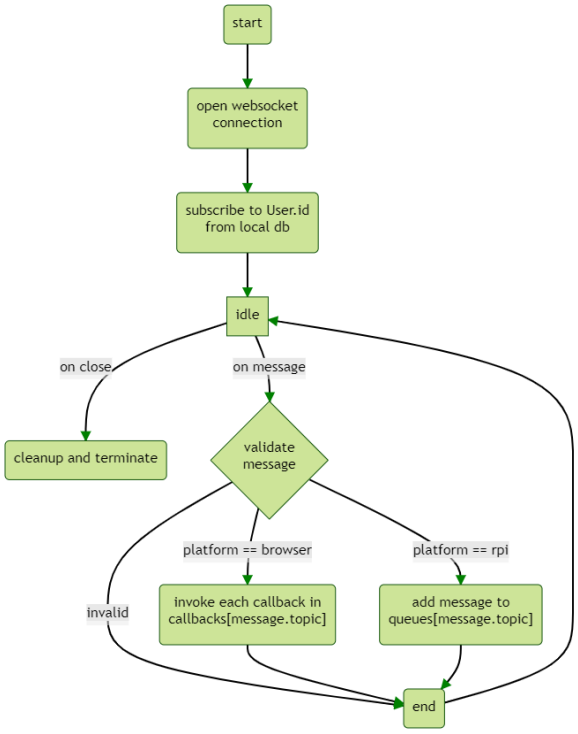


Figure 3.3 WebSocket Client Flowchart

3.4 Software Design for Frontend WebInterface

The most user-facing component of the Floralyfe system is the Frontend Web Interface. The Web Interface, like the servers, will be developed using TypeScript and a functional paradigm. The website will be built on top of the Next JS framework, a derivative of React, and will use the ChakraUI library to style the majority of its components.

The general structure of the applications consists of various UI components such as buttons, forms, navigation, notes, etc. Each component will have its local state and UI definitions. Through the use of React Hooks, components will also have access to a shared central state store through the use of the Zustand library. This central state will contain all stateful values that must be accessed by multiple components or modules. For example, the central store will house all quantities that can be updated by the Raspberry Pi system such as vitals and notifications. Websocket messages will then be used to appropriately mutate this state and components subscribed to the state will react and re-render to reflect the changes. Components will also have access to the WebSocket client through hooks such that they can specify how specific messages should be handled by introducing a callback function. Figure 3.4 below details the operation of the frontend client and the types of supported interactions made possible by the architecture described above. The nature of the interaction is further discussed in Section 5: GUI.

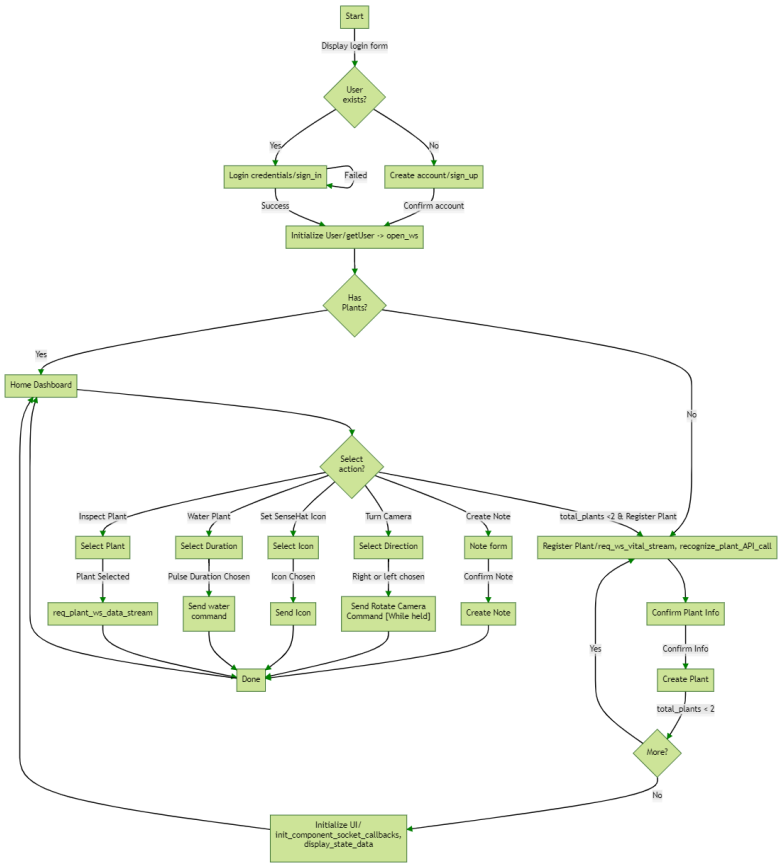


Figure 3.4 Frontend WebInterface Flowchart

4 Hardware Design

4.1 Moisture Sensor

The moisture sensor is part of the Irrigation subsystem and it provides the system with control over soil moisture. There is one moisture sensor per plant plot, the associated hardware connection can be seen in Figure 4.1.1. The moisture sensor used provides an analog output, however, the raspberry pi can only read digital signals. In order for the raspberry pi to interpret the data, the output of the sensor is connected to an

analog to digital converter (ADC), as can be seen in Figure 4.1.2. The ADC that is used is the MCP3008. The ADC has eight input channels and one output. It communicates with Raspberry Pi through the SPI communication ports. The MCP requires a voltage between 3V-5V which can be supplied through the RPi's 3.3V source. The moisture sensor requires a voltage between 3-5V so it will be connected to the 3.3V source from the raspberry pi.

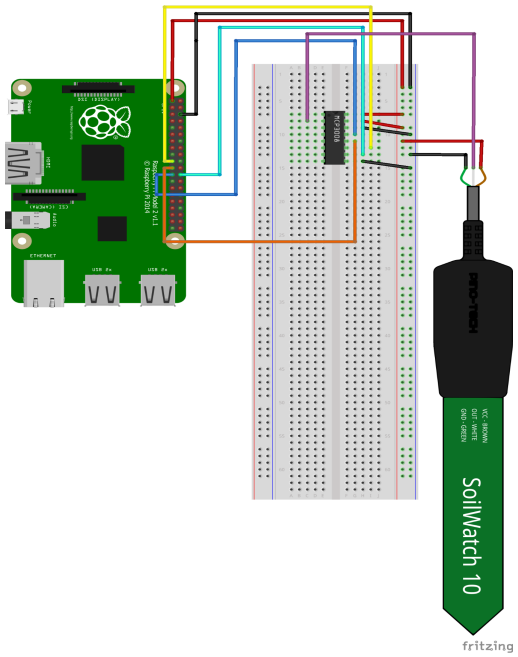
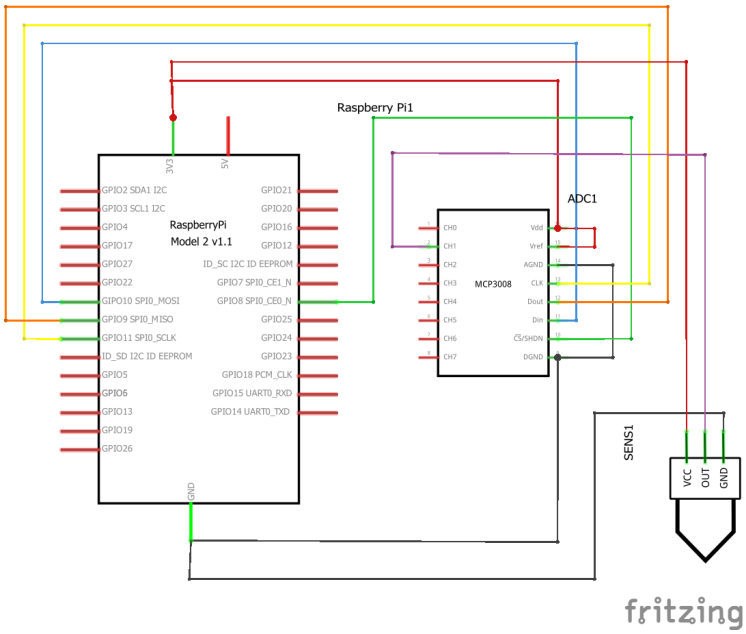


Figure 4.1.1. Moisture Sensor Schematic View Figure



4.1.2. Moisture Sensor Breadboard View

4.2 Water Level Sensor

The water level sensor is part of the irrigation subsystem and it provides the system with the water level inside the watering tank. The system will have one water tank for all the plants and therefore will only require one water level sensor. The hardware setup of the water level sensor can be seen in Figures 4.2.1 and 4.2.2. Similar to the moisture sensor, the water level sensor provides an analog output, so it will be connected to the MCP3008 ADC such that the pi can interpret the data as a digital signal. The water level sensor requires a voltage from 3-5V. It will be connected to the 3.3V source on the pi.

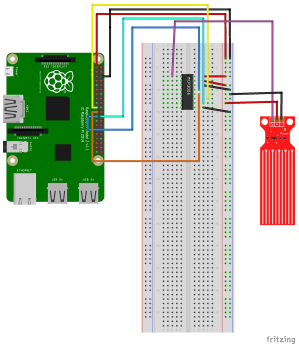


Figure 4.2.1. Water Level Sensor Schematic View

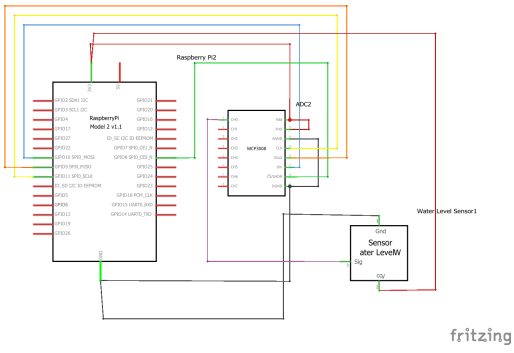


Figure 4.2.2. Water Level Sensor Breadboard View

4.3 Water Pump

The water pump is a part of the irrigation system and it is responsible for pumping water from a water tank to a user's plants. The water pump will be connected to an external power source as it requires more power than the RPi can supply at around 6V. Hence, a transistor will serve as a software-controlled switch to activate and deactivate the pump. The circuit is shown in Figure 4.3.1 and the schematic in Figure 4.3.2. Two resistors will also be required, one in series with the motor to control the power that is supplied to the motor and another one connected to the base of the transistor to control current following the resistor. The base of the transistor will be connected to a GPIO pin. When the system needs to turn on the motor, the voltage of the pin will be set to high and it will be set to low to deactivate the pump.

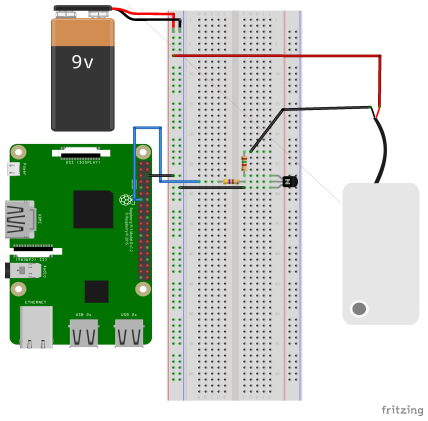


Figure 4.3.1. Water Pump Breadboard View

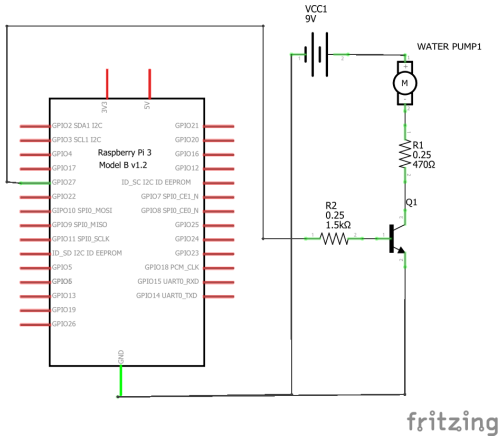


Figure 4.3.2. Water Pump Schematic View

4.4 Servo Motor & Pi Camera

Figure 4.4.2 shows the schematic of the servo motor and Pi camera connected to the Raspberry Pi which is part of the Camera Subsystem. The servo motor has three wires, one for ground, one for power, and one to carry out the command pulses. The black wire in Figure 4.4.1 displays the ground wire being connected to the ground pin on the RaspberryPi. The red wire is connected to pin 18 on the Raspberry Pi, this wire will be the one that carries out the command pulses from the RaspberryPi to the servo motor. Lastly, the yellow wire is connected to the 5V pin on the RaspberryPi which supplies power to the servo motor. The model of the servo motor being used is DFRobot MD9GMS, it has an operating voltage of 3.5 to 5V and a supply voltage of 4.8V. The Pi camera will be connected to the RaspberryPi through its camera module port.

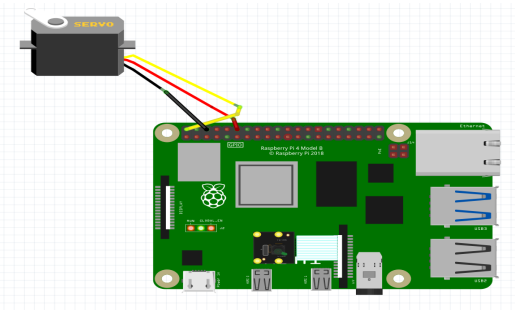


Figure 4.4.1. Camera System Breadboard View

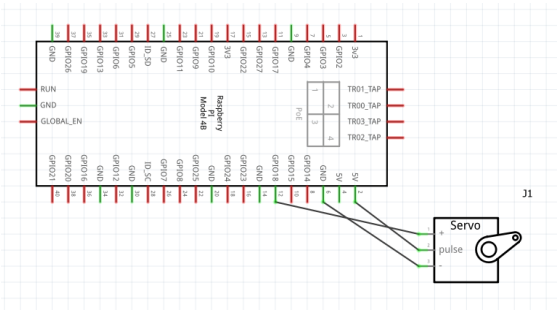


Figure 4.4.2. Camera System (Servo) Schematic View

*Note: Pi Camera is not shown in Figure 4.4.2 but it will be connected to RPi via the camera module port.

4.5 Sense Hat

The Sense Hat is part of the Vital Monitoring subsystem and it is connected to the RPi as seen in Figures 4.5.1 and 4.5.2 below. It provides the temperature and humidity of the atmosphere surrounding the plants. Also, the LEDs on the Sense Hat will display the current health of the plant and can be overridden through the user interface of the system. Only Sense Hat is needed per system. The Sense Hat communicates to the RPi through the I2C pins and it requires both the 3.3V and 5V power sources depending on the hardware being used on the sense hat.

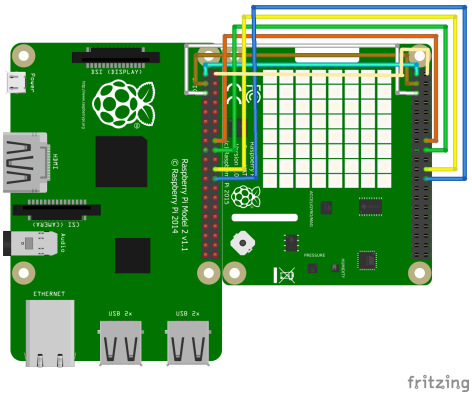


Figure 4.5.1. Sense Hat Breadboard View

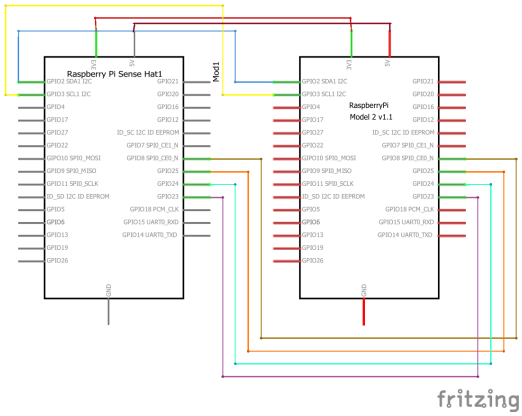


Figure 4.5.2. Sense Hat Schematic View

5 GUI Design

Floralyfe's graphical user interface serves as the user's view into the system and, in turn, the health of their plants. The interface will appear similar to the wireframe displayed in Figure 5.1. below. The majority of the supported interactions are present in the flowchart detailed in Figure 3.4 above but will be further elaborated on below.

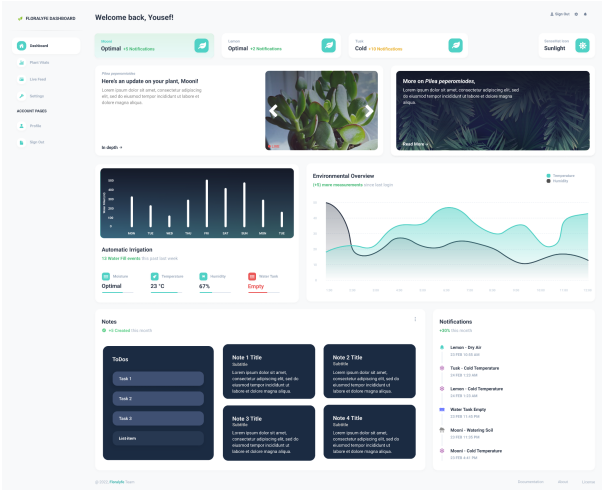


Figure 5.1. Floralyfe dashboard wireframe modified from Purity UI template [6]

The interface will be designed to be a single-page application served on a dashboard layout as seen above. Before accessing the dashboard, users will first be greeted with a login screen where they must either sign up or log in to the system. Following successful authentication, the user will be greeted with the dashboard provided they already have at least one registered plant. In the case of a new account, the system will prompt the user to register at least one plant.

Plant registration consists of displaying a live stream from the user's device and having them send camera rotation commands until their desired plant is in center view. All connected sensor channels will also be displayed and the user can select the ones associated with the plant being registered. Once the user confirms their selection, the frontend uses the last image frame sent to make an API call to a plant recognition service called PlantId. PlantId returns a list of the most probable plant species according to the sent image. This species list is used to query the WUCOLS IV database to estimate the plant's optimal living environment (if available). The user is then presented with the information and can accept the recommendation or override certain fields. The user is also prompted to provide a name for their plant. Once the information is confirmed, a GraphQL call registers the plant on the system and the user is sent to the dashboard. Note that the registration flow is also accessible from the dashboard and can be used to register up to two plants or update the current registration.

From the dashboard, users will be able to press a button on a top bar to specify one of their plants to display information for. Selecting a plant will notify the user's device of the selected plant channel such that the camera turns to face the selected plant and that sensors from the selected plant channel are read. Users will then be able to see a live image feed of the currently selected plant, a live-updating feed of measured vitals along with their corresponding plots, a list of past notifications events for the plant along with its associated notes. Live data feeds are obtained through the WebSocket server while persisted data is requested from Firestore through the GraphQL server.

The user interface is also bidirectional in that it can send control messages to the user's device as well. Namely, the user can press on arrows overlaying the image feed to rotate the servo motor supporting the system's camera. They can also press a water-tank button to override the feedback irrigation control system and manually water the plant for a specified timeout. Finally, the user can toggle between several icons to be displayed on the SenseHat - effectively overriding the Vital Monitoring Subsystem. All of these real-time requests will be transmitted through the WebSocket Server.

5.1 Table of Users/Roles

The *Floralyfe* system is designed to be used by everyday plant owners and consumers. As such, the system only considers a single type of user and associated role: a plant owner. It follows that the system will only expose a single type of web interface that aligns with the specifications outlined in this document.

6 Tests Plans

6.1 End to end Communication Demo Test Plan

Since the Floralyfe system is founded on a distributed architecture, all nodes must be able to effectively communicate. It follows that integration testing, specifically end-to-end communication, will be a core focus

of the team's test plan. At this level, three components must be thoroughly tested: the WebSocket Clients found in the RPi and Web Interface, the WebSocket Server and the GraphQL Server's Subscriptions.

The WebSocket Server and WebSocket Clients complement each other as the latter send and receive messages through a medium mediated by the former. To test both components, an exhaustive list of predetermined messages will be developed to cover all the real-time messages detailed in the tables within Section 2.1: Communication Protocols. Two test scripts will be written on both the frontend and RPi nodes to send messages to the opposite node in succession. Once a message is received, it's payload will be compared to the expected one using the received topic. If no topic is received or the messages don't match, the test will fail. The messages will also be logged as they are sent and received to offer visual feedback. The Raspberry Pi and Web Interface will also perform several mutations on the GraphQL server while the frontend is subscribed to ensure the expected functionality of GraphQL subscriptions.

To be explicit, our end-to-end demo plan consists of the following steps.

1. The test begins by testing the Raspberry Pi's connection to the GraphQL server, and the server's connection to Firebase. A script will be run on a teammate's RPi to create one of each entity supported in our Firestore schema.

2. During the creation of entities, it will be demonstrated that both the vital and notification mutation events are subscribed and sent as mutation events to the frontend client. The associated data will be verified to be correct through comparison to an expected model.

3. Another teammate's RPi will then query the GraphQL server to read all the published models. Again, the acquired data will be compared with expected values to be verified.

4. The frontend client will then read and verify the published data from the GraphQL server.

5. The front end client will then run two functions, triggered by two buttons, to create a user and a note. These will be retrieved and verified by one of the Raspberry Pi clients. This concludes the GraphQL server test.

6. Next, real-time communication through the WebSocket server will be tested. First, a Raspberry Pi node will send a vital message to the frontend which the latter will display and verify.

7. The Raspberry Pi will then be sent a series of messages from the frontend (each triggered by a button) and the messages will be logged and verified by the Pi.

8. The demo will conclude with a live image stream being initiated from a Raspberry Pi and displayed on the frontend.

6.2 Unit Test Demo Test Plan

To ensure proper functionality of each node and to ease debugging, the *Floralyfe* system will adopt various Hardware and Software unit tests for quick system verification.

6.2.1 Hardware Tests

Table 6.2.1. *Hardware test table for devices under test*

Device Under Test	Hardware Test Description
Moisture Level Sensor **(Owner: Abdalla)**	The moisture level sensor provides a reading of the moisture in a specific area. To test that the sensor functions, a script that continuously reads the moisture will be run. The moisture sensor will also be placed in a dry area then placed in water. Both readings will be compared to ensure that they are located at opposite ends of the sensor's operation spectrum which indicates the sensor is functioning.
Water Level Sensor **(Owner: Abdalla)**	The water level sensor provides a reading of the current water level in a water tank. To test its functionality, the same test as the moisture sensor will be executed. The output of the sensor, when placed in water, should differ compared to when it is out of the water. If it differs by a sufficient amount then this indicates that the sensor is functional.
Water Pump **(Owner: Abdalla)**	The water pump supplies water to the plant when moisture is low. To test the functionality of the water pump, the voltage of the GPIO pin that is connected to the switch transistor will be set to high. The user should then observe the pump pumping water. Then after two seconds, the GPIO pin voltage will be set to low and the user should observe the pump as turned off.
Servo Motor & Camera **(Owner: Zakariyya and Yousef)**	To test the Servo Motor, a command will be sent to the servo motor to rotate it to a certain amount of degrees. The servo will then be observed and compared to its previous position through a labeled paper to verify it turned to the appropriate angle.
SenseHat **(Owner: Yousef)**	The Sense Hat will provide readings for both the temperature and the humidity of the atmosphere. It will also be providing the vitality of the plant by displaying it on the LEDs. To test the Sense Hat, a reading of the temperature and humidity will be acquired. Then the user will have to check these values to ensure that they make sense according to their environment - likely with another sensor or their phone. Then the LEDs on the Sense Hat will be set to red and the user will have to check to ensure that the LEDs changed colors.

6.2.1 Software Tests

Table 6.2.2. *Software test table for units under test*

Unit Under Test	Software Test Description
Camera Monitoring System	The Camera System test consists of taking pictures of each plant and uploading them to the local database, then checking to make sure each picture was uploaded to its corresponding

(Owner: Zakariyya)	plant and user. This test will ensure the servo motor is rotating to each plant and taking pictures, as well as that the appropriate data is being uploaded to the database correctly. The system's state machine and expected outputs will also be asserted through a script using the gpiozero mocking factory. This will ensure the system responds appropriately to all its input messages.
Vital Monitoring System **(Owner: Yousef)**	The Vital Monitoring System is responsible for interfacing with all the sensors connected to the Raspberry Pi, reading their data, and sending it to the client. In order to test the system, the hardware will be simulated using both a gpiozero mocking factory, in addition to the tkgpio library. Unit tests will create known scenarios by hard coding the values of certain pins and it will ensure that the system is properly transitioning between states, sending vitals to the cloud, and properly identifying critical vital events. Light and green growth are measured using two custom OpenCV filters. These filters will be tested independently against a set of control images with known output quantities which will be asserted.
Irrigation Subsystem **(Owner: Abdalla)**	The irrigation system is responsible for monitoring and watering the plants. This system will be implemented through a state machine as previously mentioned in section 6.1.1. In order to test this subsystem, the hardware will be simulated using a module called tkgpio. The unit testing of the irrigation subsystem will ensure that the subsystem executes the correct states based on the predetermined simulated stimulus values through the gpiozero mocking factory library.
Websocket Client (RPi and Web Interface) **(Owner: Yousef)**	The WebSocket Clients are responsible for receiving messages and routing them to the correct handler. Routing is either passing the message to a queue in the Raspberry Pi or invoking the appropriate callback in the NextJS frontend. The clients will be tested as part of the end-to-end test - a sequence of known messages will be passed to each running client successively until a final “end” message is sent. All messages will be stored in a queue. Following the receival of the end message, each client will check their received messages against a queue of expected messages to be received. Moreover, clients will be unit tested through being passed local test messages and asserting their routing to the correct subsystem through a callback flag.
WebSocket Server	The WebSocket Server is responsible for allowing clients to subscribe to other clients’ messages: namely, a device should subscribe to its user’s frontend client and that frontend should subscribe to the device. Messages should be routed between both nodes afterward. This communication is also inherently tested in the end-to-end test. However, it is also possible to stub the clients on the server-side. Hence, the server will be

(Owner: Yousef)	tested by creating multiple client stubs along with subscription pairs. It will be verified that clients are able to successfully subscribe and unsubscribe from each other's messages. Moreover, known messages will be sent from one client stub to another and asserted to be received as expected.
GraphQL Server **(Owner: Yousef)**	The GraphQL Server wraps the Firestore database and exposes a common query API to both the frontend and RPi systems. The “supertest” module, along with a Firestore emulator, will be used to stub the server and test its functionality. One by one, each query will be sent to the server and the expected changes to the database, along with the expected response, will be asserted to ensure proper functionality. This will ensure that queries and mutations are working properly. Subscriptions cannot be stubbed as this requires modification of internal library implementations that are abstracted by the Apollo library - hence subscription functionality will only be tested at the end-to-end testing level.
Frontend Web Interface **(Owner: Yousef)**	The frontend interface serves as the user’s view into the *Floralyfe* system. The interface will be tested using the React Testing Library which supports rendering component trees in a simplified test environment and asserting on their output. Each component’s interface will be thoroughly tested to ensure that components react appropriately in response to various external triggers such as entering information in a form or pressing a button. The testing library will also be used on React Hooks to ensure consistency within the shared Zustand state store following various component queries and mutations.

6.3 Final Demo Test Plan

In order to validate the final *Floralyfe* system, a final demo plan is detailed below to serve as an acceptance test.

6.3.1 Remote Vital Monitoring

The remote vital monitoring objective of Floralyfe enables the user to set up plants with vitals that are periodically measured and displayed on the web interface. To test this objective, two plants will be set up before the demonstration for two days. This will allow the system to have enough time to measure some of the vitals and display them on the web interface. Moreover, the periodic vital recording loop will be shortened and vital recording will be demonstrated in real-time.

6.3.2 Automatic Irrigation

The goal of automatic irrigation for Floralyfe is to monitor and ensure that a user's plants' moisture levels are at optimal values. To demonstrate this goal in the final demo, two plant pots will be set up, one with optimally moisturized soil and another one with dehydrated soil. When the system checks the moisture level of the optimal plant, the system should do nothing because there is no need to add extra water.

However, when the system checks the moisture of the non-optimal plant, the system should start pumping water to hydrate. A notification will also be sent and displayed.

Furthermore, the system should demonstrate what happens if the water tank is empty. To demonstrate this, the water tank will be initially empty. Hence, when the dehydrated plant is to be watered, a notification will be sent indicating an empty water tank. The tank will be refilled and the plant will be watered on the next cycle. Once again, the periodic timing loop will be sped up for the demo.

6.3.3 Vital Notifications

The vital notification object of Floralyfe enables the user to keep track of the health of their plants. When a plant's vitals are below the set thresholds, the user should receive a notification with the critical vitals and the LEDs on the Sense Hat should also communicate this information. To demonstrate this objective, two plants will be set up, one with optimal vitals and another with low-moisture vitals. When the system checks the vitals, it should notify the user about the plant with poor vitals. Furthermore, when the user proceeds to see the plants themselves, they should observe that the LEDs also display a moisture icon to indicate a dehydrated plant. All supported vitals will be displayed: soil moisture, temperature, humidity, water level, light, and plant green growth.

6.3.4 Plant Live Camera Feed

The plant live camera feed's objective is to give Floralyfe the ability to frequently send image frames of the selected plant to the frontend. Users are allowed to either alternate between selected plants, upon which the camera will rotate to face the new plant, or be able to rotate the camera freely allowing them to observe their plants' environment. This will be demonstrated by having two plants set up. The user will be able to select a specific plant or rotate the camera to view the plant's environment through the frontend and see their specified plant or its environment.

6.3.5 Plant-specific Notes

Plant-specific notes allow a user to track the progress of their plants by creating notes for each of their plants to list important information, akin to a plant diary, to track plant growth. To demonstrate this functional requirement, two plants will be set up and the user will be able to add notes to a plant of their choosing. These notes will be displayed on the interface. Furthermore, notes will be created before the demo to demonstrate the fetching functionality of persisted notes.

7 Project Update

Development of the Floralyfe system has started well. The team has successfully developed a strong core foundational architecture for the application at both the Raspberry Pi and Server levels. This architecture has successfully enabled end-to-end communication between nodes that is clean and maintainable. Moreover, the WebSocket client system integrated on the Frontend and Raspberry Pi facilitates simple component integration which will be helpful in the future.

The majority of the work to be done on the backend server level has been completed and all that remains is the creation of the frontend interface and integration of sensor data. The website has been created with a rough UI as the effort required for a professional interface was underestimated. As such the interface

component from Milestone 1 has been moved to March 18 as part of Milestone 5. Work on the interface is underway.

The team received hardware components around a week ago and has begun the creation of test circuits and unit tests to completely understand the capabilities and behavior of each component. That being said, integration of the PiCamera along with image transmission and filtering has already been implemented since the camera was available before last week.

In summary, our proposed milestones still make sense though some tasks have been delayed to compensate for the delay in hardware integration and interface creation. Considering future tasks, such as PiCamera integration, to already be completed, the team remains to be balanced and is still on schedule to deliver the final version of the proposed Floralyfe system.

7.1 Project Milestones

In accordance with our team's current workflow, implementation of the Floralyfe system will continue to incrementally take place through a series of revised milestones described in Table 7.1 below.

Table 7.1. *Major milestones in Floralyfe implementation*

Milestone Number	Milestone Name	Description	Deadline
1	Model and View Initialization	Create the frontend interface (with fake data). Create database schemas with adapter wrappers (firebase and local). **[Complete]**	February 23 **[Interface moved to March 18]**
2	Bidirectional and Multiplexed Communication	Achieve multiplexed bidirectional end to end communication with a websocket. Create a backend server to query firebase and integrate with the frontend. Develop a notification sending mechanism and deploy development backend and frontend servers.	**February 27** **[Complete]**
3	Assemble and Interpret Hardware System	Acquire, write to and interpret sensor data. Assemble basic hardware systems. Integrate data with interface and add user login system.	March 13
4	Camera Monitoring Subsystem and Plant Registration	Complete camera monitoring system. Have images being sent periodically from each pi's camera to the frontend. **[Complete]** Add the ability to rotate cameras freely and to face selected plants. Store one picture of a plant, daily, to a local database. Add plant registration system to frontend.	March 17
5	Smart Irrigation Subsystem and	Complete automatic irrigation system. Add a closed feedback loop to check and restore soil moisture levels	March 18

	Establish Vital Notifications	for each plant. Complete notification system to email users of critical vitals and low water tank levels. **[Complete]** Update SenseHat display with representative icon for the plant it's facing. Integrate sensor data on frontend and control watering from client.	
6	Combine Subsystems	Combine all subsystems together and ensure proper functionality. Make final additions to each subsystem, touch-up interface and add ability to create and read notes for plants.	March 20
7	Debugging	Quality assurance check. Run integration tests across the system and perform debugging to finalize the demo procedure and project as a whole.	March 31

Note: Unit tests are to be implemented incrementally throughout each milestone.

7.2 Schedule of Activities

Figure 7.2. below consists of a Gantt chart that displays a timeline with all the aforementioned project milestones and deliverables with their deadlines. A list of tasks leading up to each milestone is found towards the bottom half of the chart. Completed tasks and milestones have been bolded and marked with a check icon.

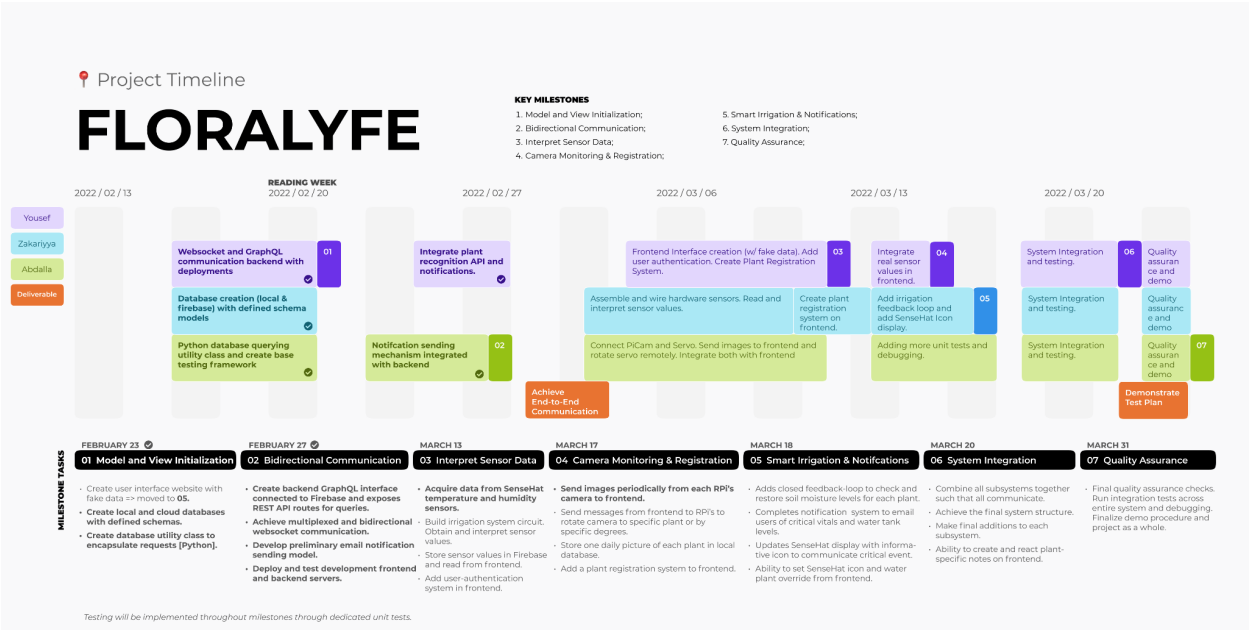


Figure 7.2. Project Timeline Gantt Chart modified from BIAsia Template [7]

References

[1] A. Williams, “How to build a plant monitor with Arduino,” *Electronics Weekly*, 12-Sep-2016. [Online]. Available: https://www.electronicsweekly.com/blogs/gadget-master/arduino/build-plant-monitor-arduino-2016-09/. [Accessed: 09-Feb-2022].

[2] “Houseplant statistics in 2022 (incl.. Covid & Millennials),” *Garden Pals*, 17-Jan-2022. [Online]. Available: https://gardenpals.com/houseplant-statistics/. [Accessed: 08-Feb-2022].

[3] B. Bharti, “The houseplant industry is thriving, thanks to millennials and their 'plant babies',” *nationalpost*, 08-May-2019. [Online]. Available: https://nationalpost.com/news/canada/the-houseplant-industry-is-thriving-thanks-to-millennials-and-their-plant-babies. [Accessed: 08-Feb-2022].

[4] “Raspberry pi camera connector pinout and type (MIPI CSI-2),” *Arducam*, 26-Jan-2021. [Online]. Available: https://www.arducam.com/raspberry-pi-camera/connector-type-pinout/. [Accessed: 08-Feb-2022].

[5] “Graphql subscriptions,” dgraph. [Online]. Available: *https://dgraph.io/docs/graphql/subscriptions/*. [Accessed: 08-Mar-2022].

[6] “Purity UI dashboard by Creative Tim & Simmmple,” *Creative Tim*. [Online]. Available: https://demos.creative-tim.com/purity-ui-dashboard/?_ga=2.165130298.429844829.1644303643-921955921.1644303643#/admin/dashboard. [Accessed: 09-Feb-2022].

[7] “Project roadmap timeline 项目规划时间线,” *Figma*. [Online]. Available: https://www.figma.com/community/file/1045731141584677708. [Accessed: 09-Feb-2022].