# Scrum Assignment 3

## SmartPot

Anthony Fiorito 40000808
Marial Grace De Leon 40008844
Jack Wang 26345220
Fozail Ahmad 40008728

Concordia University
COEN/ELEC 390 Team Design Project
5 December 2017

## ABSTRACT

The following report is the final edition of project document for SmartPot and its companion application, Botanical Journal. The SmartPot is a self-watering pot where users can use their smartphone to access their plant data made available through the Botanical Journal app. The product backlog lists the items that need to be completed in order to fulfil the requirements. All of the required software and hardware components are summarized through a system architecture. Firstly, a moisture sensor retrieves data from the plant and relays it to a Wi-Fi Arduino board. The board then activates a pump to release water to the plant when the moisture level is too low. The moisture data is also sent to the cloud and into the Botanical Journal app where users can keep track of their plants. A wireframe diagram presents the design of the app while the software architecture explains its MVC design patterns and implementation. The hardware architecture describes the hardware components such as the pot, sensor, Arduino board and pump. The link between the hardware and software components is Amazon Web Services. The execution flow of the cloud and app interactions is summarized through use cases and sequence diagrams. Finally, test cases were performed on the software and communication components of the product in order to verify the product.

## TABLE OF CONTENTS

## LIST OF FIGURES

# LIST OF TABLES

# 1 INTRODUCTION

Taking care of plants can be a challenging task due to busy life schedules. The majority of plants require water and sunlight to be able to survive. Our project SmartPot aims to solve issues with owners neglecting their beloved plants. The project is composed of two main parts: a smart pot named SmartPot which will automatically water a plant when moisture levels are low, and an android application named Botanical Journal which receives real time information from the pot for the plants contained within. The hardware aspect of the project involves using a moisture sensor connected to an Arduino microcontroller to automatically water the plant and communicate with the application through the cloud. On the software side, the android application is used to keep track of plants and display the real-time plant data.

## 1.1 Purpose

At its core, the project allows individuals to better maintain their plant life in an automated way. The main goal of the project is to facilitate the watering of plants. The hardware combined with the software will provide a seamless experience for keeping track of and caring for plants. Ideally, all the user would need to do is fill the internal reservoir of the pot with water and the rest will be handled for them. By adhering to a few simple requirements, SmartPot will revolutionize the plant watering experience.

## 1.2 Requirements

The most important requirements for the desired functionality of SmartPot are as follows:

1. There must be an android app for the user to track their plants
2. The app must receive periodic updates on the status of the plant in the pot
3. The pot should connect to the smartphone through the cloud
4. The plant must be automatically watered when moisture level is too low.
5. There must be a container (pot) for the plant to be placed in

## 1.3 Target Audience

Due to the popularity of plant life in many homes, offices and other living spaces around the world, SmartPot can be used by anyone wanting to take care of a plant that will fit inside the pot. The moisture of the plant can be measured and watered automatically for the user. However, some users may not wish to purchase SmartPot but may still want to keep track of their plants. The Botanical Journal app can be used by anyone with an Android device. It is important to note that users not purchasing the pot will not receive real time updates about their plants.

## 1.4 Definitions, Acronyms and Abbreviations

The following table provides explanations for specific terms used throughout the report.

Table 1.1: Definitions for Abbreviations Used Throughout the Report

| Abbreviation | Term | Meaning |
|---|---|---|
| API | Application Programming Interface | Set of definitions or protocols for communicating between different components |
| AWS | Amazon Web Services | A platform from providing cloud computing services |
| HTTP | Hyper Text Transfer Protocol | Primary protocol for communicating between client and server over the web |
| MVC | Model View Controller | Architectural pattern for separating responsibilities among software components |
| REST | REpresentational State Transfer | Methods for communicating over the web |

# 2 REQUIREMENTS: PRODUCT BACKLOG

Table 2.1: Product Backlog

| PRODUCT BACKLOG | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Story ID | Story Title | Card | Story Points | Sprint | Status | Conversation | Confirmation | Story Tag |
| MCU-5 | Sounds | As a user, I want the pot to emit a sound when the water level is low so that I know exactly when I need to refill it | 5 | 4 | Planned | The sound should not be annoying to the user. | 1. Does the pot emit a sound when the water level is low? | Feature |
| POT-4 | LEDs | As a user, I want LEDs on the pot so I can determine the status of the pot by looking at it | 6 | 4 | Planned | There are two options for this: completely redesign the pot or get a drill. | 1. Does the pot display its status via LEDs? | Feature |
| APP-12 | Notifications | As a user, I want to receive notifications about the plant status so that I don't always need to check the app | 9 | 4 | Planned | There would need to be setup of additional server architecture to handle this. | 1. Can the user receive notifications | Feature |
| APP-11 | More Graphs | As a user, I should be able to see water level and last watered data in a graph so I can have a better understanding of my plants over time | 8 | 4 | Planned | The graphs should look similar to the current graph so we maintain a consistent design. | 1. Can the user see a graph of the water level?<br>2. Can the user see a graph of the last watering? | Feature |
| MCU-4 | Water Level | As a user, I want to know when the pot reservoir is empty so that I know when I need to refill it | 7 | 3 | Completed | We'll have to place this in the bottom of the pot. It will require a new API route. | 1. Does the pot tell the user when the water level is low? | Feature |
| MCU-3 | Sending Plant Data | As a user, I want the pot to send data to the application so that I can always have the latest information | 5 | 3 | Completed | The plant data should be sent periodically to the cloud and whenever the plant is watered. | 1. Does the user receive up to date plant data when they load the application? | Feature |
| APP-10 | Remote Watering | As a user, I want to be able to manually water the plant regardless of the water level so that I have full control of my plant. | 5 | 3 | Completed | The Arduino will have to constantly ping the server to check for the remote signal. | 1. Can the user remotely water the plant? | Feature |
| APP-9 | Stop/Start | As a user, I want to be able to start/stop the automatic pot | 5 | 3 | Completed | The Arduino will have to constantly ping the server. | 1. Can the user start the pot? | Feature |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | watering system so that I can use it as a regular pot | | | | | 2. Can the user stop the pot? | |
| APP-7 | Settings | As a user, I should be able to configure the app to my liking so that it is more personalized to my tastes. | 8 | 3 | Completed | We need to think of some settings. setting how often they want moisture to update. | 1. Can the user access their settings? | Feature |
| APP-8 | Moisture Graph | As a user, I should be able to see historical moisture data for my plant so I can track the plant habits. | 10 | 3 | Completed | How many data points should we keep? | 1. Can the user see a historical graph of moisture data? | Feature |
| POT-3 | Connecting Components | As a user, I want the smart pot to be an automated solution to taking care of my plants so that all the work is done for me. | 10 | 3 | Completed | The pot should be assembled so that all components are communicating in a reliable way. | 1. Does the user have to manually check their plant?<br>2. Does the user have to manually water the plant? | Feature |
| MCU-1 | Pump Interfacing | As a user, I want the smart pot to automatically water my plant when the moisture level is too low so that I don't have to do it. | 7 | 2 | Completed | When low moisture is detected the pump should be turned on the rehydrate the plant. | 1. Does the pot automatically water the user's plant when moisture is low? | Feature |
| APP-4 | Adding Plants | As a user, I want to be able to add plants so I can catalogue all my plants. | 3 | 2 | Completed | The plant should be created with several fields available to fill in, so the user can describe their plants better. Should we limit the type of inputs?<br>name, last time watered, date received, plant phylogenic origin, etc. | 1. Can the user add a plant to their collection? | Story |
| APP-5 | Updating Plants | As a user, I want to be able to update my plants so that I can keep them updated. | 3 | 2 | Completed | We might have to validate the inputs by the user | 1. Can the user update a plant? | Story |
| APP-6 | Deleting Plants | As a user, I want to be able to delete my plants so that if they die I don't have to be constantly reminded of their existence. | 3 | 2 | Completed | The delete should cascade so that it deletes all other tables. | 1. Can the user delete a plant? | Story |
| APP-7 | Plant Photo | As a user, I want to be able to take a photo of my plant to display in the app so I can easily identify my plants | 5 | 2 | Completed | The file shouldn't be stored in the database. | 1. Can the user take a photo of their plant? | Story |
| APP-2 | Storing Plants | As a user, I want to be able to keep my plants data after the app has been closed so that I | 5 | 2 | Completed | This will facilitate the user to keep track of their plants even after they quit the app. | 1. Are the user's plants still available after they quit the app and return? | Feature |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | can keep track of my plants for an extended period of time. | | | | | | |
| POT-2 | Printing & Assembling | As a user, I want to interact with the pot without seeing the internals of the smart pot so that I can focus on taking care of my plants. | 7 | 2 | Completed | All the components should be hidden inside compartments contained within the body of the pot. | 1. Can the user interact or see the internal components of the pot? | Feature |
| APP-3 | Navigation Menu | As a user, I want to have a navigation menu so that I can easily traverse the different screens of the app. | 2 | 1 | Completed | The navigation should probably use the default android behaviour so that users don't get confused | 1. Can the user open the navigation menu?<br>2. Can the user navigation to other screens? | Story |
| APP-1 | Displaying Plants | As a user, I want to be able to catalogue/list all my plants along with their status so that I can always have the most up to date information about all my plants. | 5 | 1 | Completed | There should be an indication to the user when there are no plants. There should be the opportunity to display a picture of the plant. Status info should include moisture level (if pot is linked) and last time the plant was watered. | 1. Can the user view their plants?<br>2. Can the user see their plants' statuses? | Feature |
| POT-1 | Smart Pot Model | As a user, I want the smart pot to contain my plant and electronics so that I have a convenient place to store my plant. | 10 | 1 | Completed | The pot should be able to contain all the components inside to make the experience seamless to the user | 1. Can the user place their plant inside the pot | Feature |
| MCU-2 | Moisture Sensor Interfacing | As a user, I want to measure the moisture of my plant so that I can see the health of my plant in terms of water. | 7 | 1 | Completed | The moisture sensor will have to be touching the dirt to detect the moisture of the plant. | 1. Can the user see the moisture level of their plant? | Feature |
| CLD-1 | Plant Linking | As a user, I want to be able to directly link my SmartPot with the plant that is currently in the pot so that I do not have to configure the pot every time. | 5 | 1 | Completed | Each SmartPot would have a unique identifier which the user can use to link to the pot. The data would be persisted to the cloud. | 1. Can the user link the smart pot to their plant? | Story |
| CLD-2 | Fetching Data | As a user, I want the smart pot to continuously keep me updated with plant information so that I am always kept up to date with my plant. | 5 | 1 | Completed | The moisture sensor would be sending updated moisture data from the SmartPot. | 1. Does the user continuously receive new plant information? | Feature |

# 3 DESIGN

## 3.1 System Architecture

The design of the entire project is broken up into three main components: android application, AWS REST API and the SmartPot hardware design/manufacturing. All three components must be carefully designed since they all interact with each other. All data collection from the plant begins inside the pot where the moisture is measured via the moisture sensor. The Arduino will automatically water the plant if the moisture level is too low. In addition, the Arduino will make requests via the internet with its on-board Wi-Fi chip. The plant data is sent to the cloud through a REST API hosted on AWS. From the android application, the data is loaded from the cloud and displayed to the user. Overall, the project requires a combination of many different design domains to create a functioning system (see Figure 3.1).

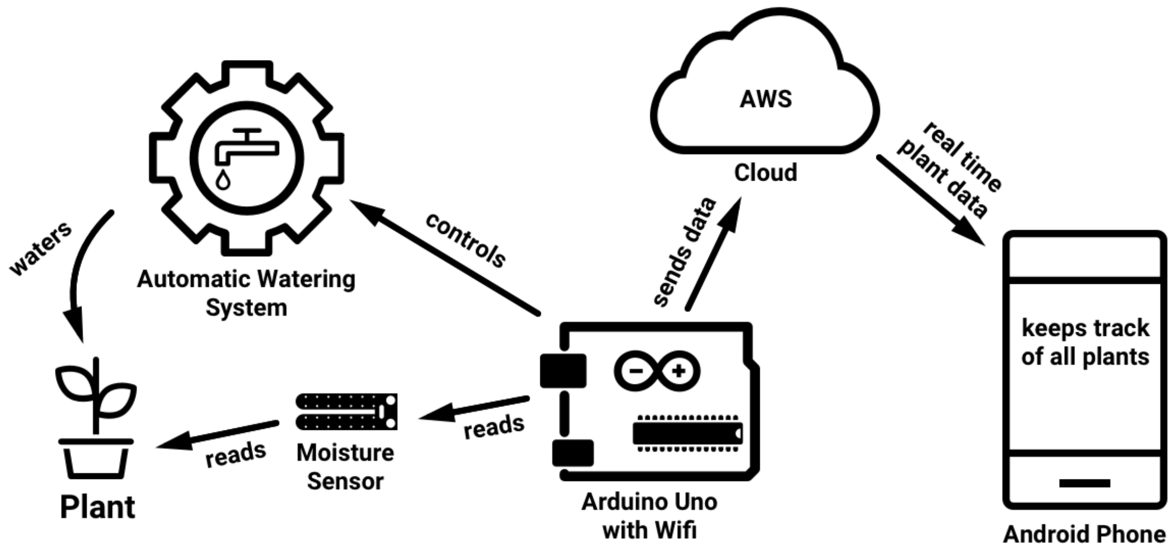
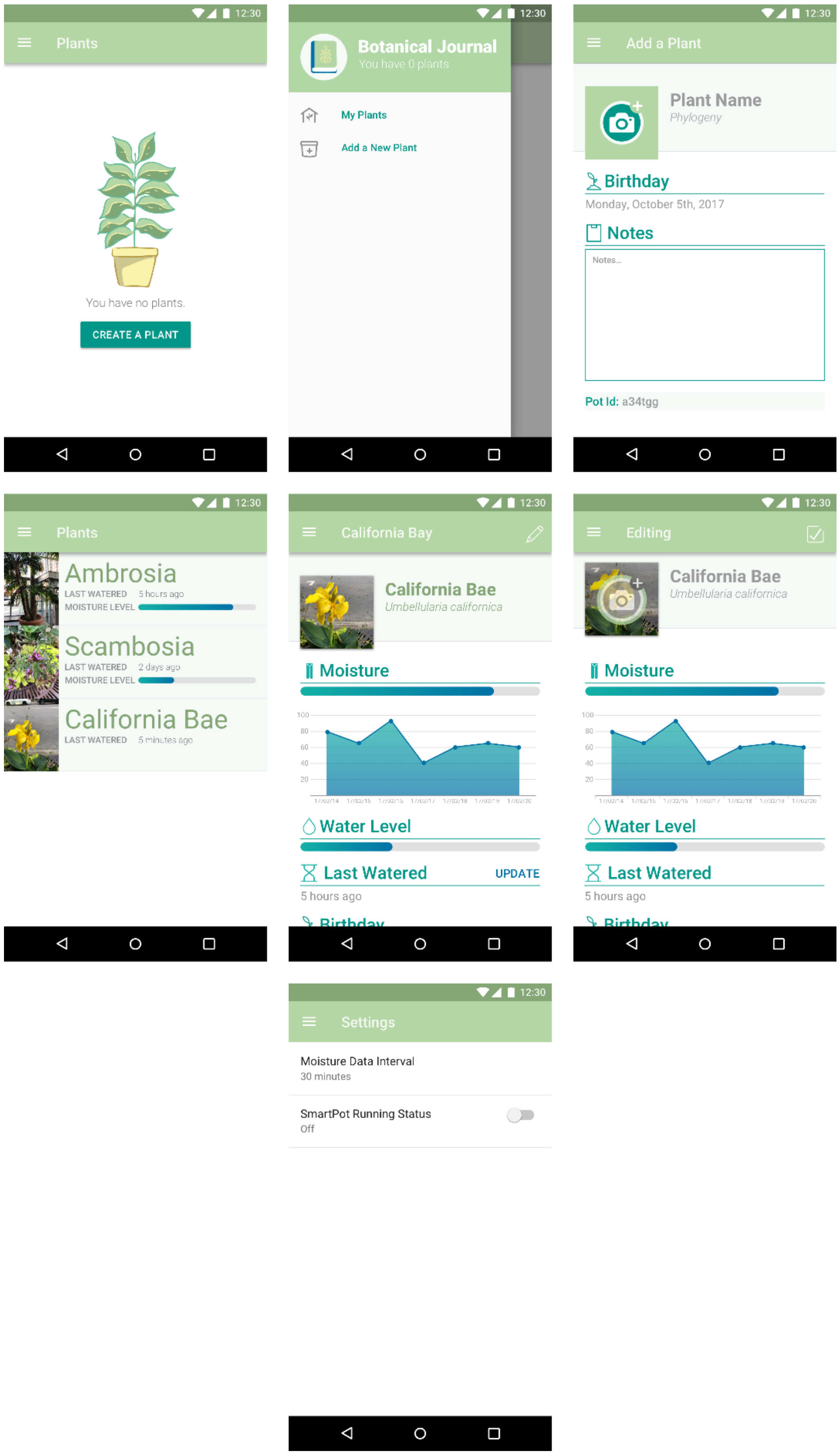


Figure 3.1: SmartPot Architectural Flow

## 3.2 Android Application Wireframe

The wireframes are used to expedite the creation process of the application screens. Having a general idea of how the screens will look before implementing them significantly speeds up the development process. The mockups are made in Sketch, a tool made for rapid prototyping of user interfaces. During the implementation phase, the application is programmed to be as close to the wireframes as possible.



Moisture Data Interval
30 minutes

SmartPot Running Status
Off

Figure 3.2: Wireframe of Botanical Journal App

## 3.3 Software Architecture

The android application uses the popular MVC design pattern for its architecture. The activities/fragments represent the view; it consists of presentational logic only. Moreover, the application makes use of custom modules (controllers) which will communicate with the data model (see Figure 3.3). The main model of the application will store plant data in an SQLite database which includes plant name, type, birthday, moisture data, water level data, last watering time, image path and notes.

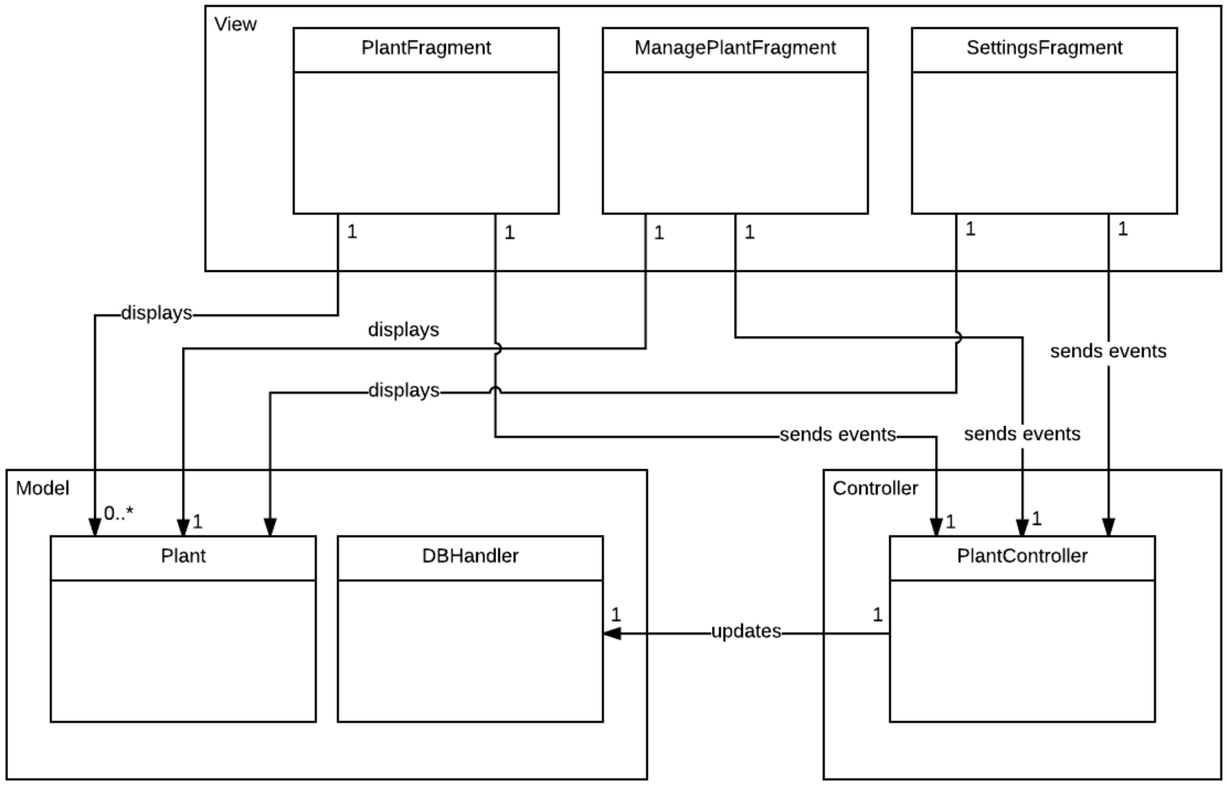


Figure 3.3: MVC Architecture Diagram of Botanical Journal Application

The application contains two essential Fragments which contain the majority of the application's features: PlantFragment and ManagePlantFragment. Both can be accessed via the NavigationActivity which provides a navigation sidebar. PlantFragment is the first view users see when opening the application. It contains a ListView with a custom adapter to display all the plants stored in the SQLite database. To retrieve the plants from the database, PlantFragment (view) delegates the task to PlantController (controller) which contains an instance of DBHandler (model). The controller retrieves the queried data from the model and sends it to the view. In addition, PlantFragment retrieves the real-time plant data from the cloud through the PlantController which makes the HTTP request. The second important view is the ManagePlantFragment which allows the user to view, create, update and delete their plants. The ManagePlantFragment operates similarly to PlantFragment by delegating task to the PlantController which makes calls to the database to return data or updates the model.

## 3.4 Hardware Architecture

The hardware portion of the project requires making a smart self-watering pot. The main hardware components required for a functional pot are: a moisture sensor, a water level sensor, an Arduino (with Wi-Fi) and a water pump (see Table 3.1). The components communicate in order to provide a seamless plant watering experience to the user (see Figure 3.4).

Table 3.1: Use of Hardware Components for the SmartPot

| Component Name | Usage/Purpose |
|---|---|
| Moisture Sensor | Senses the moisture value in the plant soil to be sent to the Arduino. |
| Water Level Sensor | Senses the water level in the pot’s reservoir |
| Arduino with Wi-Fi | 1. Responds to moisture from the sensor and automatically waters plant when moisture is below a threshold.<br>2. Sends moisture data, water level data and last time plant was watered to the cloud via HTTP requests. |
| Peristaltic Water Pump | Receives instructions from the Arduino to begin and stop watering of the plant. |

At the heart of hardware is the Arduino microcontroller. The Arduino board is the WEMOS D1 that has an integrated ESP8266 Wi-Fi chip in order to communicate wirelessly with the AWS API. To upload the Arduino sketches, the WEMOS Arduino board must be connected with a USB-to-serial connection to a computer (see Appendix for the code). In addition, the Arduino is powered by a 12 Volt DC supply to provide the necessary voltage for the pump.

The Arduino receives data from two analog sensors: the water level sensor and the moisture sensor. Since the Arduino only has one analog input, the data from the sensors are controlled using an analog multiplexer, the CD4051B IC which has 8 input/output channels. The IC is powered from the 5V of the Arduino through pin 16 and is grounded at pin 8. The chip receives the moisture data from channel 0 (pin 13) and the water level data from channel 1 (pin 14). To switch between the moisture and water level data, the IC has 3 selects: A (pin 11), B (pin 10) and C (pin 9). Since we only need to read from 2 values, selects B and C are grounded while select A of the IC is switched on and off with the D4 pin of the Arduino.  Finally, the common out/in of the IC (pin 3) outputs the analog values.

The moisture and water level sensors have 3 pins: Signal, Ground and VIN. The moisture sensor is powered by connecting the VIN pin to the D6 pin of the Arduino. In order to receive information from the moisture sensor, the signal pin is connected to pin 13 of the multiplexer. Similarly, the moisture sensor receives power through pin D7 of the Arduino and sends its data to pin 14 of the multiplexer. Ground of the sensors is connected to ground of the Arduino to provide a reference point.

The final hardware component is the peristaltic pump which uses the motor control circuit to control the pump according to the moisture level. The pump is powered by the VIN of the Arduino and grounded to the motor control circuit. In order to protect from inductive kickback, a diode was soldered across the motor's terminals. The motor control circuit consists of a PN2222A BJT and a 330ohm resistor. The BJT is controlled using the D5 pin of the Arduino. When D5 is high, the BJT operates in the saturation region and acts as a short, allowing the motor to run. When low, the BJT is in cut-off and the motor stops.

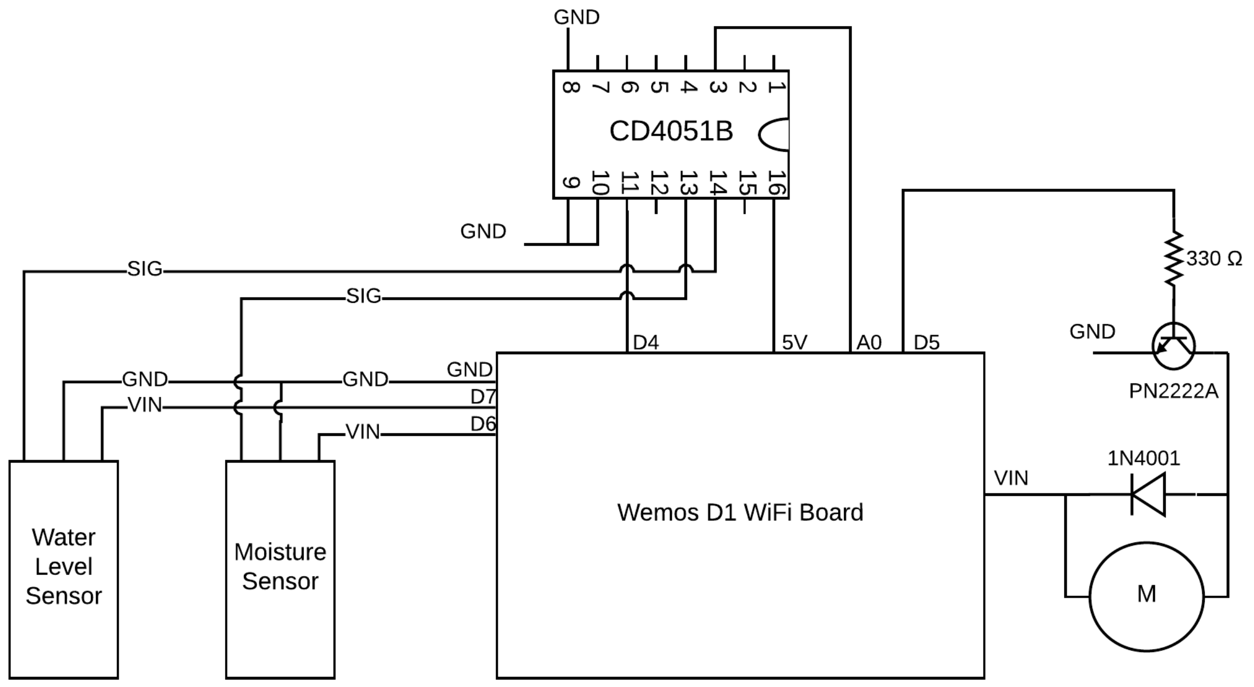


Figure 3.4: Arduino Electronic Circuit

Furthermore, the body of the smart pot was designed in a CAD software for 3D printing (see Figure 3.5). As a result, the body will be mainly composed of polyethylene terephthalate glycol-modified plastic with dimensions 14x14x22cm. It is important that the pot must be able to contain all the required components mentioned above, a reservoir to hold water and a section to hold the plant and soil. Due to the natural separation of responsibilities of the pot, the design can be split up into three layers (see Figure 3.6).

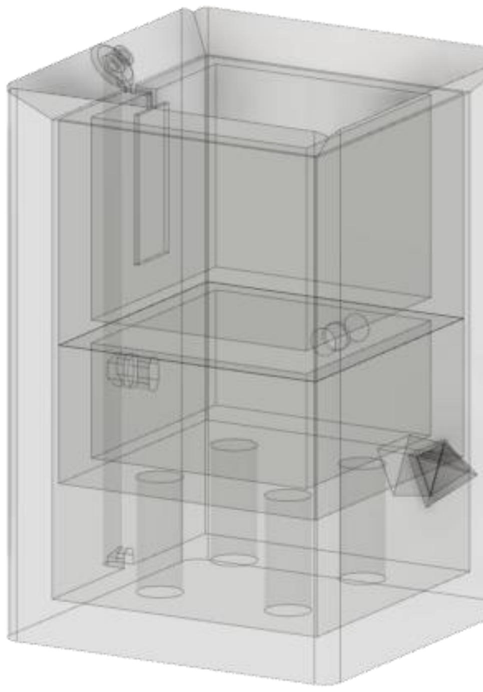


Figure 3.5: 3D Model of The Smart Pot to Be Printed

Firstly, the top layer is where the user places their plant with soil. Secondly, the electronics layer is where we will house the Arduino, the pump and the necessary wiring for connecting components. The pump consists of two tubes: one going to the soil layer above and one into the water layer below. The Arduino is the center of communication for all the other components of the pot. Lastly, the bottom layer contains an empty cavity so that it can hold a reservoir of water which is used by the pump to water the plant.

To construct a three-level design, there is a large exterior case and two interior cases. Inside one of the walls of the external case we have a vertical shaft running from top to bottom that has horizontal access shafts on the interior wall at the floor, at the electronics layer and on the chamfered top. The floor of the external case has raised pillars to support internal cases on top of it and allow a water cavity. On the wall opposite to the shaft, there is a triangular spout which is used to replenish the water reservoir, and the wall perpendicular to the shaft has a cutout at the electronics level for the power cord. The bottom interior case rests directly on top of the pillars inside the exterior case and is for the electronics level. This case also has a cutout for the AC-in adapter which powers the Arduino and cutouts to access the horizontal shafts for the water pipes. The top interior case rests on top of the latter and has a moisture sensor embedded in one of the interior walls so that it can be in direct contact with the soil. Finally, there is a spout object on the chamfered top of the exterior case that is used to direct the water appropriately and hold the pipe in place.

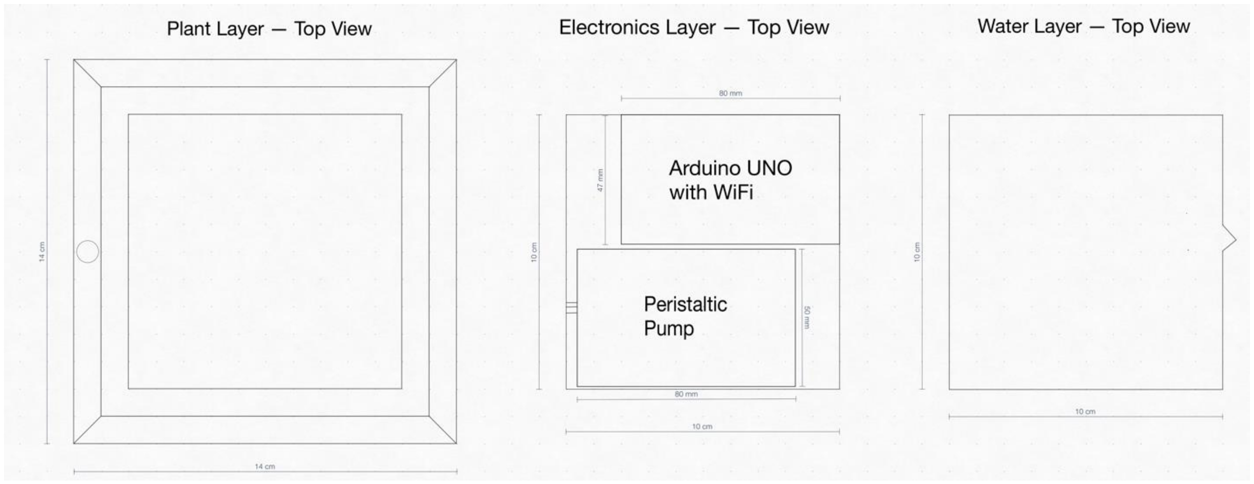


Figure 3.6: SmartPot Layer Composition

## 3.5 Communication Interfaces

In order to communicate between the pot and the android device, the project leverages the AWS Cloud Platform. Although AWS provides different forms of compute services, SmartPot uses AWS Lambda to host a REST API. Lamdba allows for the creation of stateless code containers which only execute when a request is made. As a result, AWS lambda is infinitely scalable since it can add more workers depending on the number of incoming requests. It does not require a constant monthly payment which is common when hosting cloud compute instances. Therefore, using AWS is an economically viable solution to providing a backend on the cloud for SmartPot. The REST API can be accessed via HTTP requests from the Arduino and android device to send and receive data.

In order to store the data, all SmartPots are given a unique identifier which is used by DynamoDB as a primary key. DynamoDB is a NoSQL database hosted on the AWS which provides a simple and performant database solution on the cloud. In addition to the primary key, the database holds information about the smart pot such as moisture level, water level and last watering time. The Lamdba containers can all access the DynamoDB datastore to store and query data.

The code containers are written in JavaScript making use of the NodeJS runtime environment. Every container essentially performs the same execution steps aside from the differences in their computational tasks. The containers respond to the requests through various HTTP method calls such as *GET*, *POST*, *PUT*, *UPDATE*, etc. (see Table 3.2). All containers perform similar execution steps:

1. Receive HTTP request from the client
2. Read the request
3. Perform a database operation
4. Send a HTTP response to the client with success or error

Table 3.2: AWS Lambda REST API

| HTTP method | URL | Function |
|---|---|---|
| GET | `/smarpot/{potId}` | Retrieve a smartpot from the database |
| GET | `/smarpot/{potId}/moisture` | Retrieve the moisture of a smartpot |
| PUT | `/smarpot/{potId}/moisture` | Update the moisture of a smartpot |
| GET | `/smarpot/{potId}/watered` | Retrieve the last watered time of a smartpot |
| PUT | `/smarpot/{potId}/watered` | Update the last watered time of a smartpot |

## 3.6 Use Cases and Sequence Diagrams

The PlantFragment view is an essential component to the plant tracking experience. It requires displaying all the plants from the SQLite database in a ListView using a custom adapter. The PlantController is responsible for interacting with the model (DBHandler) to retrieve the plants from the database. Once the plants are retrieve, they are iterated to check if they contain a potId. If a potId exists for a plant, a network request is made to update the moisture, water level and last watering time of the plant so the user can always have the most up-to-date information. (see Figure 3.7).

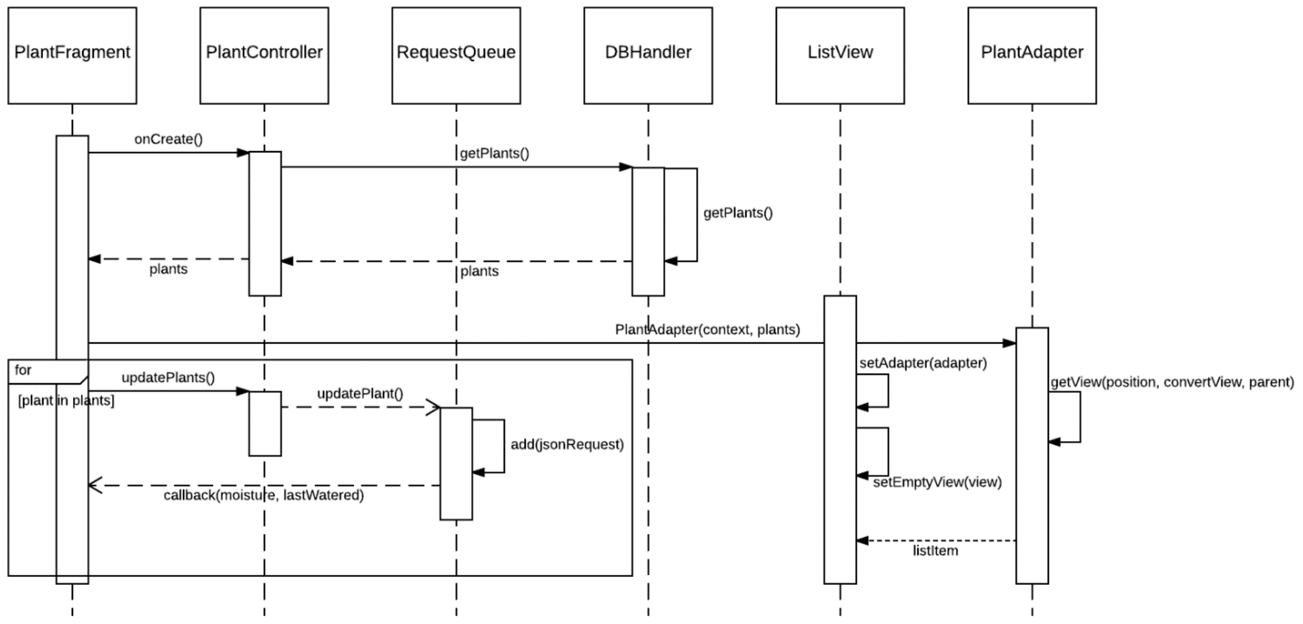


Figure 3.7: Sequence Diagram for PlantFragment

The database handler performs SQLite operations to maintain the plant data inside the application. The typical operations involve:

1. Building the SQLite query string

2. Performing the database query to retrieve a cursor
3. Loop through the cursor to obtain data values
4. Accumulate the values in a data structure
5. Return the accumulated values to the caller

For this use case, the necessary steps for retrieving plants of the data are shown in Figure 3.8. The query selects all plants in the database. After executing the query, the cursor for the plants is returned. A loop goes through the cursor and inserts a new plant into an ArrayList of Plants for each entry in the cursor. Finally, the method returns the ArrayList of Plants to the caller.

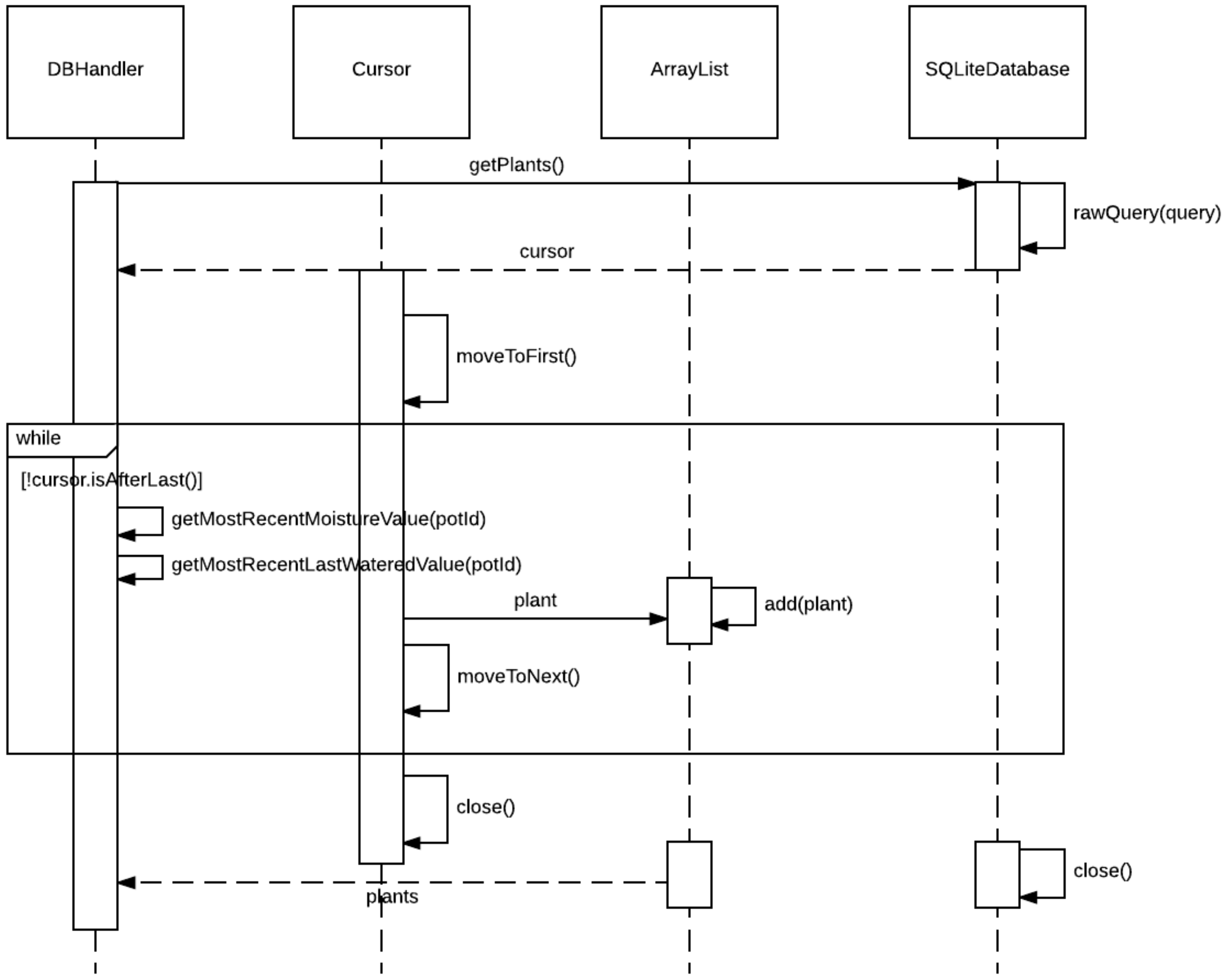


Figure 3.8: Sequence Diagram for Database Handler

The most complex sequence of events currently implemented in the application is creating and updating plants (Figure 3.8). When the user presses the done button, the application must save all the plant data to the database. The potId field of a plant is unique since it needs to be compared to a potId in a database on the cloud. Therefore, the first step when creating or updating a plant is to check if the pot id entered is valid. The following steps involve ManagePlantFragment performing all business logic through the controller and model. This involves adding a plant to the database when creating a plant and updating a plant in the database when updating a plant.

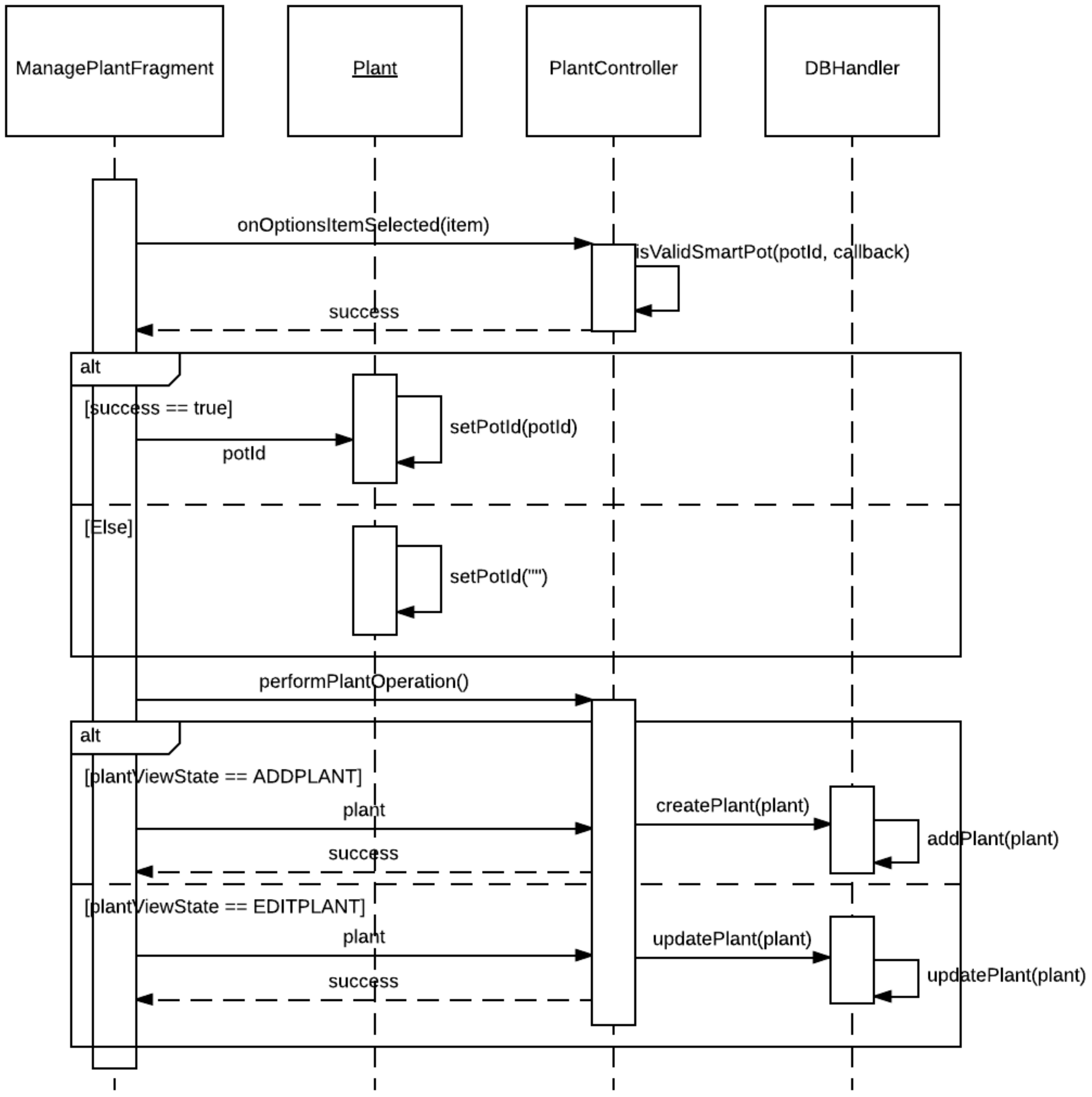


Figure 3.9: Sequence Diagram for Creating and Updating Plants

The SettingsFragment provides useful features for the user. It allows a user to customize their plant tracking experience and directly interact with the pot. For example, one of the settings allow the user to turn on and off the pot with the flip of a switch. The sequence of events to implement this feature involve toggling a switch which initiates an asynchronous request to the server via the Volley RequestQueue. Next, the setting for the plant needs to be persisted across application launches so the toggle action is propagated to the DBHandler to store it in the SQLite database.

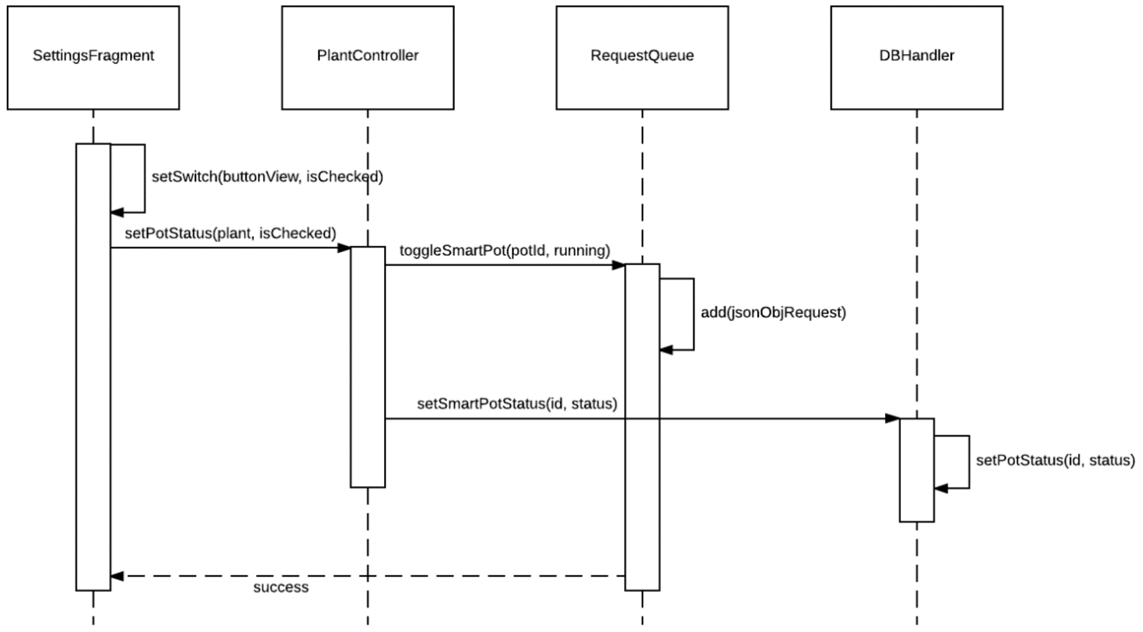

SettingsFragment
PlantController
RequestQueue
DBHandler
setSwitch(buttonView, isChecked)
setPotStatus(plant, isChecked)
toggleSmartPot(potId, running)
add(jsonObjRequest)
setSmartPotStatus(id, status)
setPotStatus(id, status)
success

Figure 3.10: Sequence Diagram for Modifying a Plant Setting

On the server side, all HTTP requests are handled by event driven handler functions (containers). The sequence diagram displays the execution flow to retrieve a smart pot and send it as a response to the user (see Figure 3.11). Although there are many endpoints, the get smart pot handler sequence diagram is similar for other endpoints. The handler is activated once it receives an event (in this case a HTTP request) then performs some database action and returns a response to the client. *Note: JavaScript doesn't require formal classes so the sequence diagram describes the modules or functions that implement the business logic.*

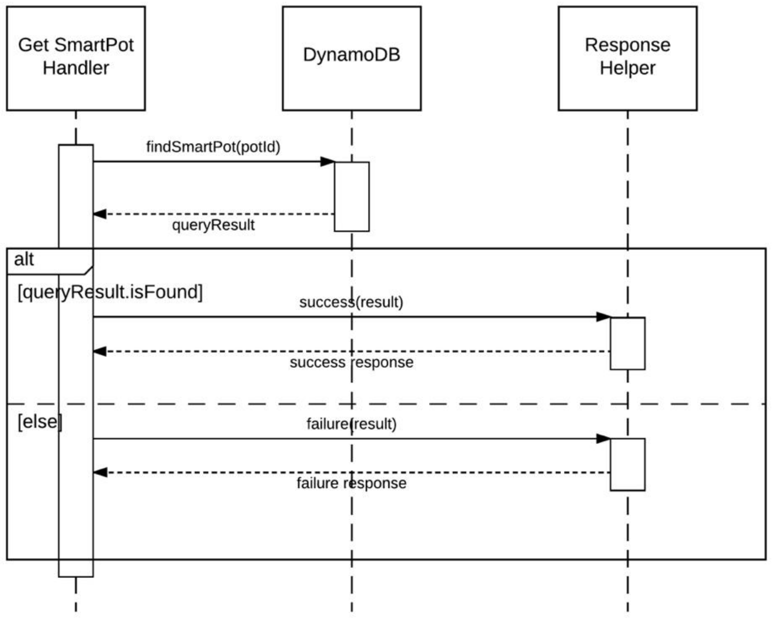
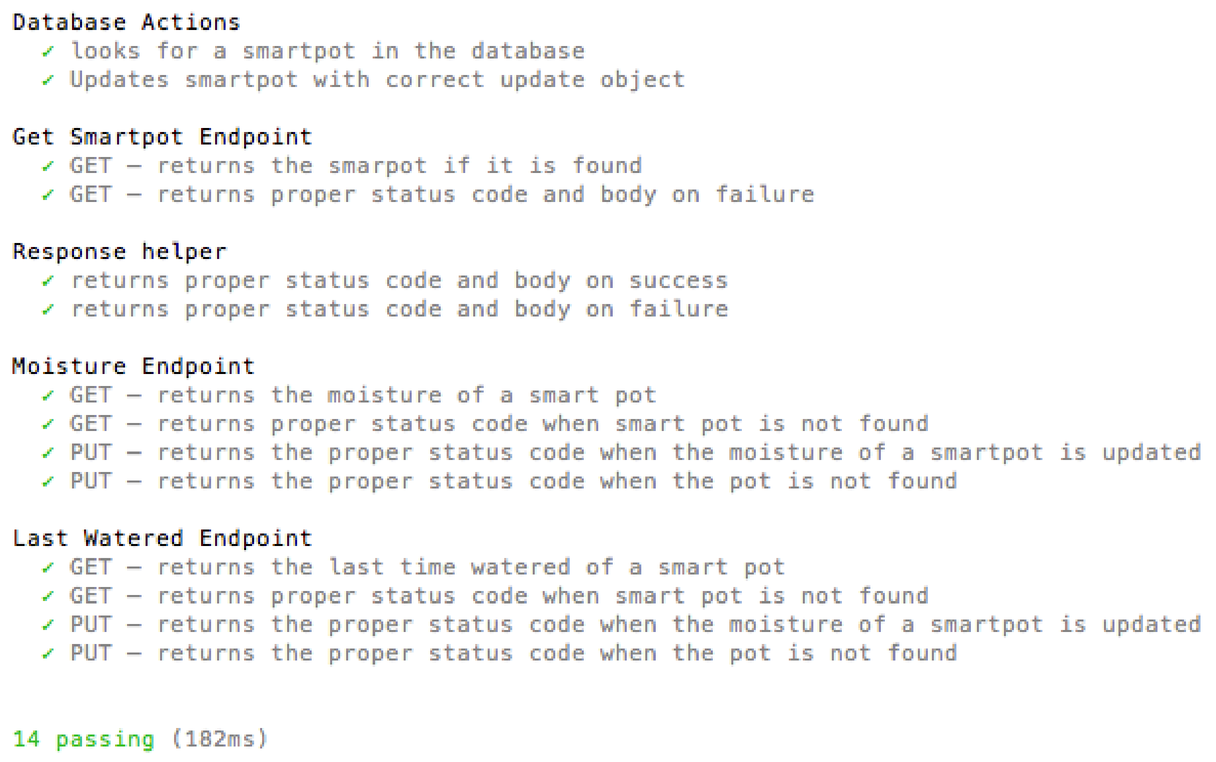


Figure 3.11: Sequence Diagram for Get Smart Pot HTTP Endpoint Handler

# 4 TESTING

The testing performed for the project is mainly black box testing. At the high level testing we are performing, the black box testing is acting more as acceptance tests based on user interaction with the interface of the android application. However, white box testing is being used as well to ensure high code quality and better long-term maintenance of the project. For the server-side JavaScript code, automated testing frameworks mocha and chai are used for unit testing (see Figure 4.1). Moreover, black box testing is performed on all high-level interactive user stories of the product backlog. For example: displaying plants and allowing navigation. As further sprints were completed, more functionalities were tested such as adding a plant, using the device's camera for the plant picture, viewing a plant and deleting a plant. The results of the testing showed that all test cases passed and are performing as expected.



```
Database Actions
  ✓ looks for a smartpot in the database
  ✓ Updates smartpot with correct update object

Get Smartpot Endpoint
  ✓ GET - returns the smarpot if it is found
  ✓ GET - returns proper status code and body on failure

Response helper
  ✓ returns proper status code and body on success
  ✓ returns proper status code and body on failure

Moisture Endpoint
  ✓ GET - returns the moisture of a smart pot
  ✓ GET - returns proper status code when smart pot is not found
  ✓ PUT - returns the proper status code when the moisture of a smartpot is updated
  ✓ PUT - returns the proper status code when the pot is not found

Last Watered Endpoint
  ✓ GET - returns the last time watered of a smart pot
  ✓ GET - returns proper status code when smart pot is not found
  ✓ PUT - returns the proper status code when the moisture of a smartpot is updated
  ✓ PUT - returns the proper status code when the pot is not found


14 passing (182ms)
```

Figure 4.1: Mocha and Chai Unit Testing Output for Server-Side Code

## 4.1 Testing Plan 1: Displaying Plants

### 4.1.1 Summary

**Requirement ID: APP-1, Displaying Plants**

If there are no plants, the user should be prompted to create a plant. On the other hand, if there are plants, the user should be able to see their plants along with the plant details: picture, name, time last watered and moisture level (optional).

#### 4.1.2 Test Cases

Table 4.1: Test – No Plants

| Test Case 1.1 | | |
|---|---|---|
| Pre-conditions:<br>1. Application installed.<br>2. User has no plants | | |
| Steps | Expected Results | Actual Results |
| 1. Launch application | A button prompts the user to create a plant. | As expected. |
| 2. Click on "Create a Plant" button. | The add plant activity is shown. | |
| Result: Pass | | |

Table 4.2: Test – Available Plants

| Test Case 1.2 | | |
|---|---|---|
| Pre-conditions:<br>1. Application installed.<br>2. User has plants | | |
| Steps | Expected Results | Actual Results |
| 1. Launch application | Available plants are displayed with name, picture, time last watered and moisture level (optional). | As expected. |
| Result: Pass | | |

## 4.2 Test Plan 2: Navigation Menu

### 4.2.1 Summary

**Requirement ID: APP-3, Navigation Menu**
The user should be able to access different views through the sidebar navigation. From the sidebar, the user should be able to access their plants, the "create a new plant" view and their settings.

### 4.2.2 Test Cases

Table 4.3: Test – Sidebar Navigation

| Test Case 2.1 | | |
|---|---|---|
| Pre-conditions:<br>1. Application launched | | |
| Steps | Expected Results | Actual Results |
| 1. Click on the hamburger button. | The sidebar navigation slides open to the right. | As expected. |
| 2. Click on “My Plants” | The plant activity is shown. | |
| 3. Click on “Add Plant” | The add plant activity is shown. | |
| 4. Click on “Settings” | The settings activity is shown. | |
| Result: Pass | | |

## 4.3 Testing Plan 3: Adding a Plant

### 4.3.1 Summary

**Requirement ID: APP-4, Adding Plants**
The user should be able to add plants to their list of plants. When adding a plant, the user should be able to include the plant’s details such as its name, type species, picture, birthday, notes and the pot ID (if the user has a SmartPot). The overall test covers setting the fields editable or not (Test 3.1). The name, type species and pot ID fields are regular EditText fields. However, the birthday field requires an additional Calendar Fragment for the date dialog. This was also tested (Test 3.2).

### 4.3.2 Test Cases

Table 4.4: Test – Adding a Plant

| Test Case 3.1 | | |
|---|---|---|
| Pre-conditions:<br>1. Application launched | | |
| Steps | Expected Results | Actual Results |

| | | |
|---|---|---|
| 1. Open the navigation sidebar and click on “Add Plant”. | The Add Plant activity is shown along with editable fields for the Plant Name, Type Species, Birthday, Notes and PotId. | As expected. |
| 2. Enter plant information or leave the information fields empty for name, phylogeny, birthday, notes | The user should be able to enter the information into the appropriate fields | |
| 3. Click on the Done icon | The screen is now in view mode. The information entered in the fields are retained and the fields should not be editable. Only fields with entered information should remain. | |
| Result: Pass | | |

Table 4.5: Test – Birthday Input

| Test Case 3.2 | | |
|---|---|---|
| Pre-conditions: 1. Application launched 2. Opened the navigation sidebar and clicked on “Add Plant” | | |
| Steps | Expected Results | Actual Results |
| 1. Click on “Enter Birthday”. | A dialog opens to enter a date. | As expected. |
| 2. Click “Okay.” | The “Enter Birthday” text in the Profile is replaced with the entered birthday. | |
| 3. Click on the Done icon | The plant’s birthday is visible and not editable. | |
| 4. Click on the Edit icon and click on the “x” next to the birthday | The birthday text is replaced once again with “Enter Birthday”. | |
| 5. Click on the Done icon. | The text field is not editable. | |
| Result: Pass | | |

## 4.4 Testing Plan 4: Camera for a Plant’s Image

### 4.4.1 Summary

**Requirement ID: APP-7, Plant Photo**
When viewing their selected plant in editing mode, the user should be able to click on the image field and prompt to take a picture for their plant’s profile. This image should thus be displayed as the image in the plant’s profile.

### 4.4.2 Test Cases

Table 4.6: Test – Take a Photo With the Camera

| Test Case 4.1 | | |
|---|---|---|
| Pre-conditions:<br>1. Application launched<br>2. Viewing a plant in Edit mode | | |
| Steps | Expected Results | Actual Results |
| 1. Click on Add Image picture | The default camera application launches | As expected. |
| 2. Take a picture and confirm the image | The Add Image picture displays the taken picture | |
| Result: Pass | | |

## 4.5 Testing Plan 5: Storing a Plant

### 4.5.1 Summary

**Requirement ID: APP-2, Storing Plants**
After the user has created a plant, the plant information should display in the profile after editing (Test 3.1). The plant should also appear in the list of user’s plants. Finally, when the user closes the application, previously created plants should remain accessible.

### 4.5.2 Test Cases

Table 4.7: Test – Storing a Plant

| Test Case 5.1 | | |
|---|---|---|
| Pre-conditions:<br>1. Application launched<br>2. Created at least one plant by opening the navigation sidebar, clicked on “Add Plant” and clicked on the Done icon<br>3. Viewing the recently created plant. | | |
| Steps | Expected Results | Actual Results |
| 1. Open the navigation menu and click on “View Plants” | The newly created plant is at the bottom of the list of plants with all its entered information. | As expected. |
| 2. Close the application and re-open it | The previously created plants are available with all their entered info. | |
| Result: Pass | | |

## 4.6 Testing Plan 6: Deleting a Plant

### 4.6.1 Summary

**Requirement ID: APP-6, Deleting Plants**
When the user is viewing a plant in editing mode, a delete button should be at the bottom of all the details. If the user clicks on the button, the plant should be removed from the plant list.

### 4.6.2 Test Cases

Table 4.8: Test – Deleting a Plant

| Test Case 6.1 | | |
|---|---|---|
| Pre-conditions:<br>1. Application launched<br>2. Created at least one plant and viewing its profile in Editing mode | | |
| Steps | Expected Results | Actual Results |
| 1. Click on the delete button | The list of plants is displayed without the deleted plant | As expected. |
| Result: Pass | | |

## 4.7 Testing Plan 7: Linking a Plant

### 4.7.1 Summary

**Requirement ID: CLD-1, Linking Plants**
When the user enters a potId, it needs to be verified with the server to determine if the potId is valid. The http request should not block the main thread but instead should provide feedback to the user that the network request is in progress.

### 4.7.2 Test Cases

Table 4.9: Test – Linking Plants

| Test Case 7.1 | | |
|---|---|---|
| Pre-conditions:<br>1. Application launched<br>2. Connected to the internet | | |
| Steps | Expected Results | Actual Results |
| 1. Open Navigation menu and click create a plant, or view a plant in edit mode | The user will have the options to enter text into the potId field. | As expected. |
| 2. Enter a potId in the potId field and click the Done icon | If potId is valid the user will be in view mode with a visible moisture level, water level and pot settings. Otherwise, the user will be in view mode but a toast will be displayed telling the user their potId is not valid. The moisture level, water level and pot settings are not visible. | |
| Result: Pass | | |

## 4.8 Testing Plan 8: Choose Pot Settings

### 4.8.1 Summary

**Requirement ID: APP-7, Settings**
If the user's plant has a valid potId, when editing their plant, a settings button should appear for the user to configure their pot. The settings page will allow the user to set the moisture data interval which will display in a graph. The user could have updates every 30 minutes, hourly, daily or weekly. Furthermore, the user should be able to turn off their SmartPot through the settings page. By default, the interval should be set to 30 minutes and the SmartPot off.

### 4.8.2 Test Cases

Table 4.10: Test – Choosing Pot Settings

| Test Case 8.1 | | |
|---|---|---|
| Pre-conditions:<br>1. Application launched<br>2. Created at least one plant with a valid pot Id and viewing its profile in Editing mode | | |
| Steps | Expected Results | Actual Results |
| 1. Click on the Settings button | The user pot settings activity is shown with the Moisture Data set to 30 minutes and the pot set to off. | As expected. |
| 2. Click on the Moisture Data Interval Field | A dialog shows up allowing the user to see updates every 30 minutes, hourly, daily or weekly. | |
| 3. Click outside of the dialog or on the back button | The dialog disappears and the text under the Moisture Data Interval label shows the user's selection. | |
| 4. Click on the SmartPot Running Status switch | The switch toggles and the text under the SmartPot Running Status displays the switch status. | |
| 5. Click the back button and return to the pot's settings | The user's last set settings are displayed correctly. | |
| Result: Pass | | |

## 4.9 Testing Plan 9: Automatic Watering

### 4.9.1 Summary

**Requirement ID: MCU-1, Pump Interfacing**
If the user created a plant with a valid pot Id and enabled the pot on, the pot should automatically water the plant if a low moisture value is detected. On the other hand, when the moisture level is high, the pot should not water.

.

### 4.9.2 Test Cases

Table 4.11: Test – Automatic Watering

| Test Case 10.1 | | |
|---|---|---|
| Pre-conditions:<br>1. Application launched<br>2. Created a plant with a valid pot Id and enabled the pot on | | |
| Steps | Expected Results | Actual Results |
| 1. Fill the pot with dry soil | The pot should start watering. | As expected. |
| 2. Observe the pot watering and the soil gaining moisture | The pot should stop watering. | |
| Result: Pass | | |

## 4.10 Testing Plan 10: Remote Watering

### 4.10.1 Summary

**Requirement ID: APP-10, Remote Watering**
The user should be able to water their plant regardless of the moisture level if they created a plant with a valid SmartPot.

### 4.10.2 Test Cases

Table 4.12: Test – Remote Watering

| Test Case 10.1 | | |
|---|---|---|
| Pre-conditions:<br>1. Application launched<br>2. Created a plant with a valid pot Id with the pot set to on | | |
| Steps | Expected Results | Actual Results |
| 1. Click on a plant with a valid pot Id and view its profile | In the Last Watered section, a water text is shown. | As expected. |

| | | |
|---|---|---|
| 2. Click on the Water button | After a few seconds, the pot starts watering and the last watered time is updated to 0 minutes ago. | As expected. |
| Result: Pass | | |

## 4.11 Testing Plan 11: Start/Stop Pot Watering

### 4.11.1 Summary

**Requirement ID: APP-9, Stop/Start**
In the settings page, the user has the choice to turn on or off their pot watering (Test Plan 8). If the switch is on, the pot will water the plant when the moisture is low. If the switch is off, the pot will not water regardless of the moisture level.

### 4.11.2 Test Cases

Table 4.13: Test – Start/Stop Pot Watering

| Test Case 10.1 | | |
|---|---|---|
| Pre-conditions:<br>1. Application launched<br>2. Created a plant with a valid pot Id | | |
| Steps | Expected Results | Actual Results |
| 1. Click on a plant with a valid pot Id and view its pot settings | The SmartPot is by default set to off. | As expected. |
| 2. Switch the pot on | The pot waters the plant when a low moisture level is detected. | |
| 3. Switch the pot off | The pot does not water regardless of a low moisture level detected. | |
| Result: Pass | | |

## 4.12 Testing Plan 12: Moisture Graph

### 4.12.1 Summary

**Requirement ID: APP-8, Moisture Graph**
If the user's plant has a valid potId, a moisture graph should display the moisture values of their plant according to the interval set in the pot settings. The user should be able to view and hide the graph.

### 4.12.2 Test Cases

Table 4.14: Test – Moisture Graph with New Pot

| | | |
|---|---|---|
| Test Case 9.1 | | |
| Pre-conditions:<br>1. Application launched<br>2. Created a plant with a valid pot Id<br>3. The user has just turned on the pot in the pot settings page | | |
| Steps | Expected Results | Actual Results |
| 1. Click on a plant with a valid pot Id | The moisture level section is shown. | As expected. |
| 2. Wait the amount of time set for the moisture data interval and review the plant's profile | The moisture graph is shown with a point corresponding to the current time and moisture value. | |
| 3. Exit the plant's profile, wait the amount of time set for the interval and renter the plant's profile | Another point is displayed in the moisture graph. | |
| 4. Click on the Moisture Level label | The graph is hidden. | |
| 5. Click on the Moisture Level label again | The graph is shown. | |
| Result: Pass | | |

## 4.13 Testing Plan 13: Water Level

### 4.13.1 Summary

**Requirement ID: MCU-4, Water Level**
A water level indicator should let the user know of how much water is available in their pot to know when the pot needs to be refilled.

### 4.13.2 Test Cases

Table 4.15: Test – Water Level

| Test Case 10.1 | | |
|---|---|---|
| Pre-conditions:<br>1. Application launched<br>2. Created a plant with a valid pot Id<br>3. The user has not added any water to their pot | | |
| Steps | Expected Results | Actual Results |
| 1. Click on a plant with a valid pot Id and view its profile | In the Water Level section, a water level bar should show. | As expected. |
| 2. Add water to the pot | The water level progress bar should have a higher water level | |
| 3. Click on the water text and swipe down on the plant's profile | The pot should water the plant and the progress bar should have a lower level | |
| Result: Pass | | |

# 5 APPENDIX

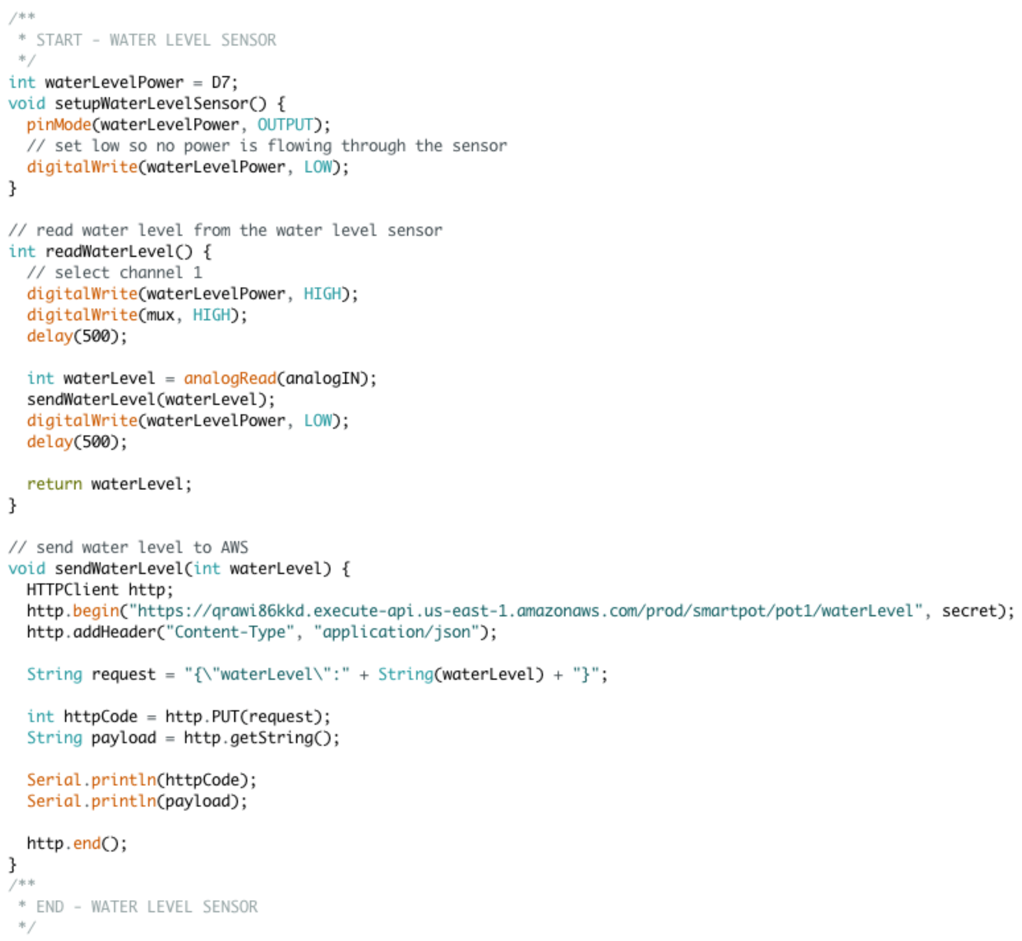
## 5.1 Arduino Sketch

The following sketch was loaded into the microcontroller to communicate between the hardware components and cloud.

```
#include <ESP8266WiFi.h>
#include <ESP8266HTTPClient.h>
#include <ArduinoJson.h>

#define ssid "Powerhouse"
#define pwd "a7b6e7c9d9"
#define secret "65 39 E1 1C AE 10 AF A4 DB EC 8E 7E A3 D4 68 E7 DC 36 F5 70"
#define cycle_time 5000
#define water_time 20000

/**
 * GLOBAL VARIABLES
 */
int motorPin = D5;
int mux = D4;
int analogIN = A0;

/**
 * SETUP
 */
void setup() {
  Serial.begin(115200);

  pinMode(mux, OUTPUT);
  pinMode(motorPin, OUTPUT);
  digitalWrite(motorPin, LOW);  // make sure motor is off
  pinMode(analogIN, INPUT);

  setupNetwork();

  setupMoistureSensor();
  setupWaterLevelSensor();
}

/**
 * LOOP
 */
void loop() {
  if(WiFi.status() != WL_CONNECTED) {
    setupNetwork();
  }

  int potStatus = checkSmartPot();
  switch(potStatus) {
    case -1:
      delay(cycle_time);
      return;
    case 1:
      water();
      setWatered();
      delay(cycle_time);
      return;
    default:
      break;
  }

  int waterLevel = readWaterLevel();
  Serial.println("Water Level: ");
  Serial.println(waterLevel);

  int moisture = readMoistureValue();
  Serial.println("Moisture: ");
  Serial.println(moisture);

  // If there's no water, no point turning the motor on
  //if(moisture < 100 && waterLevel > 5) {
    if(moisture < 300) {
    Serial.println("Watering...");
    water();
  }

  delay(cycle_time);
}
```

```
/**
 * Connect to WiFi network
 */
void setupNetwork() {
  WiFi.begin(ssid, pwd);

  Serial.print("Wating for connection...");

  while(WiFi.status() != WL_CONNECTED) {
    delay(1000);
    Serial.print(".");
  }
  Serial.println();
  Serial.println("Connected");
}

/**
 * Gets smartpot and checks the running & watering status
 * return -1 if smartpot should stop running
 * return  0 if smartpot should run but not start watering
 * return  1 if smartpot should start watering
 */
int checkSmartPot() {
  HTTPClient http;
  http.begin("https://qrawi86kkd.execute-api.us-east-1.amazonaws.com/prod/smartpot/pot1", secret);

  int httpCode = http.GET();
  String payload = http.getString();

  StaticJsonBuffer<200> jsonBuffer;
  JsonObject& root = jsonBuffer.parseObject(payload);

  int shouldRun;
  int shouldWater;

  if (root.success()) {
    shouldRun = root["running"];
    shouldWater = root["watering"];
  } else {
    return -1;
  }

  Serial.println(httpCode);
  Serial.println(payload);

  http.end();

  if(shouldWater == 1) {
    return 1;
  }

  if(shouldRun == 0) {
    return -1;
  }

  return 0;
}

/**
 * Water the plant inside the pot
 */
void water() {
  // turn the motor on by turning on transistor
  digitalWrite(motorPin, HIGH);
  delay(water_time);
  digitalWrite(motorPin, LOW);
  updateLastWatered();
}
```

```
void setWatered() {
  HTTPClient http;
  http.begin("https://qrawi86kkd.execute-api.us-east-1.amazonaws.com/prod/smartpot/pot1/watering", secret);
  http.addHeader("Content-Type", "application/json");

  int httpCode = http.PUT("{\"watering\": 0}");
  String payload = http.getString();

  Serial.println(httpCode);
  Serial.println(payload);

  http.end();
}

// update the last watering time on AWS
void updateLastWatered() {
  HTTPClient http;
  http.begin("https://qrawi86kkd.execute-api.us-east-1.amazonaws.com/prod/smartpot/pot1/watered", secret);
  http.addHeader("Content-Type", "application/json");

  int httpCode = http.PUT("");
  String payload = http.getString();

  Serial.println(httpCode);
  Serial.println(payload);

  http.end();
}

/**
 * START - MOISTURE SENSOR
 */
int moistureSensorPower = D6;
void setupMoistureSensor() {
  pinMode(moistureSensorPower, OUTPUT);
  // set low so no power is flowing through the sensor
  digitalWrite(moistureSensorPower, LOW);
}

// read the moisture from the moisture sensor
int readMoistureValue() {
  // select channel 0
  digitalWrite(moistureSensorPower, HIGH);
  digitalWrite(mux, LOW);
  delay(500);

  int moisture = analogRead(analogIN);
  sendMoisture(moisture);
  digitalWrite(moistureSensorPower, LOW);
  delay(500);

  return moisture;
}

// send moisture to AWS
void sendMoisture(int moisture) {
  HTTPClient http;
  http.begin("https://qrawi86kkd.execute-api.us-east-1.amazonaws.com/prod/smartpot/pot1/moisture", secret);
  http.addHeader("Content-Type", "application/json");

  String request = "{\"moisture\":" + String(moisture) + "}";

  int httpCode = http.PUT(request);
  String payload = http.getString();

  Serial.println(httpCode);
  Serial.println(payload);

  http.end();
}
/**
 * END - MOISTURE SENSOR
 */
```



```
/**
 * START - WATER LEVEL SENSOR
 */
int waterLevelPower = D7;
void setupWaterLevelSensor() {
  pinMode(waterLevelPower, OUTPUT);
  // set low so no power is flowing through the sensor
  digitalWrite(waterLevelPower, LOW);
}

// read water level from the water level sensor
int readWaterLevel() {
  // select channel 1
  digitalWrite(waterLevelPower, HIGH);
  digitalWrite(mux, HIGH);
  delay(500);

  int waterLevel = analogRead(analogIN);
  sendWaterLevel(waterLevel);
  digitalWrite(waterLevelPower, LOW);
  delay(500);

  return waterLevel;
}

// send water level to AWS
void sendWaterLevel(int waterLevel) {
  HTTPClient http;
  http.begin("https://qrawi86kkd.execute-api.us-east-1.amazonaws.com/prod/smartpot/pot1/waterLevel", secret);
  http.addHeader("Content-Type", "application/json");

  String request = "{\"waterLevel\":" + String(waterLevel) + "}";

  int httpCode = http.PUT(request);
  String payload = http.getString();

  Serial.println(httpCode);
  Serial.println(payload);

  http.end();
}
/**
 * END - WATER LEVEL SENSOR
 */
```

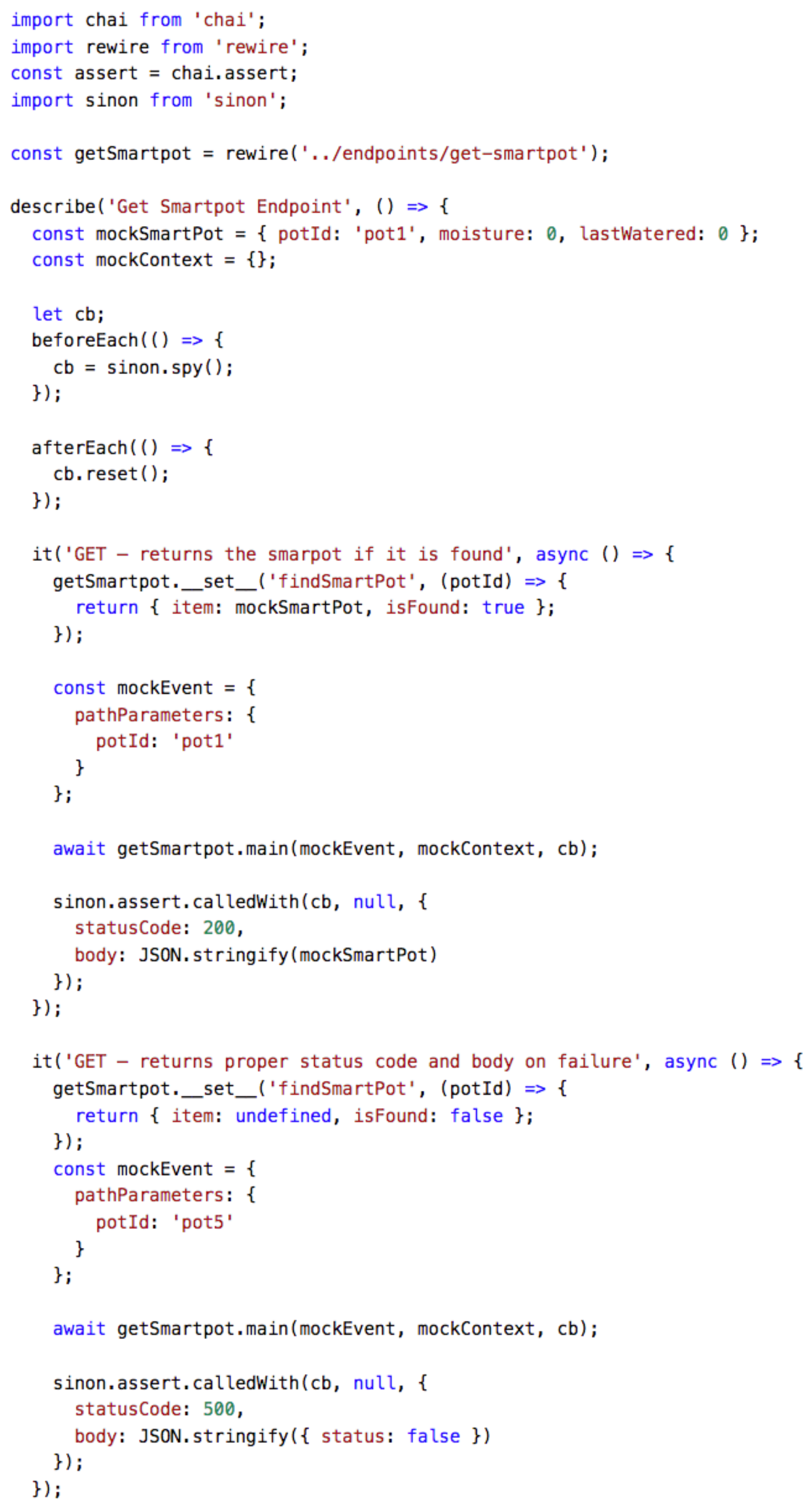
Figure 5.1: Arduino Sketch for the Wemos D1 Microcontroller

## 5.2 Automated Testing with Mocha and Chai

Creating an automated testing environment requires at the minimum two components: a test framework and an assertion library. It is common for the assertion library to be built into the test framework but it is not always the case. Mocha is the test framework which is responsible for running all the test cases and displaying the results of the tests. Chai is the assertion library used to express the expected outcome of the test cases.

Running the test suite is simple. It only requires running a single script from the `package.json` file: `npm run test`. The script executes the mocha command to run the test in watch mode: `mocha –watch –recursive –require babel-register`. This means that the tests will be continuously when the code is being updated to ensure tests are passing at all times. Mocha will recursively look for all files with a `.test.js` extension and run all test cases inside them. The `describe` function indicates a test suite which contains multiple test cases which are denoted by the `it` functions. See the code for the test suite in Figure 5.2.

An extra library sinon is used for spying on functions to check if they were called and creating mocks of functions outside of the testing module. Creating mocks is helpful to test functionality that depends on external dependencies such as another library or a database single whose code cannot be tested. Overall, the testing solution is completed automated so that development can be sped up and code quality can be assured.



```
import chai from 'chai';
import rewire from 'rewire';
const assert = chai.assert;
import sinon from 'sinon';

const getSmartpot = rewire('../endpoints/get-smartpot');

describe('Get Smartpot Endpoint', () => {
  const mockSmartPot = { potId: 'pot1', moisture: 0, lastWatered: 0 };
  const mockContext = {};

  let cb;
  beforeEach(() => {
    cb = sinon.spy();
  });

  afterEach(() => {
    cb.reset();
  });

  it('GET - returns the smarpot if it is found', async () => {
    getSmartpot.__set__('findSmartPot', (potId) => {
      return { item: mockSmartPot, isFound: true };
    });

    const mockEvent = {
      pathParameters: {
        potId: 'pot1'
      }
    };

    await getSmartpot.main(mockEvent, mockContext, cb);

    sinon.assert.calledWith(cb, null, {
      statusCode: 200,
      body: JSON.stringify(mockSmartPot)
    });
  });

  it('GET - returns proper status code and body on failure', async () => {
    getSmartpot.__set__('findSmartPot', (potId) => {
      return { item: undefined, isFound: false };
    });
    const mockEvent = {
      pathParameters: {
        potId: 'pot5'
      }
    };

    await getSmartpot.main(mockEvent, mockContext, cb);

    sinon.assert.calledWith(cb, null, {
      statusCode: 500,
      body: JSON.stringify({ status: false })
    });
  });
```

Figure 5.2: Mocha Test Suite for Getting a Smart Pot