universidade de aveiro
theoria poiesis praxis

deti
universidade de aveiro
departamento de eletrónica, telecomunicações e informática

Grow Mate

GrowMate – Sensing My Home

Projeto em Informática

Relatório Final

2022/2023

Licenciatura em Engenharia Informática

Entregue em 30 de maio de 2023

Trabalho Elaborado Por:

André Butuc (103530)

Artur Correia (102477)

Bruna Simões (103453)

Daniel Carvalho (77036)

Sob a Orientação de:

Prof. José Luís Oliveira, Professor Catedrático no DETI

Abstract

Over the last few years, a growing number of people have begun taking up the hobby of indoor plant care and growing, with the numbers of “plant parents” rising considerably in the western countries after the Covid-19 pandemic, and interest for gardening surging within the millennial’s generation. Alongside this trend, the use of apps for management of daily life activities and tasks has also seen a sharp increase, with people using apps to plan their to dos for the day.

However, the vast diversity of different plants, varieties, and species, as well as the different optimal conditions for the growth of each of them, can be quite overwhelming, not only for newbies of this hobby, but also for people with a large collection of plants already present in their homes.

Thus, we believe that an application that allows the management of indoor plants, serving as a catalogue for information about their conditions and parameters, as a planner for the tasks required to maintain them, and supporting the integration with other plant monitorization systems, like sensors, could be of interest to the market segments interested in the indoor plant gardening activity, from the most experienced users to the least.

As such, we developed the proof of concept for a mobile application with these characteristics, with this report providing the description for the entire software development process, from the early inception and requirements gathering phase, to the code development and usability testing. Though not all of the functionalities and use cases we envisioned for the perfect plant care app were implemented, we believe this project serves as a very solid stepping stone for a reliable and interesting product. Usability tests conducted on the final product at the end of the implementation cycle validated the design of the application, as well as the good flow between the different components and screen of the pages, and the usefulness of the implemented features. Future work is recommended regarding the testing of the system, enrichment of the supporting plant catalogue, and implementation of additional features, such as pest and disease detection.

Keywords

Plant Care App, Indoor Gardening, Full Stack Application, Sensor Monitoring, Spring Boot, React Framework, PostgreSQL, RabbitMQ

Abbreviations

API – Application Programming Interface

APK – Android Package

CRUD – Create, Read, Update and Delete (Operations)

DBMS – Database Management System

HTTP – Hypertext Transfer Protocol

JPA – Jakarta Persistence API

JSON – JavaScript Object Notation

OS – Operating System

ORM – Object Relational Mapping

REST – Representational State Transfer

SQL – Structured Query Language

MQTT - MQ Telemetry Transport

Table Of Contents

List of Figures

List of Tables

1. Introduction

In an overworked society it becomes increasingly difficult to manage several aspects of our daily lives. In order to turn their houses livelier and more accommodating, some people pick up the hobby of raising and taking care of domestic indoor plants. This trend as seen a big surge in recent years, particularly galvanized by the Covid-19 pandemic, and with the millennials generation in particular showing keen interest on this hobby.[1] In fact, around 70% of millennials now call themselves “plant parents”, with 66% of households in America owning at least an indoor plant.[2] The benefits of growing indoor plants are also well known, with studies showing that 15 minutes of interaction with houseplants reduce stress levels,[3] and with the observation that houseplants remove up to 87% of airborne toxins in just 24 hours.[4]

However, for people with little to no experience, this hobby becomes more cumbersome, since the amount of research one must do to raise a plant is very unpleasant and might end up scaring away some new plant owners. In fact, most new plant owners feel very overwhelmed by all the varieties of species, and their respective diverse optimal growing conditions, with most of the worries and troubles when raising plants coming from not knowing their optimal conditions and requirements.

In addition to this, some people tend to forget to take care of the plants they have, which end up going unhealthy and eventually dying. In fact, it is estimated that most new plant owners have killed at least 7 plants they’ve brought home.[5]

With this project, we wanted to create a solution that would solve most of the pains associated with growing plants, in order to allow owners to manage them in an easier and better way and help them get the information they need to raise their plants in a healthy manner. Therefore, the best way a user could reach this information would be in their mobile phone, since it’s a very convenient tool and used broadly every day.

The main goals for our project, therefore, consisted in the development of the proof of concept for a mobile application which would store the information about a user’s plants, the tasks necessary to grow them as well as the monitoring of some of their parameters via the use of sensors. Our system also allows users to find new plants to grow, based on their experience and the plant’s difficulty.

The relevant links for the repositories and documentation of the project, as well as the instructions for installation, are presented in Appendix A.

2. State of the Art

Our application consists, as mentioned in the previous section, of a mobile application that would provide users with services to grow and manage their domestic plants. Our main goal is to help users with plant growth and increase the success rate of raising a plant. The application would also assist users in finding adequate plants based on their experience, for example, if a user has little experience, the application will start suggesting plants with lower maintenance.

In order to better understand how our application is similar to others and know the methods adopted by them, we can acquire inspiration for certain functionalities and add our own.

2.1 Flower Care

Flower care is an application that uses the principle of sensors to better keep track of plants' condition. The idea of modeling our project with sensors and record their measurements in a way that would provide more accurate data about the state of a domestic plant came principally from this application and allowed us to better understand how we would implement the mechanics of sensors in our application and work with them.[6]

This application, *Flower Care,* used sensors of light, temperature, moisture and fertility to monitor the environment around plants and registered the value, displaying different outputs and tasks whenever the records were too extreme, to adapt the plants to those extreme conditions. Depending on the species, the value could be adequate or inadequate since optimal conditions vary widely across species.

It is used with a specific brand of sensors, the *Xiaomi Mi Flower Care Plant Sensor* which allows a quick connection to the application and efficient synchronizing, which measures 4 different types of data: nutrients, soil moisture, temperature and light, which are helpful to monitor the amount of fertilizer, water, temperature and light needed for the plant to develop in the best conditions, respectively.

Since sensors are very specific and related to a pre-existent application, we decided to use three different types of sensors and connect them manually to our application, which correspond to an air humidity sensor, an air temperature sensor and a plant moisture sensor.

2.2 Planta – Care for your plants

Planta is an application that allows users to manage tasks and inform when the next task should be performed. It is also possible, through this application, to divide plants into several sites (divisions). Their priority is to help users reduce the mortality rate of domestic plants and create a user-friendly environment in which users can associate plants to different divisions in order to manage them more efficiently. It is possible for each division to access the different tasks of those plants, which aids the process of completing tasks and remember which plants are already taken care of for the day.[7]

The application also comes with an integrated feature which helps detect, through a picture of a plant, its species and some diseases it might be suffering from, by analyzing visually certain aspects of the plant.

Our application will adapt the division of plants by site, in which the user will be able to associate different plants to a division. Each division can be associated with two sensors, one for air temperature and one for air humidity, and all plants in that division will share the records performed by those sensors.

3. Conceptual Modelling

3.1. System Requirements

In this section, we are going to present the overview of the requirements elicitation process conducted at the start of the project, that allowed us to elaborate the use cases, system requirements and functionalities to be implemented during the project, serving also as a basis for the architectural design decisions taken during the semester.

3.1.1. Requirements Elicitation

With the goal of further elaborating our solution for the presented problem, we decided to do various activities focused on requirements elicitation and gathering, to better understand and formalize the potential features and functionalities of our system, the priority and relevance of their development, as well as to profile the different type of users and market segments that could be interested in our app. In this chapter, we describe the process through which we conduct this elicitation, as well as discuss the main results and conclusions obtained from it.

State of the Art Analysis

First, we started by analyzing the options already available in the market, as discussed on Chapter 2, to figure out what kind of functionalities are already widely present, which are absolutely needed to satisfy the minimum requirements and needs of a user of these sort of applications, and which problems might still not be solved by potential competition, implying the exploration of potential opportunities in the existing market, to further the competitive strength of our solution.

The main conclusions presented on Chapter 2, allowed us to have a first notion of which kinds of functionalities we could explore in this project.

Interviews

After this, we decided to interview potential users of the app, to further elaborate which needs real users would see satisfied from the use of such a system. In order to so, we decided to interview people that fit the two main personas we had envisioned for the system at this point: someone that has no experience with plant growing, that wants to try out this new hobby, but needs further support and guidance; and someone that might have already taken up the gardening hobby, but that sees the value in keeping a central platform to keep a record of their plant inventory as well as the tasks required for their maintenance, and that further might see the value in measuring the state of some plants' conditions with the use of sensors.

After we found people that fit these criteria, we conducted informal interviews with them, where the guiding goal of the discussion was to understand what potential problems, they might have had on their quest for expanding their gardening hobby, and

to realize how an application or a system like ours could help them overcome these problems and help them organize their daily life.

A little summary of the interviews, as well as the main thoughts and conclusions gathered from them, follows:

Ana (26 years old)

Ana is a person with some experience on the growth of indoor plants, having had a very varied collection of plants at home, from succulents to flowering plants.

However, a lot of Ana's plants ended up dying, sooner or later, due to various factors, some of which might be controlled or regularized with monitorization by our app.

For example, Ana had a lot of difficulty in keeping in mind the watering regimes of the different plants she was growing, becoming very overwhelmed with the different conditions required for different varieties. Another big difficulty she encountered was in which room to keep each plant, and how much sunlight they required, has it can be difficult to find the required luminosity and sunlight hours for different plants in a consistent source on the web.

Ana therefore considers that the creation of an app for house plant monitoring, associated with the measurement of some factors by sensors, would be very useful for someone looking to grow new plants the measurement of parameters such as temperature and humidity. She also suggested that a system like this could automatize mechanisms to trigger automatic watering based on the measurements done, with settings being defined for each individual plant.

Maria (30 years old)

Maria never had the responsibility of growing or maintaining indoor plants before, but now that she has moved to a new house with boyfriend, she has decided to buy some new plants for decoration, and potentially invest in the plant growing hobby. However, Maria has a lot of difficulties in knowing which plants are more appropriate for her current climate, which conditions are required for which, and as such she feels like she should start by finding and growing plants that are easy to maintain, and that don't need a lot of assistance to be kept alive.

She has already found some information online on her research, but she considers that having an app that would gather this sort of information in one place would have really helped her on this process, particularly if the app also helped her monitoring the condition of her plants. She would really be interested if the app suggested new varieties to grow as her experienced with gardening increased.

Clotilde (60 years old)

Clotilde has a lot of experience in keeping plants alive, not only in her own house, but as part of her job as a maid for other families.

However, despite having quite a lot of knowledge and experience on this activity, compared with the other interviewed people, Clotilde considers that the watering frequency and the sunlight needs of different plants are variables very difficult to control and to research, with lots of plants having ended up dying or drying up due to the lack of watering, or because they drowned due to overwatering, with the balance being quite hard to find.

Clotilde doesn't have much experience with newer technologies, which means that browsing the internet for this sort of information isn't very easy for her. Therefore, she considers that an app that gathers the information in a central place would be very beneficial for her, with

her main necessities in such an app being eventual functionalities for the automatic scheduling of the tasks to notify her when she should water the plants, or do other activities should has fertilizing the plant or repotting it; it would also be essential for Clotilde that this app is easy to use and intuitive. Another suggestion made by Clotild would be the suggestion of extra nutrients to enrich the soil quality and to help the growth rate of the plants.

At the end of this step, and after several sessions of brainstorming and discussion between the team members where we looked at the conclusions drawn from the state-of-the-art analysis and the interviews, we had a first consensus on the main basic features our system would have to support:

- Central inventory of the house plants owned by the user, that allowed to check the state of the plants immediately.
- The possibility of supporting measurements done by sensors, installed in the plant, to further characterize their current condition.
- Management of the maintenance tasks required for each plant.
- Suggestion of new plants for the user to grow, based on their current inventory or experience.
- Information for the conditions required for each species.

Questionnaires

To validate the features we identified, and to further evaluate them in terms of their usefulness and importance, and how they can potentially solve the problems faced by the target market of our app, we then conducted a wider questionnaire, with the goal being for it to be completed by different people of varying backgrounds.

The full questionnaire, as well as the results and corresponding graphs and tables, are presented in Appendix B.

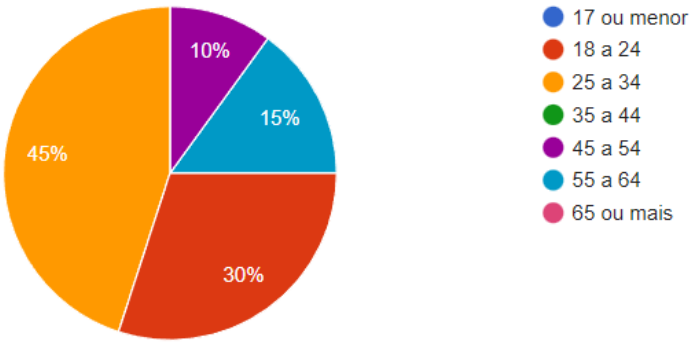


Figure 1 – Demographic information of the users that replied to the questionnaire.

In total, 20 people completed the questionnaire, with a big variety of age groups being represented (Figure 1), with diversity also found in the experience with the growth of plants, with 75% of the users having some sort of previous experience. Out of the latter group, in a scale of 1-5 where 5 means “very experienced”, most considered themselves to have around 2-3 in terms of plant growth experience.

When it comes to problems found when growing plants, most have considered the problems identified during our previous requirements elicitation as being a considerable hardship in their experience, as seen in the results presented on Table 1.

Table 1 – Evaluation of the potential plant growing problems done by the users that replied to the Questionnaire. Users had to assess each of the challenges in a scale of 1-5, in terms of difficulty. The table presents the average difficulty found for each challenge. Only the 15 users who had previously had experience growing plants replied to this specific question.

Challenges	Avg Difficulty*
Appropriate Luminosity	3.4
Humidity	3.3
Watering Frequency	3.3
Soil Change Frequency	3.3
Pot Change	3.1
Plant Acclimatization	3.5

In fact, the challenges regarding the monitorization of the main parameters for plant growth, such has the appropriate luminosity, humidity, watering frequency or the frequency of soil change, had the highest average difficulty found in the questionnaire. Besides these, among the other challenges listed by users when growing plants, are included, for example:

- Having a hard time keeping up with all the maintenance tasks.
- Not knowing when to apply extra fertilizer or nutrients for plant growth.
- Managing pests.
- Managing diseases and noticing symptoms of diseases.

Out of these, over 85% of the users have already had indoor plants dying, with most of them reporting to “have killed” over 3 plants in their lifetime experience. When rating from a scale of 1-5 how difficult it was to keep in mind all the necessary tasks for plant maintenance, over 85% of people replied with a value of 3 or higher (Figure 2), with a big majority of users considering that an application would be really helpful to help them monitor their plants growth and management (Figure 3).

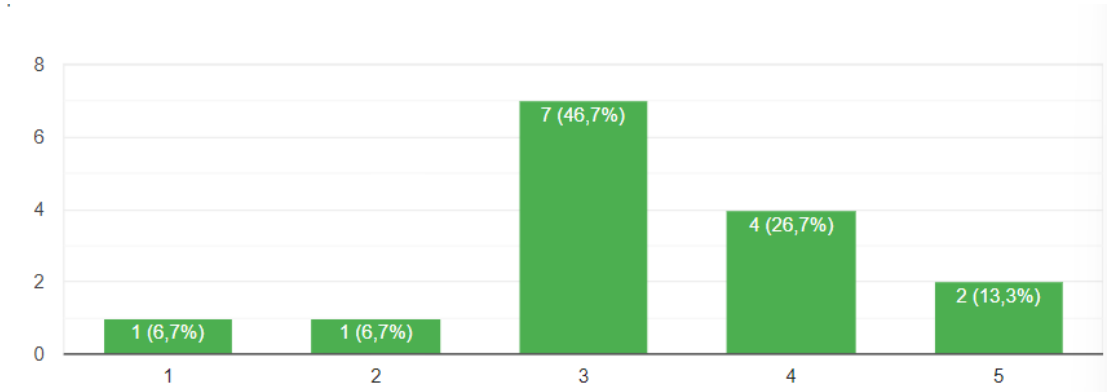


Figure 2 – Replies to the question “How hard was it to keep in mind all of the plants maintenance tasks?”. Users had to evaluate from a scale of 1-5, with 1 being very easy, and 5 being very hard.

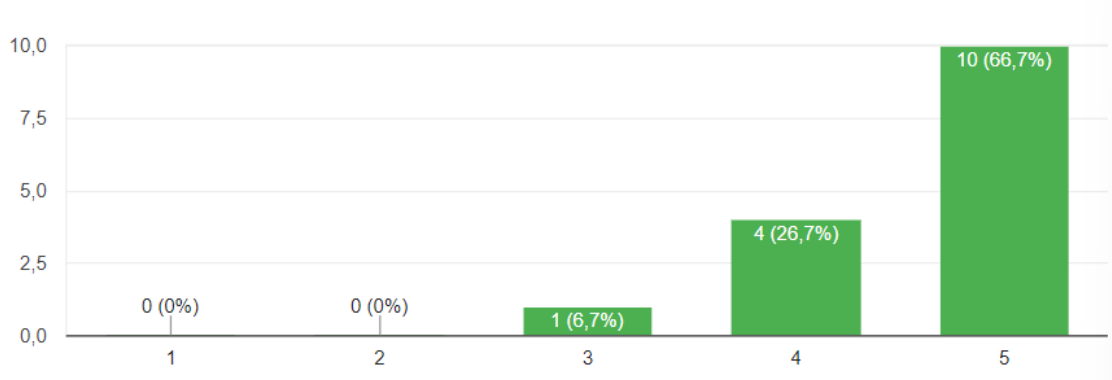


Figure 3 - Replies to the question "In your opinion, how useful would be a mobile app for plant growth monitorization and tasks management?". Users had to evaluate from a scale of 1-5, with 1 being 'not helpful', and 5 being 'very useful'.

As we can see, these results validate the need and interest for an app like GrowMate, for users that want to partake in the indoor gardening hobby. They also validate the main challenges and issues identified as topics of interest by the team at this point.

The next section of the questionnaire dealt with how useful the previously mentioned functionalities would be for users, again, on a scale of 1-5, with 5 being "very useful". The main results are summarized on Table 2.

Table 2 - Evaluation of the potential identified functionalities for the GrowMate mobile app, done by the users that replied to the Questionnaire. Users had to assess each of the functionalities in a scale of 1-5, in terms of usefulness. The table presents the average usefulness found for each challenge.

Functionalities	Usefulness Index*
Suggestion of Optimal Growth Conditions for Plants	4.6
Plant Inventory and Condition Monitorization	4.5
Task Management and Scheduling	4.4
Sensors Integration	4.4
Task Notifications	3.9
Suggestions of New Plants to Grow	3.5
Tips From Other Users	3.2

The team used the results of these questionnaires to finalize the definition of the actors of the system, as well as the use cases and core functionalities associated with them. Besides, these results also allowed us to prioritize our development work, serving as an estimation of the use cases that were necessary to be implemented, as well as of the use cases that were more secondary, and that were defined as being implemented only if time allowed us to do so.

3.1.2. Identified Actors

Following the requirements gathering and elicitation work already described, we identified the main potential actors for our system. One of the main conclusions we had, following the questionnaires and informal interviews, was that for many users, they saw the use of an app such as GrowMate as a very casual daily application, that should be intuitive and easy to use, and where they can see the general plants for the task immediately. Some of these users felt like the inclusion of measurements by different sensors could be quite overwhelming, even scaring them away from installing the app, if it wasn't made clear that such a functionality, and the physical installation of sensors, wasn't required to take some value from the use of the app.

Thus, we decided to clarify that users could use the app without having to install any monitorization equipment, and we felt that the best to differentiate these different use cases or personas for the system, was to develop two separate flows of execution: one for non-premium users, with base plant inventory and task management features and plant suggestion algorithms; and another for premium users that do wish to more closely monitor their plants with sensors. Besides this, when it comes to a business model standpoint, the requirement for a premium version for the application for users that would require physical sensors, that comes with a more complex architecture and system design, would make financial sense if this system was developed as a business in the future.

As such, the final identified actors for the system are presented as follows:

- **System Administrator –** This user is the administrator of the GrowMate platform. They are responsible for the maintenance of the database, being able to perform the various CRUD operations on the catalogue of plant species and varieties supported by our system. They are also able to see operational statistics, and information regarding the users.
- **User Without an Account –** This user is only able to browse the public catalogue of plants supported by the application, as well as the tips and information about them (such as optimal growth conditions, optimal soil mix, among others).
- **User With Non-Premium Account –** Besides the browsing of the plant catalogue, these users have access to all the core functionalities, such has registering plants in their house, checking the required tasks for their maintenance based on their optimal parameters and conditions, personalizing the settings for these tasks, accessing the user forum, and posting tips for other people to see, having access to the plant suggestion algorithm, among others.
- **User With Premium Account –** Has the same functionalities as described for the previous actor, but also with the inclusion of the sensor monitorization. As such, these users will be able to associate sensors to their individual plants (for example, for soil moisture), as well as sensors to the divisions of their house (to measure atmospheric parameters). They will be able to check the records for the measurements over the last few days, and the measurements will be used to estimate the current condition of each plant, and whether they require attention or not.

3.1.3. Use Cases

Having identified the main actors of the system, it was now possible to elaborate use cases for each of them, finalizing the definition of the different functionalities to be supported by the application. The use cases are shown in the diagram presented in Figure 4 and are further explored below.

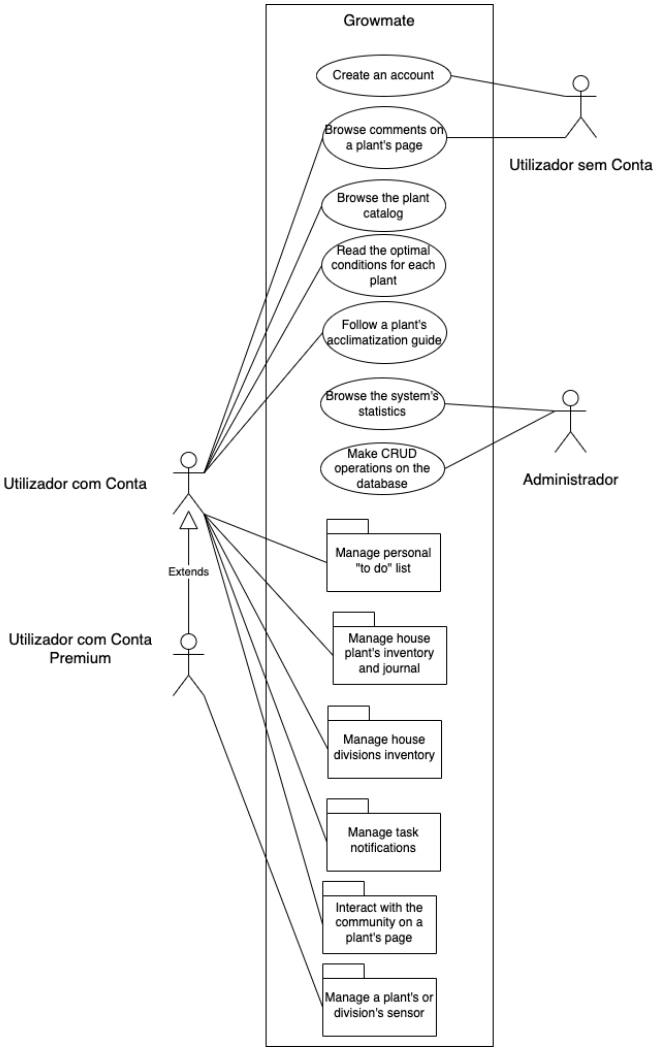


Figure 4 - UML Diagram for the use cases identified after the requirements gathering.

The use cases available for **all users** are as follows:

- **Create an account –** Every person should be able to create a new account on the GrowMate platform.
- **Browse the app's plant catalogue –** Every person, whether they have a registered account or not, can use the mobile app to check the catalogue of plants supported by the app.
- **Read the optimal conditions for each plant –** Everyone can check the informational page about the species supported by the app. These pages should show details about the related plant species, such as scientific and common names, their cycle, their leaf color, and their optimal values for parameters such as temperature, humidity, luminosity, or watering frequency.

- **Browse comments on a plant's page –** On each plant page, there is a section regarding the comments and tips left by Users about the growth of this plant. The tips should appear in an order defined by other user's upvotes and downvotes, and users with more experience should have more relevance in the discussion box. Non-registered users should be able to check these comments too.

The use cases available for both **premium and non-premium users** are:

- Manage house plants' inventory and journal package. This package deals with the use cases related with the plant inventory of the user:
    - **Add plant to house inventory** – Registered users should be able to associate their house plants to their account, and these plants should appear in the home screen once they log in, the so-called "user inventory". In this page, users should see, at a glance, the current condition of the plant (evaluated via a traffic light system), their names and their photos.
    - **Remove plant from house inventory –** Users should be able to remove a previously registered plant at any point in time.
    - **Edit plant in the house inventory** – Users should be able to edit the information about the plants they register, such as their name or their photo.
    - **Update the plant's growth journal –** Users should be able to add new entries to their plant's journals. This could include, for instance, a photo for the plant on that day, and a little text description about the current condition.
- Interact with the community on a plant's page package – This package deals with a proposed forum feature for the application, where users can share tips about the plants they've grown.
    - **Comment on a plant page –** Users should be able to comment on the page of a specific plant. These comments should be about tips for growing each type of plant.
    - **Remove own comment from a plant's page –** A user can remove a comment they have previously posted.
    - **Edit own comment from a plant's page –** A user can edit a comment they have previously posted.
    - **Upvote/Downvote a comment on a plant's page –** Users are free to upvote or downvote comments made by other users on a plant page.
- Manage personal "to do" list package, related with the tasks for plant maintenance:
    - **Browse personal "to do" list"** – Users can browse the tasks they have yet to complete related to the plants they own. These tasks should be automatically created by the service and must belong to different types supported by the application, such has watering tasks, tasks to add fertilizer, or general tasks to check a plant's current condition, by evaluating its leaf colors or appearance. Ideally, these tasks should be presented in both a calendar and a list form.
    - **Tick task from "to do" list –** Once a user has completed a task, they should be able to tick them from the to do list. The app should automatically schedule a new task of the same type, based on the plant's details.
    - **Change the task rescheduling from "automatic" to "manual" –** If users wish, they should be able to change the status of a given task for a

plant from being automatically rescheduled by the app, using an algorithm based on the plant's condition and characteristics, to manually dictating the time frequency between new instances of the task appearing on their calendar. Users should also be able to revert this decision.
    - **Set a new date for a task –** Whether a task is automatically scheduled by the application or not, users should be able to change the deadline date to complete a task at will.
- Manage house divisions package, relating to the management of the divisions in which a user has plants:
    - **Add division to house inventory –** Users should be able to register their divisions at will in their inventory.
    - **Remove division from house inventory –** Users should be able to remove a division from their profile.
    - **Edit division in the house inventory –** Users should be able to edit the information they have about a division, such as its name, or its luminosity.
    - **Associate a plant with a division** – Users should be able to dictate in which division each plant is located. They should also be able to check this information immediately on the app.

The use cases available **only for premium users** are:

- Manage a plant or division's sensor package, related with all the functionalities associated with the plant monitorization by sensors:
    - **Add a sensor –** Users should be able to add a new sensor to their profile. The app should support two types of sensors, namely, plant sensors (which will directly measure parameters on a plant's vase, such as their soil humidity) and division sensors (which will measure the overall parameters in a division shared by plants, such as the air humidity or temperature).
    - **Remove a sensor –** Users should be able to remove associated sensor at will, and this shouldn't disrupt the flow of the application.
    - **Edit a sensor –** Users can edit the information about a sensor, such as its name and the division or plant it's associated with.
    - **Check measurements for a sensor –** Users should be able to read the latest measurements of a sensor, as well as see graphs about the measurements in the past 3 days. These measurements should be available on the sensors tab, but also be present in the page of the related plant. If a plant is in a division with sensors associated to it, users should also see the parameters measured on this division by the sensors.

Finally, the use cases for the system administrator are:

- **Make CRUD operations on the app's database –** System administrators use the admin platform to perform CRUD operations on the plant catalogue database, such as adding new species, or editing the information of existing species.
- **Browse the system's statistics –** Admins can check the operational statistics of the platform.

3.1.4. Further Functional Requirements

Following the definition of the use cases for each actor, and from them, the functionalities to be implemented on the system, we then detailed some extra functional requirements. These requirements helped to specify and clarify some of the main considerations to be kept on mind during the implementation of the described use cases, both on the frontend of the service, as well as in the business logic in the backend.

- **Authentication and Authorization Level**
- Users can be of three types: without account, with non-premium account, with premium account.
- Different types of users should have different types of access to the application's functionality, and this should be made clear along the process.
- Non-registered users can still access the plant catalogue and check its information.
- Functionalities related with sensor monitorization should only be reserved for premium users.
- Premium users should be identified by a golden leaf icon, present not only in their profile, but also next to their name in public pages.
- **Administrative Functions**
- System administrators should be able to check the admin dashboard and see some of the main operational statistics of the platform.
- Admins should be able to edit the app plant's catalogue. They should therefore have the option to include new species, and detail their information, via the admin dashboard, in a easy to use manner.
- **Historical Data**
- The completion of tasks by users should be recorded in activity logs, kept on their profile.
- Sensor's data should be recorded on the persistent database and stored, being used to create graphs for the sensor's measurements over the past few days.
- **Reporting**
- Users should be able to report bugs in the GitHub of the project.
- **Business Rules**
- The application, in the case of premium profiles, should support the measurement of the following parameters using sensors: air temperature and humidity for division sensors; soil humidity for plant sensors.
- Users should be able to associate various sensors to their account, connected to a single server.
- The system should suggest the optimal conditions for each plant, based on their species, on the following parameters: luminosity, humidity, temperature, and watering frequency. These should be evaluated using a scale perfected for each parameter, and the values should be based on proven bibliography.
- Users should be able to check their plant's current condition immediately, using a simple "traffic light" system. The condition should be calculated via an algorithm that considers how many tasks are overdue or close to being completed for the plant and considering the sensors measurements for a premium account. In this case, the measures should be compared against the stored optimum conditions for the plant's species.
- Other suggestions included should be the addition of extra nutrients, or the expected leaf colors for the plant.

- The automatic scheduling of tasks should consider the optimal parameters of the plant.
- The system should suggest new plants for a user to grow. The plant's suggested should be calculated via an algorithm that considers the user's experience (based on their own evaluation, their inventory of plants, and the difficulty of the plants on their inventory), and the calculated difficulty to grow a species. Thus, the definition of the plant species on the system should consider their calculated difficulty, while the definition of the user profile should consider their calculated experience rating.
- The app should suggest the ideal location for a plant within the house based on the optimal luminosity condition of the plant.

3.1.5. Non-Functional Requirements

Besides the functional requirements described, the design of the system should also consider the description of some non-functional requirements, that allow the system to be planned and delivered as intended. Among the most important non-functional requirements considered are:

- **Usability -** The application should be developed bearing in mind that the target audience is comprised of people from all age groups with varied technological backgrounds. Thus, and considering the app will be developed for mobile devices, its interface should be easy to learn and intuitive, and be validated with usability tests with users representing this varied background.
- **Reliability -** The different subsystems of the architecture should be well connected and reliable. When system errors happen, the application should be responsive, so the user realizes what is happening and doesn't abandon the system.
- **Scalability -** The user should be able to connect new sensors to their account whenever needed and associate these sensors with different divisions and plants. Thus, the architecture of the message broker should be implemented with this in mind. The app should be able to accommodate multiple users connecting at the same time.
- **Security -** Given that the application won't deal with sensitive data from its users, the focus on our project will be on the protection of the login information of the users. This will be dealt not only with password encryption and the development of an authentication and authorization system, but also with the implementation of token-based authentication and security methods on the API itself.
- **Interoperability -** The mobile application is required to connect with different sensors used to monitor the plants belonging to the user (temperature and humidity sensors, as well as soil moisture sensors). In the context of the development during this project, the connection will be made via middleware installed on a Raspberry Pi.
- **Efficiency -** Given that mobile devices have limited storage and battery resources, the development of the mobile app should be made in an efficient manner to optimize the system. The Android guidelines for application optimization should be followed.

The application was also developed under the assumption that user's mobile phones must operate on the Android OS, as that was the system in which the implementation was created.

3.2. System Architecture

In this section of the report, we're going to present the general architectural and deployment diagrams and talk about the technologies we used to build our platform. The procedures and implementation details will then be further described on the following chapter.

3.2.1. Domain Model

After concluding the requirements gathering for the definition of the system requirements and its main use cases, we proceeded to create a domain model, using the UML paradigm, in which we expanded and defined the information perspective of the problem, including the main conceptual entities and their most important attributes. This provided a structural representation of the context of the problem, which helped us to conceive and design the system architecture, and to realize how all of the pieces and concepts of our platform are related and interact with each other. It also served, later, as a basis for the object-oriented design of the code implementation, and the creation of the data model used in the system.

Figure 5 presents the final domain model obtained after this process.

As previously decided, the **Users** of the application can either be **Premium Users**, or **Non-Premium Users**, with the main difference between each being that the former has access to all the features regarding the use of sensors for plant's condition monitorization, while the latter has not.

Either way, internally, all Users will have a rating that will be used to measure their experience with the app and their experience growing plants. This rating will be employed as part of the algorithm for the suggestion of new plants for each user to grow.

Users will be able to register in their account all the plants they want to monitor in their house. Thus, the association between User and **Plant** is one-to-many, to represent the inventory of plants of each user. For each Plant, the user can register in which **Division** of their house the pot is located. Besides, each plant will have a set of information related to the details described by the User, such as the plantation date or the size of the pot used.

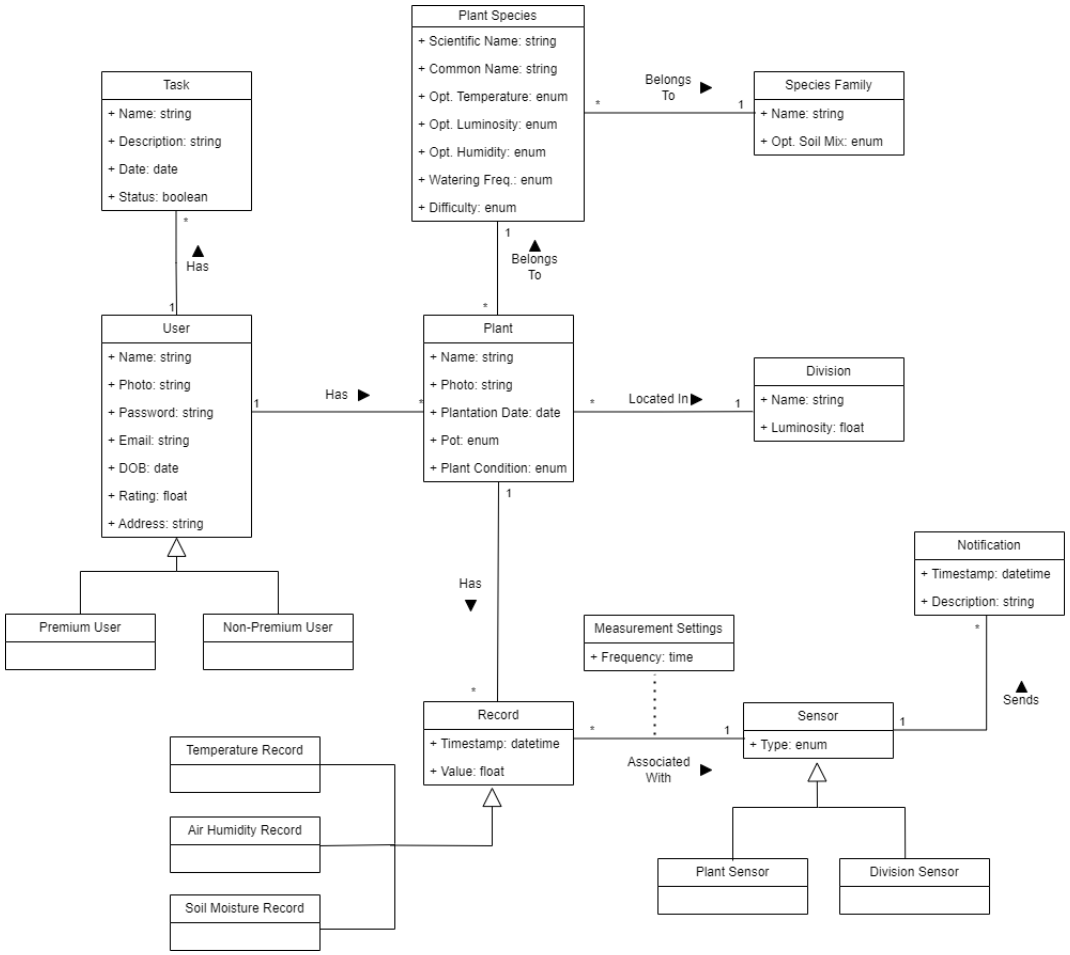


Figure 5 – Domain model, detailing the main conceptual entities of the problem, and some of their attributes. This model was designed based on the UML paradigm.

The system was designed to support a certain catalogue of **Plant Species,** meaning that all the Plants in a user's inventory will be instances of one of these species. For each Plant Species, we will catalog their ideal conditions for growth, which will be registered in the database. These conditions include their optimal temperature, optimal luminosity, optimal humidity, and watering frequency, and each of them will be evaluated based on a scale suggested by the University of Georgia, as presented on Table 3.[8]

Table 3 – Adapted categories for the main conditional parameters monitored on the different Plant Species catalogued by the system.

Monitorization Parameter	Levels in Scale	Levels Description
Light	4	1. **Low-Light Areas:** 25 ft-c to 75 ft-c 2. **Medium-Light Areas:** 75 ft-c to 200 ft-c 3. **High-Light Areas:** Over 200 ft-c, but not direct sun 4. **Sunny Light Areas:** At least 4 hours of direct sun
Temperature	3	1. **Cool:** 10ºC at night; 18 ºC at day 2. **Average:** 18ºC at night; 25 ºC at day

		3. **Warm:** 25ºC at night; 30 ºC at day
Relative Humidity	3	1. **Low:** 5% to 24% 2. **Average:** 25% to 49% 3. **High:** 50% or higher
Watering Frequency	3	1. **Infrequent:** Soil mix can become moderately dry before re-watering 2. **Average:** Surface of soil mix should dry before re-watering 3. **Frequent:** Keep soil mix moist

To further organize the catalogue of plants, we also decided to group the different species into **Species Families**, once again based on the indoor plants categorization made by the University of Georgia. The main factor differentiating these groups are their ideal soil mixes for growth.[8] The Species Families created are described on Table 4.

Table 4 – Plant Species Families present in the catalogue of plants support by the GrowMate platform.

Plant Species Families
Flowering House Plants
Foliage Plants
Bromeliads
Orchids
Succulents and Cacti
Ferns
African Violets and other Gesneriads

The **Tasks** related to the care and maintenance of each Plant will have a status associated to them (either “To be Completed” or “Done”). The application should support the log of all tasks completed by a User.

All the Plants in a user inventory will have an estimated “plant condition” attribute. As previously described, his will be a “traffic lights” system which will allow the User to glance at first sight which plants in their inventory currently need attention (with Red signaling the need for immediate attention, green signaling that the plant is in a healthy state, and yellow signaling an intermediate state between both. For all users, the status of the tasks related to the plant will be used to measure their condition, with overdue tasks having a negative impact on the score. For Premium users, the measurements done by their sensors will also contribute to the value of a plant’s condition.

Premium Users will be able to associate two types of **Sensors** to their account: Either **Plant Sensors**, that measure a parameter that varies from individual plant to individual plant (i.e., their soil moisture); or **Division Sensors**, which monitor parameters that don’t vary from plant to plant but that are dependent on environmental conditions (i.e., the air humidity and temperature). A **Record** for all the measurements will be kept.

3.2.2. Overview of the Architecture

The architecture for the solution was divided into 4 main modules: the Frontend, the Backend, the Storage, and the Sensors architecture. The diagram for the architecture is presented in Figure 6**.**

The **Frontend** of the system consists of a main mobile application, developed using the React Native framework, for the Android OS.

The choice of the React Native Framework for the implementation of this module is due to its component-based development and the ease with which we can use additional libraries for routing and client-side functionality. These characteristics make the framework ideal to implement a well-designed and usable responsive interface for the users in a relatively simple and fast manner, while making the code easy to test, to read and to maintain.

We implemented Async Storage as a client-side cache to store various data, including user session information and static content like plant Categories and Species. This cache is refreshed automatically every 24 hours to ensure the data remains up to date.

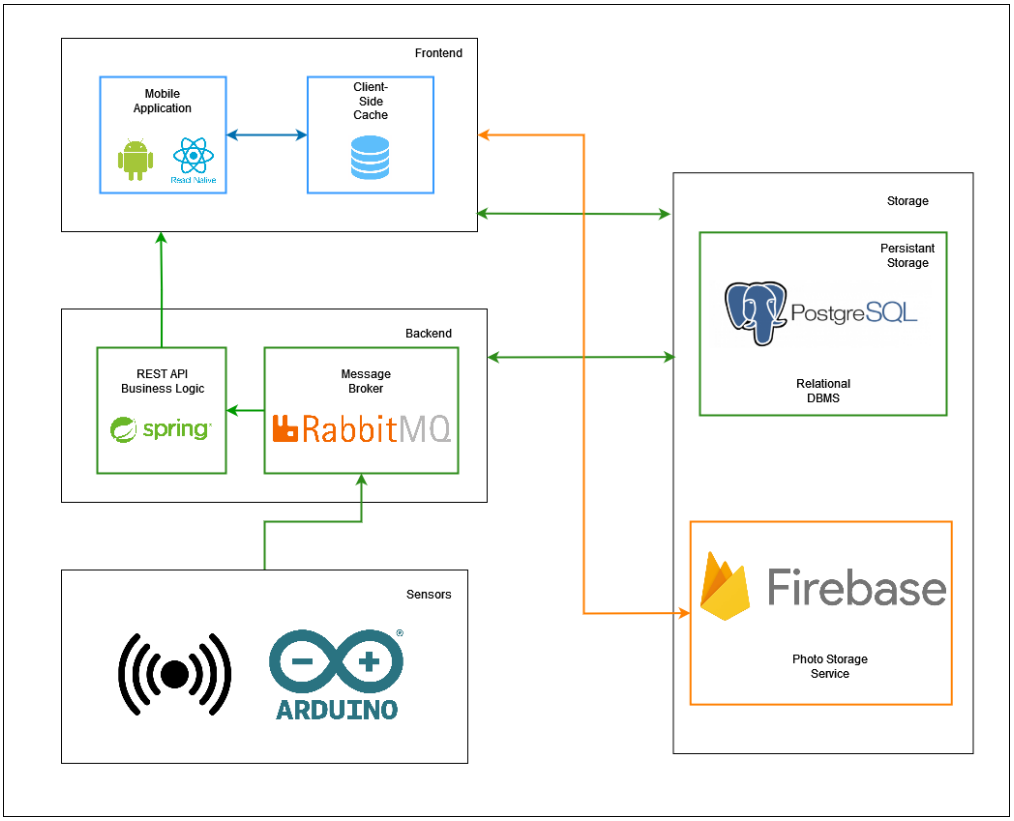


Figure 6 – Diagram for the final architecture of the developed system.

The **backend** of the system consisted of a REST API, that will serve as an interface both the Frontend and the Storage layers while processing business logic, and the Rabbit MQ message queue, which will be used has a broker to receive the measurements data from the sensors and exchange this information with the Storage module.

The REST API will be developed on the Spring Framework, which also allows the easy implementation of a data modelling and processing layer, using Spring Data.

The Spring Framework also allowed the implementation of the required business logic, such as the algorithms to automatically reschedule new dates for a completed task, the algorithm for the suggestion of new plants to the users, the determination of a plant's condition, among others.

For the **storage** module, we decided to use a relational DBMS, namely, Postgres SQL. This allows the implementation of a relational database that will hold the information about the users, their profiles, their inventory of plants, the tasks related to their inventory, etc., as well as the catalogue of the different species supported by the platform, as well as the information characterizing these different species. We decided to use relational databases to model this part of the architecture, since they ensure ACID properties, have high query processing speed, and keep data transactions secure.

The records for the measurements done by each sensor are also stored in this relational database, being received by the backend module on Spring Boot, on classes used to listen to the messages posted on their corresponding RabbitMQ broker topics.

Two types of sensors were employed for data collection. The first sensor utilized was a resistive soil moisture sensor, which accurately measured the humidity levels of the soil. Additionally, we incorporated a DHT11 sensor capable of measuring both temperature and air humidity. To facilitate the data transfer process, we connected these sensors to a nodeMCU, which is equipped with an ESP8266 microchip that includes built-in TCP/IP networking software. This configuration allowed the nodeMCU to establish a connection with the RabbitMQ server and efficiently transmit the collected measurements to their respective queues.

In addition, we employed Firebase as our cloud storage solution for storing photos. By leveraging Firebase's capabilities, we were able to securely upload and store images in the cloud. This allowed us to efficiently manage and retrieve the photos whenever needed, providing a reliable and scalable solution for handling our media assets.

Deviations from the original plan

The final architecture of the system, as seen in Figure 6, has suffered some deviations from the original plan presented at the end of the 2nd milestone. These alterations, and the reason why they were made, are as follows:

- Originally, we intended to use a timeseries database, namely InfluxDB, to store the measurements recorded by the sensors, as we felt like the nature of timeseries database allowed for the simplification of the queries regarding the sensor's measurement records, fitting the data better than a relative DBMS. However, we eventually realized that this solution didn't make much sense for our architecture, as it's usually used only in situations in which there are thousands or more vents being recorded nearly simultaneously. Thus, and considering that the implementation of this timeseries in the Spring framework was becoming quite time-consuming, we decided at the start of the 4th milestone to include the measurements of sensors in the already implemented relational database and focus more of our efforts on the stabilization of the already developed and implemented pages and features.

- Originally, we had planned to implement the sensor architecture using a Raspberry Pi instead of an Arduino controller. However, we faced limitations with the Raspberry Pi as it lacked an analog pin for sensor connection. Consequently, the values obtained from the soil moisture sensor were binary, with zero indicating the absence of moisture and one indicating its presence. Unfortunately, this binary representation proved insufficient for integrating with the algorithms we intended to develop. To overcome this constraint, we made the decision to switch to the NodeMCU module, which features an analog pin. By utilizing the NodeMCU module, we were able to obtain more precise analog readings for soil moisture. Implementing this change was successful, and we encountered minimal challenges along the way, solidifying our choice of utilizing the NodeMCU module for the sensor architecture.
- Instead of using Redis to create the user session cache, we decided to implement the native React framework cache, as this was less time-consuming and easier to use.
- On the original plan for the architecture, we hadn't included the use of the Firebase service for photos storage.

3.2.3. Overview of the Deployment

The deployment architecture consisted of three main components, each represented by different colors in Figure 6. The containers deployed within an Azure Virtual Machine were depicted as green boxes. These containers included a Spring Boot container, RabbitMQ, and a PostgreSQL database. To enable effective communication, the Azure Virtual Machine was customized to expose ports 80 for the API and 1883 for the MQTT RabbitMQ service. This customization allowed seamless interaction between the components and offered scalability, flexibility, and simplified management through Azure VM infrastructure. Furthermore, the Azure VM provided robust security measures and comprehensive monitoring capabilities, ensuring the stability and reliability of the deployed containers.

The Firebase cloud service, represented by the orange box, served as a reliable and scalable backend photo storage service for the application.

The front-end application, depicted by the blue boxes, was an Android APK developed using Expo Application Services. Expo simplified the deployment process by generating the build from the React Native application we developed.

4. Implementation and Procedures

In this chapter, we will describe the methods and procedures used to implement the architectural design presented in the previous chapter, to make the identified system functionalities and requirements come to life. We will analyze each of the systems that compose the architecture of the solution, as well as the technologies employed in their design, one by one. We will then give a small summary of the project management techniques employed during the development of the project.

4.1. Storage Module

4.1.1. Employed Technologies

PostgreSQL

As previously mentioned, the storage module of the solution was mainly implemented using a relational SQL database, with PostgreSQL being the DBMS used for this implementation.

This database holds the information regarding the User accounts and their profiles, such as the inventory of plants for each account, the tasks related to each plant, the divisions a person has in their household, the sensors employed to measure parameters in the divisions and/or sensors, among others.

Since this sort of information is highly structured and well-modelled, the decision to use a relational database system was very clear to us from the beginning, as these systems are known for their ability to handle structured data efficiently and in a reliable manner, through the definition of constraints and relationships between different tables. This is especially true due to their inherent ability to normalize data, preventing redundancy and increasing the integrity of data, and for their support of the ACID (Atomicity, Consistency, Isolation and Durability) properties while holding transactions, making sure that these are not only secure, but also have a high query processing speed.[9]

PostgreSQL is known to ensure data quality, accuracy, and reliability, and to also allow scalability, in scenarios where multiple users or applications interact with the database concurrently, which would be the case for the GrowMate application once it's launched to the public. Taken its robustness and scalability in consideration and considering that PostgreSQL is an open-source solution that is easily integrated with projects in the Spring framework, this was the reason why we chose to use this DBMS on this project.[10]

Firebase

Firebase is a comprehensive app development platform provided by Google, offering a wide range of backend cloud computing services. One of its key features is the NoSQL database, which we utilized in our project to store user-uploaded photos. This

database stores data as objects or documents and provides efficient access through key-value pairs.

Firebase possesses various qualities that make it an excellent choice for our needs. It includes continuous performance monitoring, ensuring that any malfunctions or issues are promptly detected and notified. Moreover, Firebase is highly scalable, allowing for future expansion as our app grows.

4.1.2. Database Implementation and Procedures

Database Modelling

Before we started the implementation of the storage module of the system and considering that we had settled on the relational model, we decided to create an adapted Entity-Relationship Diagram (ER) for the database. Through this, we designed the logical data model, in which we represented the entities and their attributes as tables, and the different relationships between these entities/tables, also considering the cardinality of the relationships. For this, we use the Chen notation.

The creation of this diagram allowed us to figure out the correct database design, and to understand more deeply which entities we would have to create in our data model, as well as how these entities are related to each other, and how that would have to be taken into consideration on the database mapping and design. It also allowed us to normalize the data, and to prevent unnecessary data duplication. The final diagram that resulted in the data model implemented on the system is shown in Appendix C – Database Model Diagram.

The database model created closely resembles the domain model presented in Figure 5 of Chapter 3.2.1**,** being the result of the adaptation of its entities and relationships into the relational database paradigm.

In this way, we can see that the catalogue of plants will be supported by the Plant Species entity, belonging to a Species Family entity. As part of the attributes of the former, are their optimal parameters, such as Optimal Temperature, Luminosity, Humidity and Watering Frequency, which are determined via the scale obtained from the University of Georgia, as presented in Chapter 3.2.1**,** and described in this model via the notes associated with the entity**.**[8]

We can also check that the User entity has an attribute for User_Type, which will determine whether they are Admins, Non-Premium Users or Premium Users, and thus the functionalities to which they have access to. The User entity has an association with the Plant entity, to model the connection between a user and their plant inventory. Each Plant instance will have a condition associated with it, as mentioned before.

Plants can also be associated with the Divisions they are in, and the User can manage the status of a plant's localization in their house. Premium Users can also associate Sensors to either Plants or Divisions, as previously modelled, with the measurements being saved on tables representing the support sensors type – one for the soil humidity, one for the air humidity and one for the air temperature. The ID of the sensors associated with the measure, as well as the Division/Plant ID, will allow the identification of the measurements relating to each user.

The main novelty presented on the database model, when comparing with the expectations set by the domain model, is the modelling of the tasks.

As previously stated, the application will be able to automatically manage the scheduling of tasks for each individual plant, taking into consideration the species they belong to, and the optimal parameters for said species. However, we didn't want to force the user to abide by the automatic scheduling of tasks by our algorithms, if they didn't want to, or if they felt like they have enough gardening knowledge to figure out for themselves when they should do each type of task. Thus, we modelled the task entities in a way that allows users to define whether a specific task is automatically rescheduled, or manually rescheduled, at the user's will.

To do so, we first defined the type of tasks supported by the application, considering the main requirements and functionalities presented in the previous chapter. Given that the development of this project is taken as a proof of concept for what the real published final application would be, we decided to keep the supported tasks to the central and most required functionalities based on the questionnaires and interviews we conducted during the requirements gathering stage:

- Watering of plants, based on the optimal watering frequency of the plant's species and the current season.
- Soil mix changes, based on the species' ideal soil mix and its characteristics.
- Overall checking of the conditions of plants (characteristics such as the leaf color, the dryness of the soil, among others)
- Fertilization of the plant's soil

We then decided to create 3 tables to model the tasks: Current_Tasks, Task_History and Task_Settings. For each plant, a task of the supported type will always be active with a defined due date, which is modelled by the Current_Tasks entity. Once the user completes a Current_Task, this is registered for logging purposes in the Task_History table, and a new Current_Task of the same type with a new due date is created. However, for each plant and type of task, the user can define if they want to reschedule the new dates automatically or manually for the task, with this being modeled by the Task_Settings table. Thus, if the task is set to automatic rescheduling, the GrowMate's algorithms will calculate the ideal task frequency based on the species' characteristics, the plant's condition, and the current season; meanwhile, if the task is set to manual rescheduling, the user is able to set the task frequency at will and change it whenever they want.

Database Implementation and Population

The database wasn't directly implemented on PostgreSQL using a SQL script; instead of this, we created the database by mapping the corresponding entities on the Spring framework, using Spring Data JPA and Hibernate as the object-relational mapping (ORM) framework. After creating the entities on Spring, and using the Spring Data annotations and conventions, we ran the project using the Hibernate configuration for data definition language (DDL) creation, meaning that Hibernate fully mapped the Spring Data entities and their associations into the corresponding PostgreSQL database. This process is fully explored in Chapter 4.2.2. After this, we created a SQL dump script that saved the state of the database, and we placed this script on the Docker volume used to deal with the database's persistence, meaning that the structure of the database, as well as the data it stores, is always persisted and

maintained on the volume, even if the database container is stopped, restarted or removed.

To populate the database, we first started by exploring different catalogues and directories of indoor plant species and their characteristics, to create a rich and varied catalogue of species being able to be supported by our system. We needed to use only species for which we can confirm their optimal maintenance parameters, as well as other required attributes such as the modelled species family they belong to, or their optimal soil mix. This because these attributes are absolutely required for the use of our application, and thus, if we couldn't confirm a species characteristic, we couldn't offer support for the algorithms employed in the platform, such as the tasks rescheduling, or the estimation of the individual plant's current condition. Thus, we primarily used the catalogue of indoor plants maintained by the University of Georgia[8], as well as Perenual's Plant API[11] and Trefle's API[12] to populate the database. The information in these catalogues was cross-referenced with other sources, such as North Carolina's Extension Gardener Plant Toolbox[13], or Universidade de Trás-os-Montes e Alto Douro's botanical guides[14], to ensure that the information was correct, and to supplement the catalogue with further popular indoor species.[15–41]

Considering that this project served as proof of concept, we decided that a catalogue of around 200 species would suffice.

Besides populating the catalogue of supported species, we also created some simulated users, with a plant inventory associated to them, as well as tasks and sensors, accompanied by historical sensor's measurements values. This was done so that we could properly develop and test the application and showcase the use cases designed for long-time users of the application. Once again, this information is persisted using the Docker volume defined for the database container.

Further Scripts for Database Management

Besides the previously mentioned procedures for the database creation and population, some other scripts were used to manage the database directly.

For instance, secondary indexes were created on some of the tables, to optimize query performance and do efficient data retrieval on some of the most requested attributes. These include, for example, the common_name and scientific_name attributes of the Plant Species table, as these attributes are widely used for search and sorting purposes; or the date attributes in the Current_Tasks table.

No views or explicit join tables were created in the PostgreSQL database, as it wasn't necessary to do so. This is because Spring Data's Hibernate handles the necessary optimized join operations when querying related entities, using the information from the entity mappings and relationships defined in the Spring Data entities as a basis for these queries, including the creation of the necessary join tables to support many-to-many relationships.[42]

We did, however, create a few triggers to better manage the entities saved in the database, in situations in which it felt more appropriate to do logic on the database side, instead of in the Spring project.

For instance, as visible on the database model diagram available on Appendix C, each plant species has an estimated difficulty associated with it. This difficulty is used, for example, in a plant suggestions algorithm, which looks at a user's overall experience with plants, and suggests new plants for them to grow, based on their experience.

To calculate the difficulty of growing a species, we created a trigger on the database that is activated when a new species is inserted onto the Plant Species table. This trigger considers the optimal requirements for growing the plant, with species having higher necessities for watering frequency, or for more extreme intervals of temperatures, being rated higher. This trigger also considers the plant family a species belongs to. The code snippet for this trigger is presented on Figure 7.

```
CREATE OR REPLACE FUNCTION add_plant_difficulty()
RETURNS TRIGGER AS $$
DECLARE
  plant_difficulty INTEGER;
BEGIN

  SELECT public.species_family.difficulty
  INTO plant_difficulty
  FROM public.species_family
  WHERE public.species_family.id = NEW.family_id;

  NEW.difficulty = ROUND((
      (ABS(sin((pi()/2)*NEW.optimal_temperature))*5.0)* 0.1 +
      (5.0/3.0) * (NEW.optimal_luminosity-1) * 0.2 +
      (5.0/2.0) * (NEW.optimal_humidity-1) * 0.2 +
      (5.0/2.0) * (NEW.watering_frequency-1) * 0.35 +
      (plant_difficulty-1) * 0.15
   )+0.5);

    RAISE NOTICE 'Temp: %', (ABS(sin((pi()/2)*NEW.optimal_temperature))*5.0);
    RAISE NOTICE 'Lum: %', (4.0/3.0) * (NEW.optimal_luminosity-1);
    RAISE NOTICE 'Hum: %', (4.0/2.0) * (NEW.optimal_humidity-1);
    RAISE NOTICE 'Wfreq: %', (4.0/2.0) * (NEW.watering_frequency-1);
    RAISE NOTICE 'FamDiff: %', plant_difficulty;
    RAISE NOTICE 'Exact value: %', ((ABS(sin((pi()/2)*NEW.optimal_temperature))*5.0)* 0.1 +
      (4.0/3.0) * (NEW.optimal_luminosity-1) * 0.2 +
      (4.0/2.0) * (NEW.optimal_humidity-1) * 0.2 +
      (4.0/2.0) * (NEW.watering_frequency-1) * 0.35 +
      (plant_difficulty-1) * 0.15) +0.5;
    RAISE NOTICE 'Result: %', NEW.difficulty;


  RETURN NEW;

END
$$
```

Figure 7 – SQL trigger for calculating the difficulty for growing a newly implemented plant species on the database.

4.2. Backend Module

4.2.1. Employed Technologies

Spring Framework

The Spring framework is a popular open-source Java-based application framework that provides comprehensive support for developing enterprise-grade applications. It offers a lightweight and modular approach to building software systems, emphasizing flexibility, scalability, and ease of testing. The Spring Framework has many core features and modules that allow developers to create applications in a very code efficient and lightweight manner, removing and time-consuming configuration work so that developers can focus on writing the business logic of their services.[43]

To do so, Spring web applications are usually built on the typical three-layered architecture involving the view layer, the business logic, and the data access layer, applying dependency injection so that the different classes required for each layer become loosely coupled and easier to manage and to change. Using Spring's dependency injection, together with its inversion of control mechanisms, to instantiate objects and populate dependencies, developers only need to tell Spring which objects to manage, and what dependencies exist for each class using annotations, making the code more modular, reusable, and easier to maintain.[43,44]

Besides this, other advantageous paradigms supported by the Spring framework include its Aspect-Oriented Programming, that allows the separation of concerns such as logging or transactions management from the business logic, making the code become cleaner and more modular. Spring also allows the integration of various technologies, provide abstraction layers that facilitate their programming, with this being applied, for example, in database access, transaction management or messaging.[43]

The Spring framework is very well suited to the development of RESTful APIs, through the Spring Web module, that includes features like request mapping, content negotiation, exception handling and JSON serialization.

Thus, considering all the advantages provided by this framework that are especially suited to our project, and that we already had some experience using this framework, we decided to use Spring to apply all the necessary backend logic needed in our application, including not only the REST API, but also the business logic, the data access logic to interact with the database, and the logic required to integrate the measurements done by sensors and posted on the RabbitMQ queues.

To create and configure the project, we used the Spring Boot module, which follows the principles of convention-over-configuration to automatically configure the necessary components and necessary dependencies, as well as their management, for the project.[45] Spring Boot also provides an embedded servlet container that eliminates the need for deploying the application in external servers, allowing the running of the RESTful application has a standalone executable JAR.[45] The starter dependencies, which include pre-packaged libraries for common functionalities, included in the project, where:

- Spring Web – Used for the creation of RESTful applications, using Apache Tomcat as the default embedded servlet.

- Spring Boot DevTools – Provides fast application restarts, LiveReload, and other configurations to enhance the developer experience.
- Spring for RabbitMQ – Gives the application a common platform to send and receive messages via RabbitMQ topics.
- Spring Data JPA – Allows the persistence of data in SQL relational databases, using the Java Persistence API and Hibernate to manage the data access layer.
- PostgreSQL Driver – A driver that allows the connection of Java programs to a PostgreSQL database, using standard and database independent JAVA code.

RabbitMQ

RabbitMQ is an open-source message broker that facilitates communication and coordination between distributed systems, providing a reliable and scalable messaging infrastructure. To do this, RabbitMQ employs the Advanced Message Queuing Protocol (AQMP) that acts as a mediator between the senders (also known as producers) and the receivers (also known as consumers) of the messages, through a flexible and decoupled communication channel that ensures the reliable delivery of messages, message ordering and message persistence.[46]

It employs the publisher-subscriber design pattern, with producers sensing messages to named and well-defined exchanges, while the consumers bind to the queues relevant to their activity.[46]

In the case of our application, RabbitMQ was employed in the communication between the sensors and the backend services, with sensors publishing their measurements to a relevant RabbtiMQ exchange topic, that was delivered to the backend module on the Spring framework.

4.2.2. Data Access Layer

As explained on Chapter 4.1.2., Spring Data JPA, a part of the Spring framework, was used to map the database onto entities, to generate the database schema for PostgreSQL and to create the data access layer on the backend module of the application.

Spring Data is an abstraction over the traditional JDBC (Java Database Connectivity) approach, used to significantly reduce the amount of boilerplate code required to implement the data access layers for the persistence stores. It uses the Jakarta Persistence API (JPA) to map Java objects onto relational databases, which includes specifications for entity and association mappings, for the entity lifecycle management, and for the querying of databases. The mapping of Java objects to database tables and vice-versa is called object-relational mapping (ORM), with Hibernate being the default JPA provider configured by Spring Boot, and thus being used in this project.[47]

As such, to do the ORM process, each of the entities identified in the diagram model present in Appendix C, was instantiated as a class on the Spring framework, using the @Entity annotation. These entities are essentially POJOs (Plain Old Java Objects) representing the data that can be persisted to the database tables.

The use of Spring Data also allows the direct mapping of relationships and associations between entities, being able to map one-to-one, many-to-one and many-to-many relationships, as well as inheritance and composition relationships in a very efficient and intuitive manner and allowing the manipulation of the data in such relationships in a traditional Java way. Figure 8 provides an example of the definition of an Entity involved in multiple many-to-one associations, in this case, the Plant entity, mapping the table representing the instances of Plants present in user's inventories in the database.

```java
@Entity
@Table
@Getter
@Setter
@NoArgsConstructor
@AllArgsConstructor
@ToString
public class Plant {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;

    @Column(nullable = false)
    private String name;

    @Column(name = "photo")
    private String plantPhoto;

    @Column(name = "plantation_Date")
    @Temporal(TemporalType.DATE)
    private Date plantationDate;

    @Convert(converter = PlantConditionConverter.class)
    private PlantCondition plantCondition;

    @JsonIgnore
    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name ="user_id", nullable = false)
    @ToString.Exclude
    private User owner;

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name ="species_id", nullable = false)
    @ToString.Exclude
    private PlantSpecies species;

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name ="division_id")
    @ToString.Exclude
    private Division division;

    @JsonIgnore
    @OneToMany(mappedBy = "plant", cascade = CascadeType.ALL)
    @ToString.Exclude
    private List<PlantSensor> sensors = new ArrayList<>();

    @JsonIgnore
    @OneToMany(mappedBy = "plant", cascade = CascadeType.ALL)
    @ToString.Exclude
```

Figure 8 - Definition of the Plant Entity as a POJO, using the @Entity annotation, and representing the Plant table of the PostgreSQL database.
Note the use of @Column annotations to indicate that attributes from the POJO Java class correspond to columns of the Plant table on the database. Also worth pointing out is the mapping of many-to-one associations in which Plant is involved – for instance, each User can have multiple Plants, represented by the @ManyToOne annotation in the owner attribute. On the other side of this association, in the User POJO, there exists a @OneToMany annotation representing with the list of Plants the user owns.

The main interface between Spring's data access layer and the database itself, that allows the interaction between the backend services and the database, is given by the repositories marked with the @Repository annotation, that allows the retrieval and saving of instances of a particular JPA entity on the database, serving as an abstraction for the definition of data access operations. The use of repositories also allows the automatic generation of the queries needed to be made based on the entity definitions,

with queries being directly linked to the method names created in the repository. Thus, repositories were created for each of the entities that needed to be retrieved or saved from the database. An example of a repository is shown in Figure 9, namely, the repository representing the interface with the Air Humidity Measurements table in the database.

```
public interface AirQualityRepository extends JpaRepository<AirQualityMeasurement, Long> {
    2 usages  Daniel Carvalho
    AirQualityMeasurement findFirstBySensorOrderByPostDateDesc(DivisionSensor sensor);
    1 usage  Daniel Carvalho
    List<AirQualityMeasurement> findAllByPostDateAfterAndSensorOrderByPostDateDesc(LocalDateTime date, DivisionSensor sensor);
}
```

Figure 9 - The AirQualityRepository represents the interface with the Air Quality Measurements table in the PostgresSQL database, being associated with the AirQualityMeasurement POJO entity on the Spring framework. The retrieval and saving of data operations are done by methods automatically created on the definition of the @Repository entity, associated with the JpaRepository interface, and that don't need to be declared. However, the use of personalized methods to represent other queries to the database do need to be declared, as is the case of the methods presented on this Figure. findFirstBySensorOrderByPostDateDesc(), for example, as the name implies, finds the first AirQualityMeasurement of the table, when ordered by the measurement Post Date, associated with a given specific Sensor. This association is possible due to mapping of the relationship between AirQualityMeasurement and DivisionSensor in their respective POJOs.

Other advantages gained using Spring Data include features like lazy loading, caching, and transaction management, which help optimize the performance and consistency of database operations. For instance, the transaction management provided by Spring Data JPA ensures the integrity of data, by grouping database operations into atomic units, and enabling rollbacks in the case of failures.[48]

It's also worth noting that Hibernate, the previously mentioned underlying ORM framework used by Spring Data JPA, offers features such as entity queries or entity projections, that can be used to optimize database access, fetching only the required fields from the database instead of loading complete entity objects, and thus reducing the amount of data transferred between the database and the application, vastly improving query performance.[49] Hibernate also provides another feature called "second-level caching", that consists on the caching of queries results to avoid repeatedly hitting the database for frequently accessed data.[50]

Another aspect of the data modelling practices in the backend implementation that is worth pointing out is how we chose to deal with the various attributes and parameters that belonged to a defined range of possible values. These include, as shown on the database model present in Appendix C, the following parameters:

- The optimal luminosity, optimal temperature, optimal humidity, and watering frequency attributes of a Plant Species, that are defined by to which each level of the possible condition is defined by the values presented on Table 3 of Chapter 3.2.1.
- The Task Types, which belong to one of the identified types presented in Chapter 4.1.2.
- The User Type, which is either 0 for admins, 1 for Non-Premium Users, and 2 for Premium Users.

The solution implemented was the modelling of these attributes as enums in the Spring framework backend, used in the various methods of the business logic that needed to access these parameters, and their definition as numerical attributes in the corresponding PostgreSQL tables. To convert from the numerical attributes

representation to the enum representation, and vice-versa, when interactions between the data access layer of the backend and the database happened, we used the @Converter annotation of the Spring framework, as well as Jakarta's Attribute Converter interface, which allowed us to implement customized conversion logic suited for our needs. As an example, Figure 10 shows the definition of the Optimal Luminosity enum on the Spring framework project, while Figure 11 shows the Luminosity Converter that converts values of this attribute from the enum representation to the integer representation needed to interact with the database.

```
public enum OptimalLuminosity {
    4 usages
    LOW, MEDIUM, HIGH, SUNNY
}
```

Figure 10 – Optimal Luminosity defined as a simple Java enum.

```
@Converter
public class LuminosityConverter implements AttributeConverter<OptimalLuminosity, Integer> {
    no usages  Daniel Carvalho +1
    @Override
    public Integer convertToDatabaseColumn(OptimalLuminosity optimalLuminosity) {
        return switch (optimalLuminosity) {
            case SUNNY -> 4;
            case HIGH -> 3;
            case MEDIUM -> 2;
            case LOW -> 1;
            default -> throw new IllegalArgumentException("Invalid luminosity converter: " + optimalLuminosity);
        };
    }

    no usages  Daniel Carvalho +1
    @Override
    public OptimalLuminosity convertToEntityAttribute(Integer optimalLuminosityInt) {
        return switch (optimalLuminosityInt) {
            case 1 -> OptimalLuminosity.LOW;
            case 2 -> OptimalLuminosity.MEDIUM;
            case 3 -> OptimalLuminosity.HIGH;
            case 4 -> OptimalLuminosity.SUNNY;
            default -> throw new IllegalArgumentException("Invalid luminosity converter integer: " + optimalLuminosityInt);
        };
    }
}
```

Figure 11 – The Luminosity Converter, responsible for the conversion of Optimal Luminosity values from the enum Java representation to the required numerical representation in the PostgreSQL database.

4.2.3. REST API

The REST API that serves as the backbone for the backend of the application, representing the interface between the Frontend and the Storage layers and business logic of the solution, was also developed with the use of the Spring framework, as previously explained.

The Spring framework follows the architectural pattern of separating concerns into models, views, and controllers, with this pattern being very suited to the handling of HTTP-based requests and responses, central to the REST API activity. The overall structure of the Spring Boot project for the application, as well as its interaction with other submodules of the architecture, is presented in Figure 12.

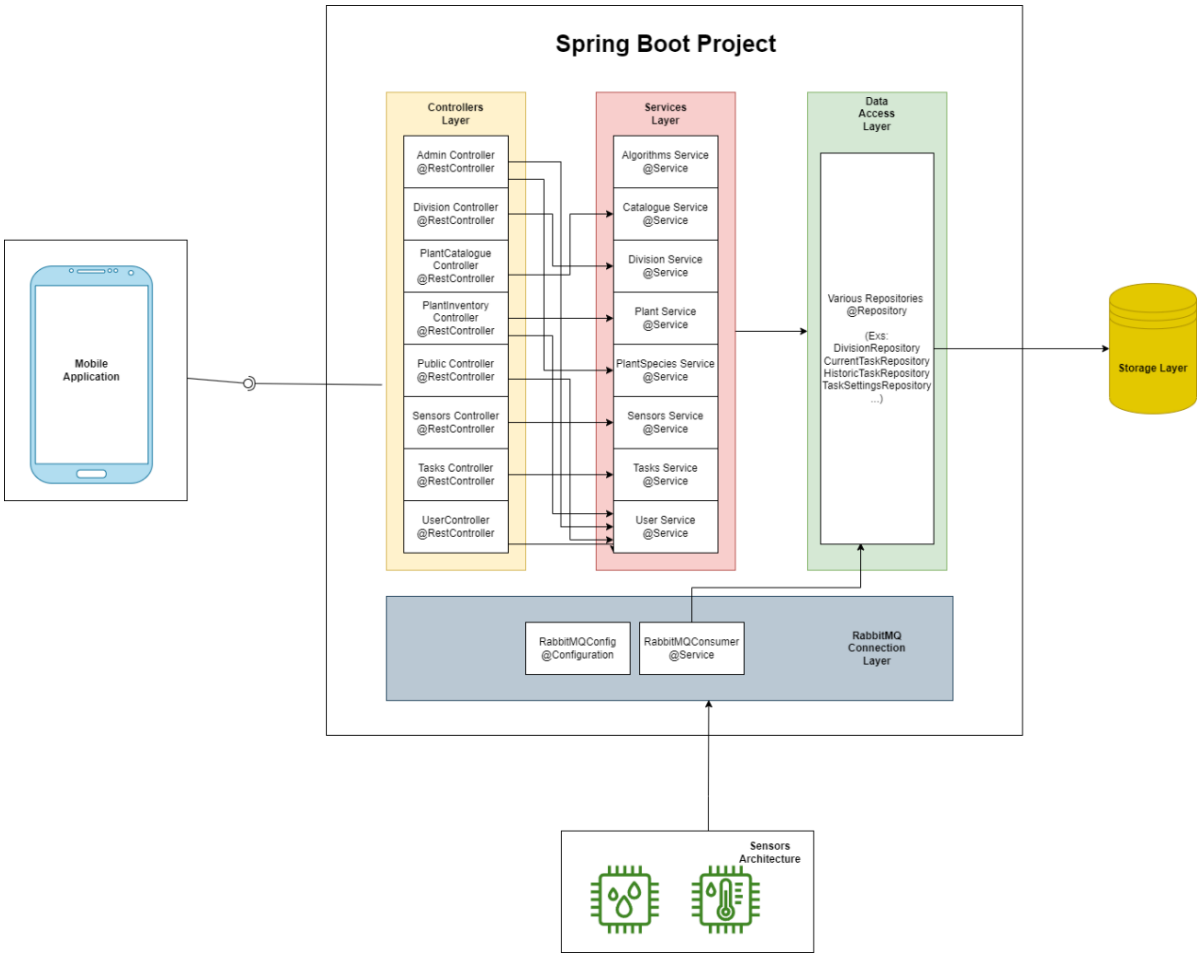


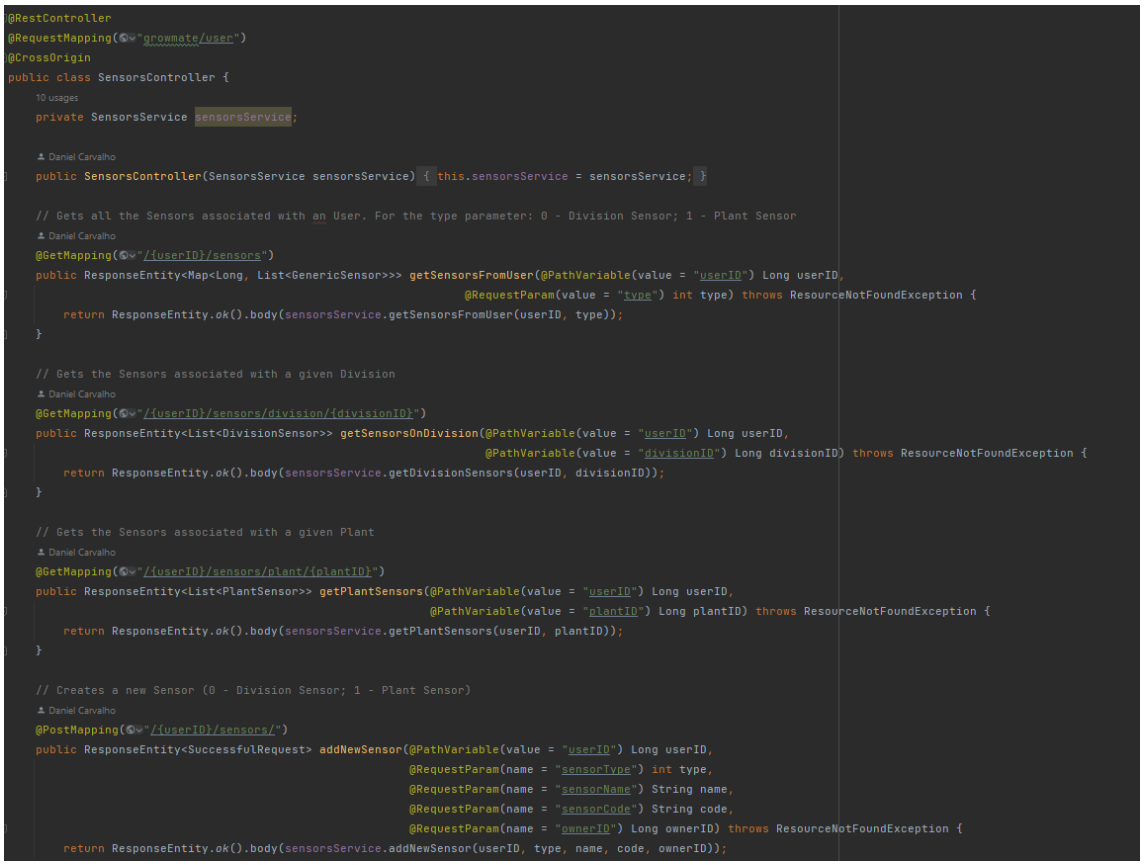
Figure 12 - Diagram of the structure of the Spring Boot project defining the backend, and its interactions with the Storage Layers, the Sensors Layers and the Mobile Application. This diagram further expands on the Controllers Layers, Services Layers and Data Access Layer that constitute the design of the SpringBoot project, as well as the interactions between the different classes in each of these layers. To facilitate the representation, we decided not to present all the Repositories, as there were 13 classes with the @Repository annotation, making the visualization of the diagram much harder to comprehend if they were all represented.

Thus, to implement our REST API, and having already created the data model layer explained in the previous section, we started by creating REST controller classes for the specification of the REST endpoints and their request mappings, using the @RestController annotation. These controllers process the requests, interact with the backend storage layers and business logic, and return the appropriate responses in the form of JSON. We started by identifying the required endpoints to fulfill the use cases, system functionalities and system requirements previously identified, and then grouping the endpoints in logical controllers. The created controllers were as follows:

- Admin Controller – For the methods related to the management of the database catalogue, such as new plant species to the database.
- Division Controller – Manages the house divisions associated with a user's accounts, such as returning all the plants located in a division, changing a plant from one division to another, or adding new divisions.
- PlantCatalogue Controller – Consists of the methods used to return the catalogue of plants present in the app, such as checking all of the species supported, the categories of plants available or the plants in all categories.
- PlantInventory Controller – Is used to manage the inventory of plants of a user.

- Public Controller – Used for publicly available endpoints, such as login methods, or registration of a new account.
- Sensors Controller – Manages the sensors associated with users' accounts, providing methods to return the measurements taken by these sensors.
- Tasks Controller – Groups the endpoints related with task management, such as the methods to get the active tasks for a user, to update the task status, to calculate tasks rescheduling or to change a task's settings.
- User Controller – Houses the endpoints necessary to manage the profile of an user.

The controllers and their place on the Spring Boot project design are shown in Figure 12**,** with all the available endpoints being documented on our API documentation website, available here. Figure 13 presents an example of part of the implementation of the Sensors Controller, showing REST endpoints associated with essential CRUD operation, such as the creation of new sensors, or the obtention of sensors associated with a User, or a given Plant. It's worth noting the use of mappings for the endpoints, and the use of path variables and request parameters to get the necessary parameters from the frontend for the call to be processed.

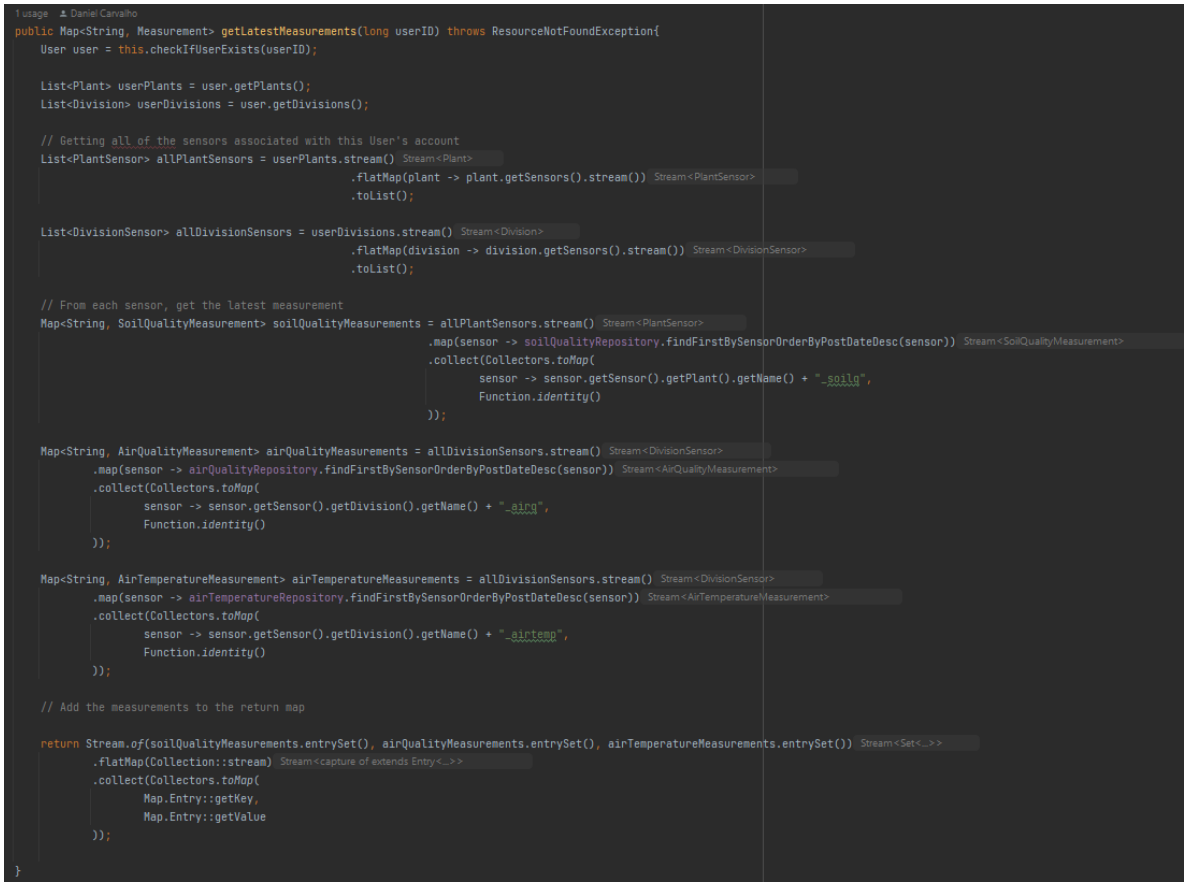


```
@RestController
@RequestMapping("growmate/user")
@CrossOrigin
public class SensorsController {
    10 usages
    private SensorsService sensorsService;

    Daniel Carvalho
    public SensorsController(SensorsService sensorsService) { this.sensorsService = sensorsService; }

    // Gets all the Sensors associated with an User. For the type parameter: 0 - Division Sensor; 1 - Plant Sensor
    Daniel Carvalho
    @GetMapping("/{userID}/sensors")
    public ResponseEntity<Map<Long, List<GenericSensor>>> getSensorsFromUser(@PathVariable(value = "userID") Long userID,
                                                                              @RequestParam(value = "type") int type) throws ResourceNotFoundException {
        return ResponseEntity.ok().body(sensorsService.getSensorsFromUser(userID, type));
    }

    // Gets the Sensors associated with a given Division
    Daniel Carvalho
    @GetMapping("/{userID}/sensors/division/{divisionID}")
    public ResponseEntity<List<DivisionSensor>> getSensorsOnDivision(@PathVariable(value = "userID") Long userID,
                                                                     @PathVariable(value = "divisionID") Long divisionID) throws ResourceNotFoundException {
        return ResponseEntity.ok().body(sensorsService.getDivisionSensors(userID, divisionID));
    }

    // Gets the Sensors associated with a given Plant
    Daniel Carvalho
    @GetMapping("/{userID}/sensors/plant/{plantID}")
    public ResponseEntity<List<PlantSensor>> getPlantSensors(@PathVariable(value = "userID") Long userID,
                                                             @PathVariable(value = "plantID") Long plantID) throws ResourceNotFoundException {
        return ResponseEntity.ok().body(sensorsService.getPlantSensors(userID, plantID));
    }

    // Creates a new Sensor (0 - Division Sensor; 1 - Plant Sensor)
    Daniel Carvalho
    @PostMapping("/{userID}/sensors/")
    public ResponseEntity<SuccessfulRequest> addNewSensor(@PathVariable(value = "userID") Long userID,
                                                          @RequestParam(name = "sensorType") int type,
                                                          @RequestParam(name = "sensorName") String name,
                                                          @RequestParam(name = "sensorCode") String code,
                                                          @RequestParam(name = "ownerID") Long ownerID) throws ResourceNotFoundException {
        return ResponseEntity.ok().body(sensorsService.addNewSensor(userID, type, name, code, ownerID));
```

Figure 13 – Code snippet of the Sensors Controller.

Customized exceptions associated with HTTP status codes, as well as a global exception handler controller, were created to handle situations in which the information passed onto an endpoint was incorrect, or the request couldn't be processed. For instance, if a API request for the retrieval of information related with a Plant is passed using a non-existing ID on the database, a personalized Request Not Found Exception is thrown, returning a HTTP Status code "404 Not Found".

The methods that contain the underlying business logic necessary to process the different API requests, and that also act as an intermediary between the controllers and the repositories of the data access layer, allowing the compliance of the backend with the operations made on the frontend of the application, are present in components annotated with the @Service annotation. The controllers invoke the methods in these components, with the mappings between controllers and services being presented in Figure 12. Presented below, on Figure 14, is a snippet of a method present in the Sensor Service, responsible for the retrieval of the latest measurements from sensors associated with a given user account. Thus, this method processes the necessary business logic required to successfully reply to a request to the GET API endpoint associated with the retrieval of this information.



```
1 usage   Daniel Carvalho
public Map<String, Measurement> getLatestMeasurements(long userID) throws ResourceNotFoundException{
    User user = this.checkIfUserExists(userID);

    List<Plant> userPlants = user.getPlants();
    List<Division> userDivisions = user.getDivisions();

    // Getting all of the sensors associated with this User's account
    List<PlantSensor> allPlantSensors = userPlants.stream() Stream<Plant>
                                            .flatMap(plant -> plant.getSensors().stream()) Stream<PlantSensor>
                                            .toList();

    List<DivisionSensor> allDivisionSensors = userDivisions.stream() Stream<Division>
                                            .flatMap(division -> division.getSensors().stream()) Stream<DivisionSensor>
                                            .toList();

    // From each sensor, get the latest measurement
    Map<String, SoilQualityMeasurement> soilQualityMeasurements = allPlantSensors.stream() Stream<PlantSensor>
                                                    .map(sensor -> soilQualityRepository.findFirstBySensorOrderByPostDateDesc(sensor)) Stream<SoilQualityMeasurement>
                                                    .collect(Collectors.toMap(
                                                            sensor -> sensor.getSensor().getPlant().getName() + "_soilq",
                                                            Function.identity()
                                                    ));

    Map<String, AirQualityMeasurement> airQualityMeasurements = allDivisionSensors.stream() Stream<DivisionSensor>
            .map(sensor -> airQualityRepository.findFirstBySensorOrderByPostDateDesc(sensor)) Stream<AirQualityMeasurement>
            .collect(Collectors.toMap(
                    sensor -> sensor.getSensor().getDivision().getName() + "_airq",
                    Function.identity()
            ));

    Map<String, AirTemperatureMeasurement> airTemperatureMeasurements = allDivisionSensors.stream() Stream<DivisionSensor>
            .map(sensor -> airTemperatureRepository.findFirstBySensorOrderByPostDateDesc(sensor)) Stream<AirTemperatureMeasurement>
            .collect(Collectors.toMap(
                    sensor -> sensor.getSensor().getDivision().getName() + "_airtemp",
                    Function.identity()
            ));

    // Add the measurements to the return map

    return Stream.of(soilQualityMeasurements.entrySet(), airQualityMeasurements.entrySet(), airTemperatureMeasurements.entrySet()) Stream<Set<...>>
            .flatMap(Collection::stream) Stream<capture of extends Entry<...>>
            .collect(Collectors.toMap(
                    Map.Entry::getKey,
                    Map.Entry::getValue
            ));
}
```

Figure 14 – Example of a method present in the Sensors Service.

Besides serving as the code necessary to process the requests made by controllers, the services also house the overall business logic of the application, which includes other methods and algorithms supported by the backend, such as the RabbitMQ consumers or the algorithms related with the use cases and functionalities supported by the application (such as automatic task rescheduling, or determination of a plant's current condition). This business logic side of the services is further explored in the following sections.

4.2.4. Business Logic Algorithms

As stated on the previous section, some algorithms were created in the business logic layer of the backend module, to accommodate the necessary functional requirements and use cases established during the requirements gathering process. Among these algorithms are included:

- An algorithm to automatically reschedule new tasks of a given type once the current task is completed.
- An algorithm to estimate a plant's current condition, based on the number of overdue tasks and the latest sensor measurements related to the plant.
- An algorithm to suggest new plants for a user to grow, based on their experience and their current inventory of plants.

These algorithms were implemented in the Algorithms Service, represented on Figure 12.

Task Creation and Scheduling Algorithms

As previously explained on Chapter 4.1.2, the modelling of the tasks related with the plant's care was done in a way that prioritized the user's preferences and needs – if they want the application to automatically calculate and schedule the tasks, based on the plant's characteristics, they can; however, if they want to manually schedule tasks themselves, they also have the option to do so. In this section, we will explain how the algorithms created on the business logic of the service support this.

As stated, and considering that the development of this project is proof of concept for what the final published application could be, we selected a limited set of task types that represent the most common plant caring tasks. When a plant is added to a user's inventory, a task of each of these types for the plant is automatically created on the backend of the service, with the default setting for these tasks being "automatic". For this, an instance of Task Current and Task Settings previously described on the database modelling chapter is created for each of the task types supported on the app. The method implementing this automatic task creation is called the service method dealing with the Plant Inventory Controller POST endpoint associated with the addition of a plant to a user's inventory, and is shown in Figure 15.

```java
// This method adds new Tasks and new Task Settings for a newly created Plant
1 usage  Daniel Carvalho *
public void addTasksForNewPlant(Plant newPlant) {

    // For each available TaskType, we will create a new Task Settings instance that will store the frequency in which
    // that task will be made, using an algorithm depending on the Task.
    // Besides this, a new instance of Tasks_Current will be created,
    // with the Date calculated based on the task frequency
    for (TaskType type : TaskType.values()) {
        // Create a new Task Settings entity for this Plant and this type of Task
        Task_Settings settings = new Task_Settings();

        settings.setAutomatic(true);            // by default, new Plants will have tasks calculated automatically
        settings.setPlant(newPlant);
        settings.setTaskType(type);

        // Call the algorithm to calculate the frequency of the new Task Type
        settings.setTaskFrequency(calculateNewFrequency(newPlant, type));

        // Save the settings on the Task Settings Repository
        taskSettingsRepository.save(settings);

        // Create a new Tasks_Current entity for this TaskType and this Plant
        Tasks_Current task = new Tasks_Current();

        task.setPlant(newPlant);
        task.setTaskType(type);
        task.setName(createTaskName(newPlant, type));

        // Calculate the new Date for the Task
        calculateNewTaskDateForSingleTask(task);

        // Save the new Task
        currentTaskRepository.save(task);
    }
}
```

Figure 15 - The method responsible for the creation of new Tasks, once a plant is added to the inventory of an user. This method assumes as a default task setting that the task will be automatically rescheduled, and thus calls on the algorithms necessary to determine the plant's task frequency and new dates, based on the plant's characteristics.

Given that the task is automatically rescheduled, we then need to determine the time frequency in which this task should be completed, based on the characteristics and optimal conditions of the species the plant belongs to. To do so, an algorithm was developed which considers the plant species, and the type of task which is being calculated. The following list summarizes the logic behind the frequency calculation for each type of task:

- For the **watering** task, the frequency is determined based on the optimal watering frequency and optimal humidity values determined for the plant, as well as based on the current season at the date the frequency is calculated. The code snippet for this method is presented on Figure 16.
    - If the species' watering frequency is classified as INFREQUENT, then the interval of watering of a plant is defined in between 10 and 20 days. The exact value is decided by the current season and the optimal humidity of a plant - if the user is currently on Fall or Winter, the interval is longer; however, if they are in the summer the watering needs of the plant is obviously higher, and the frequency ends on the lower end of this interval, depending on the optimal humidity of the plant.
    - If the watering frequency is, instead, classified as AVERAGE, the default time interval is set to between 6 and 10 days, with the season and optimal humidity of the species once again determining the exact value returned.

- If the watering frequency is FREQUENT, the default interval is between 3 and 7 days, and exact day is determined based on the previously explained logic.
- The values for these intervals were chosen based on sources for general plant watering tips found while doing research for the project. [8,51]

```
private int calculateNewWateringFrequency(PlantSpecies species) {
    Season currentSeason = getCurrentSeason();
    int wateringFrequency, optimalHumidity;

    switch (species.getWateringFrequency()) {
        case INFREQUENT -> wateringFrequency = 1;
        case AVERAGE -> wateringFrequency = 2;
        case FREQUENT -> wateringFrequency = 3;
        default -> throw new IllegalArgumentException("Invalid watering frequency");
    }
    ;

    switch (species.getOptimalHumidity()) {
        case LOW -> optimalHumidity = 1;
        case MEDIUM -> optimalHumidity = 2;
        case HIGH -> optimalHumidity = 3;
        default -> throw new IllegalArgumentException("Invalid humidity converter");
    }
    ;

    if (wateringFrequency == 1) {
        if (currentSeason.equals(Season.FALL) || currentSeason.equals(Season.WINTER)) {
            return 20;
        } else if (optimalHumidity == 3 || currentSeason.equals(Season.SUMMER)) {
            return 10;
        } else {
            return 14;
        }
    } else if (wateringFrequency == 2) {
        if (currentSeason.equals(Season.FALL) || currentSeason.equals(Season.WINTER)) {
            return 10;
        } else if (optimalHumidity == 3 || currentSeason.equals(Season.SUMMER)) {
            return 6;
        } else {
            return 8;
        }
    } else {
        if (currentSeason.equals(Season.FALL) || currentSeason.equals(Season.WINTER)) {
            return 7;
        } else if (optimalHumidity == 3 || currentSeason.equals(Season.SUMMER)) {
            return 3;
        } else {
            return 5;
        }
    }
```

Figure 16 – Method used to calculate a frequency for tasks related to the watering of a specific plant. This algorithm takes into account the current season, as well as the parameters and characteristics of the plant species.

- For the **fertilizing** task, the algorithm once again takes into consideration the current season of the year, as well as the species family and the and the cycle of the species. For instance, if a plant is a succulent, the fertilizing frequency is set to every 180 days, as sources state that cacti need only two doses of fertilizer per year. If the plant has a perennial cycle, and the current season is fall or winter, the frequency is set for 84 days; otherwise, it's set for 42 days. If a plant is not perennial, then the value is, respectively, 56 days or 28 days, depending on the season. These values were chosen, once again, based on sources found while doing research, that state the general fertilization requirements of perennial and non-perennial indoor plants, considering the rate at which they exhaust the

nutrients present on their soil. Also, during the hotter seasons of summer and spring, fertilization is more frequent, given that plants accelerate their active growth during these seasons, consuming more water and nutrients.[52]

- Meanwhile, for the task regarding the **general checking of a plant's condition**, a more arbitrary approach was used, based on a plant's estimated growing difficulty, that can be rated from a scale of 1-5, with each level having a corresponding checking frequency (the difficulty was determined using an algorithm explained on Chapter 4.1.2.). We considered that users should do this at least once per month, and for harder plants that require more attention, at least once a week, with the frequency returned by the algorithm respecting these intervals.

After the task frequency is determined, a new date for the task is calculated, with this being done simply be retrieving the frequency from the associated task settings, and adding calculating the deadline day for task completion, considering the current day as the start day for the interval. This method is shown in Figure 17**.**

```
// This method calculates the next date for a task of a given type, associated with a Plant
5 usages  Daniel Carvalho
public void calculateNewTaskDateForSingleTask(Tasks_Current task) {
    Plant plant = task.getPlant();
    TaskType taskType = task.getTaskType();

    // Getting the frequency of the Task from the Task Settings, and calculating the new Date for the Task
    Task_Settings settings = taskSettingsRepository.findFirstByPlantAndTaskType(plant, taskType);

    Date newDate = Date.valueOf(LocalDate.now().plusDays(settings.getTaskFrequency()));

    task.setTaskDate(newDate);
    currentTaskRepository.save(task);
}
```

Figure 17 – Method for the calculation of a new task deadline, based on the task frequency associated with the settings for this task. If the settings define the task is automatically rescheduled, this frequency is automatically calculated by the previously presented algorithms; otherwise, the frequency is manually defined by the user.

Besides being called when a new plant is added to the inventory, these algorithms are also called on the following situations:

- When users change a task setting from automatic to manual, they can define their desired frequency for each task, and change this frequency whenever they want. This frequency is then used to calculate a new task date, using the method previously presented.
- Similarly, when a user changes a task setting from manual back to automatic, the task frequency is recalculated using the previously presented algorithms, and the new value is used to determine the new deadline dates for the related tasks.
- When a task is completed, a new instance of the same type of task is scheduled, using the task frequency associated with that task's settings.
- Given that most of the task frequency calculation algorithms take into account the current season, the frequency is recalculated if the current season has changed.

Plant Condition Estimation Algorithm

As mentioned on Chapter 3.2.1., the current condition of the plants belonging to a user's inventory is estimated and presented on the app, via a "traffic-light" system, in which green (defined as "Great" in the backend) represents that the plant is well, yellow (defined as "Normal") represents that the plant may need attention soon, and red (defined as "Bad") represents that the plant urgently needs some care.

To calculate the value for this condition, an algorithm was implemented which considers 2 main factors:

- If a plant has sensors associated with it, this factor is given precedence, with the latest measurements from its sensors, more specifically the comparison between the values read on these measurements and the optimal conditions defined for the species to which this plant belongs, being the parameter used to define the condition of the plant. To do this, the values of the measurements are compared with the intervals related to the optimal parameters of a plant, using as reference values the data presented in Table 3, shown in Chapter 3.2.1. Different importance is given into the different types of sensors supported by the application:
    - The primary values considered in this factor are the measurements from the sensors directly installed on the plants, that is, the soil humidity sensor: is the If a measurement is outside of the range of the optimal interval, the Plant Condition is immediately considered as bad, and requiring attention.
    - In the case in which the measurements taken from the sensors on the division the plant is in, that is, the air temperature and humidity, falls out of the optimal range, the condition of the plant is only regarded as "Normal" and needing attention soon. This is done because these parameters are much harder to control by the user, and there is nothing they can do directly to the plant to better their condition on these parameters, other than possibly moving them to a better location in their house.
- The dates of the tasks to be done are associated with the plant. If the plant has any tasks overdue, its condition is immediately considered bad. However, if it has no task overdue, but some tasks need to be completed in the next 2 days, its condition is set as "Normal", but needing attention soon.

To ensure the smooth and logical flow of execution of the application, this algorithm is called in the following situations:

- Each time a use logs in, the condition of all the plants in their inventory is calculated.
- When a task is completed, the condition of the respective plant is calculated, as the estimation of the number of tasks overdue is an important parameter for this algorithm.
- Each time a task's deadline date is changed.
- When a sensor measurement out of the optimal range for the plant is received on the backend.

The code for the main method applied in this algorithm is presented, as an example, on Figure 18.

```java
// This method calculates the current condition of a plant, taking into account as factors the tasks associated with
// it, and the latest measurements of the sensors associated with it, if applicable
// 0 - BAD; 1 - NORMAL; 2 - GREAT
1 usage  Daniel Carvalho
public int calculatePlantCondition(Plant plant) {
    boolean hasDivSensors, hasPlantSensors;

    // Checking whether the Plant, or its Division, has sensors associated with it or not
    hasDivSensors = plant.getDivision().getSensors().size() > 0;
    hasPlantSensors = plant.getSensors().size() > 0;


    // If a Plant has Plant Sensors, and their last measurement is out of the normal range, return 0
    if (hasPlantSensors) {
        for (PlantSensor sensor : plant.getSensors()) {
            SoilQualityMeasurement measurement = soilQualityRepository.findFirstBySensorOrderByPostDateDesc(sensor);

            if (measurement != null && !checkPlantHumidity(plant, measurement)) {
                return 0;
            }
        }
    }


    // Checking the current Tasks attributed to the Plant
    List<Tasks_Current> currentTasks = plant.getCurrentTasks();

    // If a plant has tasks overdue, return 0; If it has tasks overdue in the next 3 days, return 1;
    for (Tasks_Current task : currentTasks) {
        int state = this.checkTaskDates(task);

        if (state == 0) {
            return 0;
        } else if (state == 1) {
            return 1;
        }
    }

    // If the Plant has Division Sensors, and the temperature is out of range, return 1; otherwise return 2;
    if (hasDivSensors) {
        for (DivisionSensor sensor : plant.getDivision().getSensors()) {
            AirTemperatureMeasurement measurement = airTemperatureRepository.findFirstBySensorOrderByPostDateDesc(sensor);

            if (measurement != null && !checkPlantTemperature(plant, measurement)) {
                return 1;
            }
        }
    }
```

Figure 18 – Main method that implements the algorithm for the estimation of a plant's current condition. The flow of this method shows the importance given to each of the factors taken into consideration for this algorithm, with precedence being given to the latest measurements from any sensors associated with the plant.

Suggestion of New Plants Algorithm

The application also should provide users with a list of plants of the same difficulty for a user. A user with less experience should not be presented with plants of immense difficulty and that, therefore, are hard to take care of.

This algorithm takes into consideration the number of plants a user has, the condition of plants that user has and the difficulty of those plants. Additionally, the user, when first creating their account, is presented with a query in which they should answer on a scale of one to five about their previous experience with plants. The algorithm works by giving each of these parameters a weighted score, producing a result of one to five, and plants with that difficulty will be selected to be suggested to the user.

- Based on the number of plants the user has, the score will continuously increase towards 5, but never surpassing that limit. For this, we used a sigmoid function

which calculates the score associated with the number of plants a user has, in a way that a user with lower plant number has less score in this parameter compared to a user with higher plant number. However, since it's a sigmoid function, both users with a high plant number will get the maximum score, five. The algorithm will increase progressively less depending on the number of plants, with its maximum limit being 5 and minimum 1. This parameter has a weight of 15%.

- To calculate the average plant condition, since the condition of individual can be one of three values – "great", "normal" and "bad", we attributed an integer value to each of these values, "great" being 5, "normal" being 3 and "bad" being 1. Based on the condition of each of the user's plants, the score would be added to a variable in order to perform an average score of the user's plants' condition. This parameter has a weight of 55%.
- The average difficulty of plants is calculated simply by obtaining all of user's plants and calculate the average of their difficulty. This parameter has a weight of 20%.
- Lastly, the user experience is calculated through a query when a user initially creates an account, on a scale of 1 to 5. This parameter has a weight of 10%, and therefore, after querying the plants in the database, we return a list of plants which difficulty is equal to this overall score.

```
double exponent = -k * (plants.size() - x0);
double sigmoidValue = 1 / (1 + Math.exp(exponent));

double sigmoid_score = minValue + sigmoidValue * (maxValue - minValue);

log.info("plants in user inventory: {}", plants.size());

//average score based on plant conditions
double average_plant_condition = plants.stream() Stream<Plant>
    .mapToInt(plant -> {
        if (plant.getPlantCondition() == PlantCondition.GREAT) {
            return 5;
        } else if (plant.getPlantCondition() == PlantCondition.NORMAL) {
            return 3;
        } else if (plant.getPlantCondition() == PlantCondition.BAD) {
            return 1;
        } else {
            return 0; // Default score if condition is unknown
        }
    }) IntStream
    .average() OptionalDouble
    .orElse( other: 0);

double average_plant_difficulty = plants.stream() Stream<Plant>
    .mapToInt(plant -> plant.getSpecies().getDifficulty()) IntStream
    .average() OptionalDouble
    .orElse( other: 0);

double overall_score = sigmoid_score*0.15 + getUser(idUser).getExp()*0.1 + average_plant_condition * 0.55 + average_plant_difficulty * 0.2;

log.info("simgmoid score: {}", sigmoid_score);
log.info("average condition: {}", average_plant_condition);
log.info("average difficulty: {}", average_plant_difficulty);
log.info("user experience: {}", getUser(idUser).getExp());
log.info("overall score: {}", overall_score);

return plantSpeciesRepository.findAll().stream()
        .filter(plantSpecies -> plantSpecies.getDifficulty() == Math.round(overall_score))
        .collect(Collectors.toList());
```

Figure 19 – Code snippet of the implementation of the plant suggestion algorithm.

4.2.5. Sensors Architecture and Integration

As explained in Section 3.2.2 we used two sensors, a DHT11 sensor to measure air humidity and temperature, which we considered division sensors, for the implementation there is only one sensor, but for the conceptualization of the product we separated them. Furthermore, we used a resistive soil moisture sensor to take soil humidity measurements, which we considered as a Plant sensor. These sensors are connected to a NodeMCU module with an ESP8266 microship with wifi capabilities. This module is connected to a RabbitMQ server, as depicted in Figure 12. It's important to note that the connection between the module and the server does not utilize a standard RabbitMQ client. Since the module only supports MQTT communication, we opened a dedicated MQTT port in the RabbitMQ server to facilitate the connection.

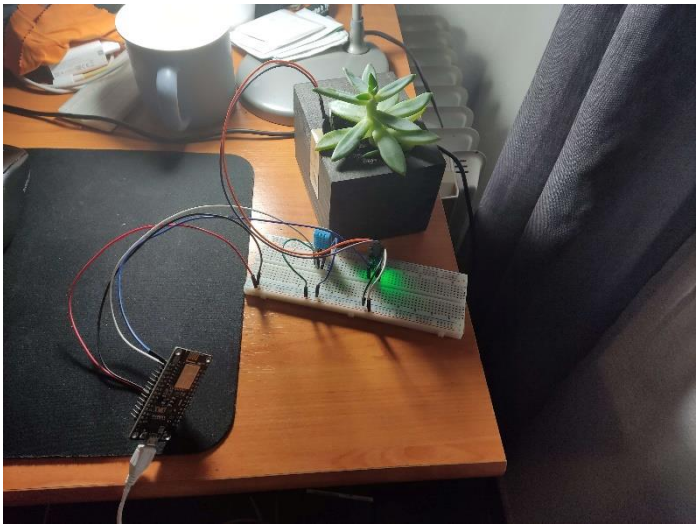


Figure 20 - Sensor integration with real plant

The Spring Boot backend has three RabbitMQ consumers, one for each topic related to a sensor, humidity, temperature, and soil moisture.

As depicted in Figure 21, the process flow of taking a measurement is straightforward. Every minute, the sensors capture a new measurement and transmit it to the NodeMCU module. Subsequently, the module processes these measurements and generates a JSON message for each sensor. These messages are then sent to specific topics in the RabbitMQ server, depending on the type of measurement. For instance, if it's a temperature measurement, the message is sent to the temperature topic.

Within the Spring backend, three consumers are constantly active to consume these messages. They promptly evaluate whether the measurement is within the normal range for the specific plant (in the case of Plant sensors like soil moisture), or for each plant within the division (for temperature and humidity sensors). If the value deviates from the expected range, the Plant status is instantly changed to "BAD." Additionally, the new measurement is stored in the appropriate database table for future reference.

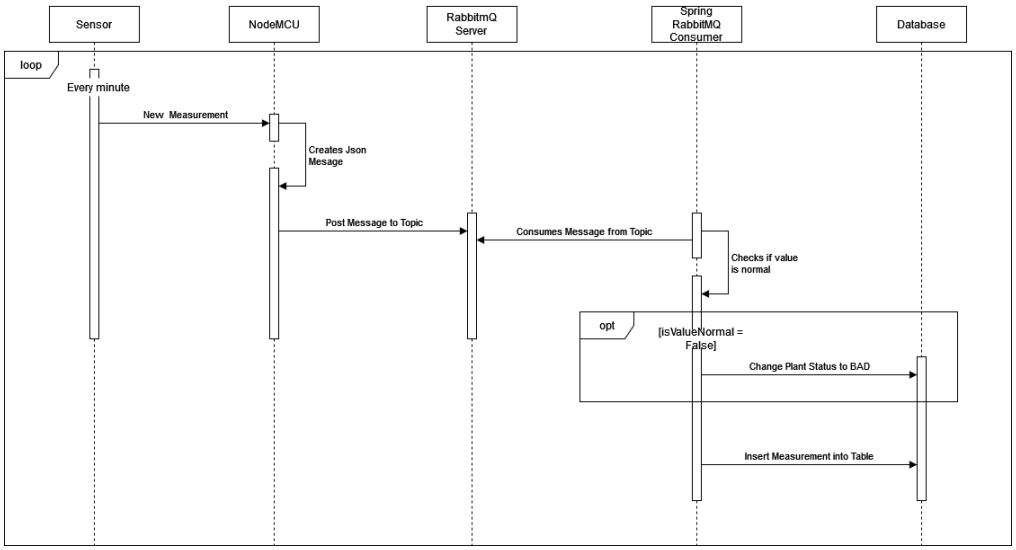


Figure 21 - Sequence Diagram for the sensors flow and integration.

```java
@RabbitListener(queues = "${rabbitmq.air.temperature.queue.name}")
public void consumeAirTemperature(JsonNode jsonNode) {
    LOGGER.info("Received message -> {}", jsonNode);

    try {
        // Extract the necessary fields from the JSON
        String code = jsonNode.get("code").asText();
        double value = jsonNode.get("value").asDouble();

        // Check if the sensor exists
        DivisionSensor sensor = divisionSensorRepository.findDivisionSensorBySensorCode(code);
        if (sensor == null) {
            LOGGER.warn("Sensor with code {} does not exist. Skipping measurement.", code);
            return;
        }

        // Create a new AirTemperatureMeasurement object
        AirTemperatureMeasurement measurement = new AirTemperatureMeasurement();
        measurement.setSensor(sensor);
        measurement.setMeasurement(value);
        measurement.setPostDate(LocalDateTime.now());

        // Save the measurement to the repository
        airTemperatureRepository.save(measurement);

        // Get all plants from the division
        Division division = sensor.getDivision();
        List<Plant> plants = plantRepository.findPlantsByDivision(division);

        // Check if the plants' temperature is within the optimal range
        for (Plant plant : plants) {
            if (algorithmsService.checkPlantTemperature(plant, measurement)) {
                LOGGER.info("Plant {} temperature is within the optimal range", plant.getName());
            } else {
                LOGGER.info("Plant {} temperature is NOT within the optimal range", plant.getName());
                // Update the plant condition to BAD
                plant.setPlantCondition(PlantCondition.BAD);
                plantRepository.save(plant);
            }
        }

        LOGGER.info("Measurement saved successfully");
    } catch (Exception e) {
        LOGGER.error("Error processing the message: {}", e.getMessage());
    }
}
```

Figure 22 - Example of a RabbitMQ consumer implemented at Spring Backend.

4.3. Frontend Module

4.3.1. Technology Used

As previously mentioned, we used React Native Framework for the front-end implementation of the application. React Native is an open-source software framework for building user interfaces created by Meta Platforms, Inc. We chose this technology because of our familiarity with the JavaScript language.

One of the main advantages of React Native is its component-based approach, which enables the reuse of written components. This approach simplifies the development process and results in cleaner code overall. Also, the wide range of community projects available greatly expedites the development process.

4.3.2. Structure

During the development process, we adhered to a screen-based approach, structuring our project accordingly. The main application project was organized with a "screens" folder, containing all the developed screens. These screens were further divided into respective subfolders. Additionally, there were separate "components" folders both at the root project level and within each screen folder. The root "components" folder housed reusable components utilized across multiple screens, such as the “BottomMenu” component. On the other hand, the screen-specific components, like the "PlatCard" component, resided within the respective screen folders.

Furthermore, we established a "services" folder that encompassed all the necessary API calls for retrieving the required data. This approach facilitated easy modification of the URI during development and deployment stages. Additionally, by not clustering the core logic within a single component, we were able to achieve cleaner code organization and structure.

4.3.3. Packages Used

The React Native community has experienced significant growth over the years, contributing to a vast repository of projects that can be leveraged in the development of Android applications. In our project, we made extensive use of various community packages to expedite development.

One of the primary packages we utilized was "react-native-paper," which is a high-quality and compliant Material Design library. This comprehensive library provided a wide range of components, including Input, AppBar, Button, and more. By incorporating these pre-built components, we were able to accelerate our development process while ensuring a consistent and polished user interface.

Another essential package we incorporated was "Axios," a promise-based HTTP client. We utilized Axios for making API calls to our backend server. Compared to the basic Fetch API, Axios offers several advantages, such as a more intuitive and

convenient syntax, automatic JSON parsing, and robust error handling capabilities. These features greatly simplified our API integration process and enhanced the reliability of our communication with the backend.

In addition to the previously mentioned packages, we used several other packages that played a crucial role in implementing key functionalities within our application. For managing tasks, we relied on the "react-native-calendars" package, which provided efficient task management capabilities. Furthermore, to visualize sensor data in the form of graphs, we incorporated the "react-native-chart-kit" package. These packages, among others, greatly contributed to the overall functionality and appeal of our application.

By utilizing community projects, we were able to develop a clean and modern-looking application. The availability of these well-established and reliable packages allowed us to streamline the development process and focus on delivering a high-quality user experience. Overall, our decision to utilize community projects proved to be a valuable choice in achieving our development goals.

4.3.4. Client-Side Cache

To implement client-side caching, we utilized the Async Storage package, which provides a key-value storage system accessible throughout the entire application. The cache was specifically employed for storing static data that did not require retrieval from the API upon every component reload. This included information such as plant categories and species. Additionally, the cache served as a session management tool for the user.

To maintain data freshness, a time-to-live (TTL) protocol was implemented. Each cached item had a designated expiration time, typically set to one day. This approach ensured that the data remained up to date and avoided serving outdated information to users.

```
const setItem = async (key, value) => {
    // ttl one day
    const ttlInSeconds = 86400;

    try {
        const item = {
            value,
            expiresAt: Date.now() + ttlInSeconds * 1000
        };

        const jsonValue = JSON.stringify(item);
        await AsyncStorage.setItem(key, jsonValue);
    } catch (e) {
        console.log(e);
    }
};
```

Figure 23 - Set an Item in the client side cache

4.3.5. Firebase Integration

Firebase, being a cloud service utilized solely for photo storage during user uploads, was solely implemented on the frontend side of the application. To enable this functionality, a configuration file was created, as depicted in Figure 24, containing the necessary Firebase credentials for the project. Additionally, the "react-native-firebase" package was installed to facilitate integration with Firebase.

```
// Import the functions you need from the SDKs you need
import { initializeApp } from 'firebase/app';
import { getStorage, ref } from "firebase/storage";

// TODO: Add SDKs for Firebase products that you want to use
// https://firebase.google.com/docs/web/setup#available-libraries

// Your web app's Firebase configuration
// For Firebase JS SDK v7.20.0 and later, measurementId is optional
const firebaseConfig = {
  apiKey: "AIzaSyCGXQOZd7YGCLf4t6IRWBIWM1wOEtAnx4g",
  authDomain: "growmate-59791.firebaseapp.com",
  projectId: "growmate-59791",
  storageBucket: "growmate-59791.appspot.com",
  messagingSenderId: "28075949794",
  appId: "1:28075949794:web:9e19ac29b54bd6e177a8bb",
  measurementId: "G-39NR2EFT59"
};

// Initialize Firebase
const app = initializeApp(firebaseConfig);

const storage = getStorage(app);

export { storage };
```

Figure 24 - Firebase configuration file

Two additional methods were developed to interact with Firebase, Figure 25. The first method allowed for adding a photo to the storage, while the second method facilitated the deletion of a photo. However, it's worth noting that no specific method was

implemented for retrieving the photo. Instead, the URL of the photo was saved in the PostgreSQL database, and that URL was used to access the photo as needed.

```
const uploadImage = async (imageUri, userID, plantName) => {
    const img = await fetch(imageUri);
    const bytes = await img.blob();

    const storageRef = ref(storage, userID + plantName);

    await uploadBytes(storageRef, bytes);

    const downloadUrl = await getDownloadURL(storageRef);

    return downloadUrl;
}

const deleteImage = async (userID, plantName) => {
    const storageRef = ref(storage, userID + plantName);

    deleteObject(storageRef).then(() => {
        console.log("DELETED " + plantName);
    }).catch((error) => {
        console.log(error);
    })
}
```

Figure 25 - Frontend methods for Firebase integration

4.4. Project Management

In this section, we will discuss the project management practices and tools used to organize our work throughout the implementation of the project.

4.4.1. Backlog Management, Agile Practices and Work Assignment

To organize the development of the project, we decided to follow an Agile philosophy the Agile framework, as this is the methodology, we've been using in all the bigger projects done along the course, and thus, is the philosophy we're most accustomed to.

Definition of the Milestones

For the first few weeks of the project, we followed the milestones calendar defined by the professors of this subject. As such, the first milestone, lasting one week, was focused on the inception of the project, the presentation of the lifecycle objectives and the definition of the project calendar.

After this, the second milestone, which took place over two weeks, focused on the elaboration phase of the project, with the goal being the definition of the functional and non-functional requirements needed to successfully accomplish the goal of the project, and the definition of the architectural design of the system.

The summary of the goals achieved on the first two milestones are presented on Table 5.

Table 5 – Main tasks completed during the first two milestones of the project.

Milestone 1 (14/02 – 28/02)	Milestone 2 (01/03 – 14/03)
• Inception of the idea • Definition of the context and the goals of the problem • Presentation of the lifecylce objectives • Definition of the calendar for the project • Creation of a risk management plan	• Functional and non-functional requirement analysis • Use cases and actors • Definition of core user stories • Non-functional UI prototypes • Determination of the sensors to be integrated into the system • Definition of the problem domain • System architecture and deployment plan

The final two milestones focused on the development phase of the project, with the goal being the presentation of a functional protype at the end of the 3rd milestone, which had nearly one month of duration; and the presentation of the final increment of the project, at the 4th milestone.

Agile Approach During Development

During the development phase of the project, we decided to execute an iterative and incremental development process, with this phase being divided into sprints, lasting from 1-3 weeks, depending on the evolution of the implementation and the needs of the

members of the team. This allowed us to divide the project into smaller and easier parts, allowing us to present regular increments to the project, and to better elaborate and specify the needs of the end user of the application, as well as the functionalities that needed to be implemented along the way.

The management of the backlog of tasks for the development was done through the JIRA platform, with a project being created to host the GrowMate organization. These tasks represented issues that needed to be tackled to implement the functionalities required for the application.

At the beginning of each sprint, the team members decided, in a Sprint planning meeting, which of the issues present in this backlog should be prioritized for that sprint, with estimations of effort and time being for each task being represented in the form of story points, rated from a scale of 1-5. This meeting allowed the prioritization of the work required for that spring, and the subdivision of issues into smaller subtasks. After discussion among everyone, each of the tasks was attributed to a member of the team, that became responsible for its implementation and management on the Jira platform.

On the Jira board corresponding to each sprint, three categories for the tasks were created: “Not started”, “In progress” and “Done”. Once a team member starts working on a task, they should move it to the “In Progress” category, finally moving it to the “Done” category once the task was completed.

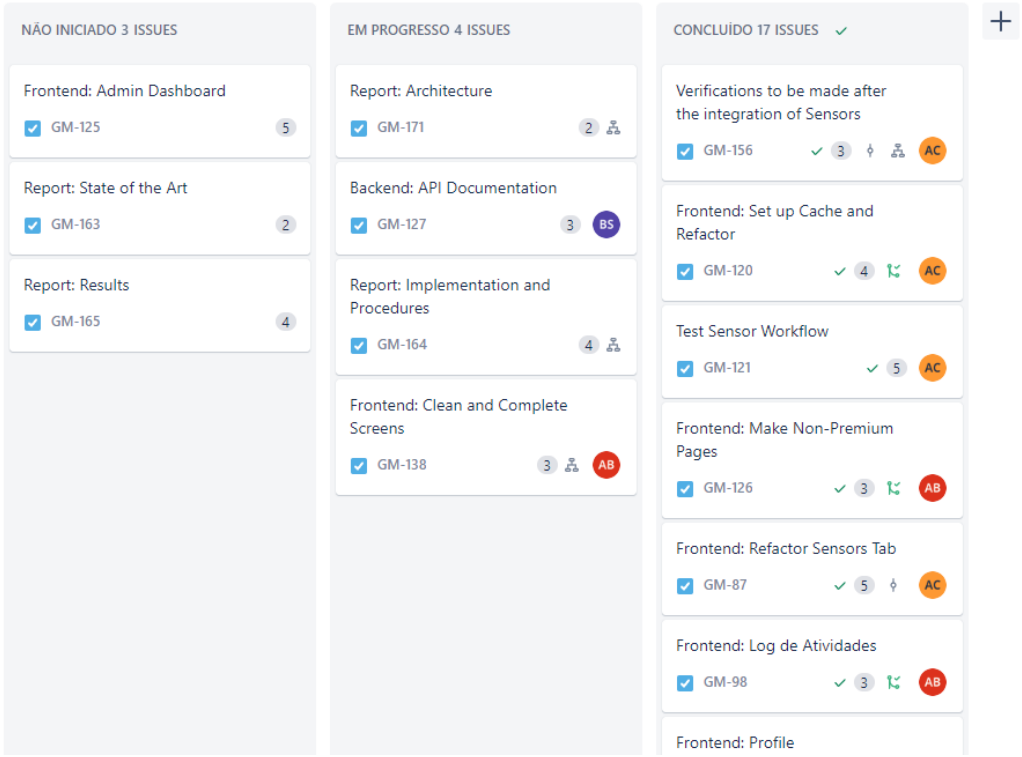


Figure 26 – Example of the JIRA board midway through the final sprint of the development.

For sprints that lasted more than one week, the team had weekly meetings in which team members updated the others on the progress of their tasks, on the difficulties they felt, and in which new emerging issues or problems where tackled and added to the backlog. Besides the weekly team meetings, we also had meetings with the Project Supervisor once every two weeks.

Besides the meetings, a team discord server was created, to encourage regular communication between the members, about the tasks they were currently tacking, problems they faced, and to further plan and discuss the goals for each week. This server also allowed us to keep notes and tabs on all the decisions made during the developmental process, as well as notes from the meetings we had, amongst the team and with our supervisor.

The sprints defined for the development phase of the project, as well as the issues tackled on each of the sprint, are summarized on Table 6.

Table 6 – Main tasks concluded on each of the development sprints.

Sprint	Tasks
Sprint 1 (15/03 – 23/03)	• Finishing and Validating Non-Functional UI prototype • Setup of sensors architecture and scripts to connect them to RabbitMQ • Skeleton of the Spring Boot and Docker Compose projects • Database modelling
Sprint 2 (23/03 – 30/03)	• Seting up of the mobile app project • Creation of the Premium User home screen • Creation of the Spring Data JPA entities • Documentation Website
Sprint 3 (30/03 – 10/04)	• Implementation of the core use cases for the premium account • Plant Inventory, Plant Page, Discover Plants, Task Management, Sensors Management, Division Management, Species Profile, and Plant Categories screens • Adaption of SQL database to include sensor measurements • Initial population of the database • Creation of the backend controllers and services required to implement the previous screens
Sprint 4	• Backend refactoring of the remodelling of the tasks to

(13/04 – 11/05)	include the possibility for task settings controller by user • Creation of the task scheduling algorithm • Creation of the plant difficulty algorithm • Remodelling of the task management screen • Add Plant, Add Division and Add Sensor screens • Implementation of Firebase
Sprint 5 (12/05 – 30/05)	• Plant suggestion algorithm • Plant condition algorithm • Initializer, Login and Account Registration screens • Management of sensors screen refactor • Setting up of the cache • Non-Premium User screens • Stabilization of the REST API, with refactoring of some methods • Activity logs of the users • Final population of the database • Conclusion of the sensor integration workflow • Report • Usability tests

4.4.2. Development Workflow

The development workflow used for this project was an adapted version of the Git Feature Branch workflow.

The code for the project was all implemented on the same Git repository, hosted on an organization created for GrowMate. In this repository, the development workflow used was an adapted version of the Git Feature Branch workflow, with the logic for the branches described as follows:

- The "**main"** branch was used to store official "releases" of the product. In our case, we considered two main releases: the project increment developed at the end of the 3rd and 4th milestones previously described. This branch should only receive pull requests from the dev branch.
- The **"dev"** branch served as the main integration hub for each feature". This branch should only receive pull requests from the backend and frontend branches.
- A **"backend"** branch, housing the integration of features related with the backend or the database implementation of the system.

- A **“frontend**” branch, responsible for the integration of features related to the frontend implementation of the system.
- A **branch for each singular feature**. The naming of these branches followed a convention, starting with either “backend” or “frontend”, and describing what feature was being implemented in the code developed to this branch. (For example, “frontend-plant-inventory-screen”.

As such, team members were encouraged to only create commits to the branches related to the features they were working in, creating a pull request to either the backend or frontend branch, depending on their work, to be ideally reviewed by another team member. Once these branches were stabilized, their work could then be pushed into the dev branch.

Team members were also encouraged to name their commits following a naming convention, that was based on the code of the task to which their commit was associated on JIRA (for example, “GM-94 finished adding species to database”). This allowed the automatic tracking of the commits and pull requests related which task on JIRA, using a JIRA-GitHub hook.

4.4.3. Decisions Taken During Development

As described on Chapter 3.3.2., some changes on the initial plan for the architecture of the system were made along the development phase of the project, such as the scrapping of the use of the InfluxDB timeseries database, or the redesign of the sensor’s architectural design.

Besides these changes, a few functionalities initially planned were eventually scrapped, due to the time constraints faced while developing the project. In order to decide which features to scrap, and which features to focus on, the team members decided to follow the results of the questionnaires undertaken during the requirements gathering phase of the elaboration of the project, as described on Chapter 3.1.1. Thus, some of the decisions taken were:

- The forum feature, where users could share tips about the plants they’ve grown, and implementing a reputation system that took into consideration the experience of the users with the application to determine which comments were given priority when showing in the plants page, was left as future work, as this was quite a complex implementation that required attention taken away from the core features of the application, and that wasn’t seen as very essential by prospective users, although most considered it interesting for future releases.
- The journalling feature, in which users could take photos of their plants along time, and take notes regarding their development, was also not implemented due to a lack of time. Again, we considered these functionalities to be not as important, and also not easy to fully test and show off with the time we had available to us for the development cycle of the project.

The scrapping of these pages allowed the team to focus on ensuring that the core functionalities of the system were well implemented and developed. This allowed us to refactor some of the functionalities, based on feedback taken from our supervisors, and also informal testing with colleagues and friends, during the development. Some of these refactors included:

- Initially, the task management by the application was going to be entirely automatic, with users not being able to manually schedule any tasks. Some feedback given by us said that, although the algorithms to automatically reschedule plants made sense and were useful, fully taking away the ability to manually schedule tasks from the users was counter productive, and the best solution was a compromise that included both possibilities. This required an almost complete redesing of the task modelling, not only on the frontend pages and screens, but also in their modelling on the database and the services of the Spring Boot project.
- The divisions and sensors management was also refactored along the development project, as initially we considered that plants and sensors were rigidly associated with a plant, taken away some flexibility from the user. This was later retought and redesigned.

5. Results

In this chapter, we will present the screenshots of the functionalities implemented on the final version of the application, presented at the end of the semester. We will describe each of the main screens developed for the project, as well as the functionalities associated with them, and in cases in which these screens and/or functionalities vary with each of the User actors previously defined, those differences will be outlined.

We will also present the results of the usability tests done on this implementation.

5.1. Implemented Functionalities

5.1.1. User Registration and Login

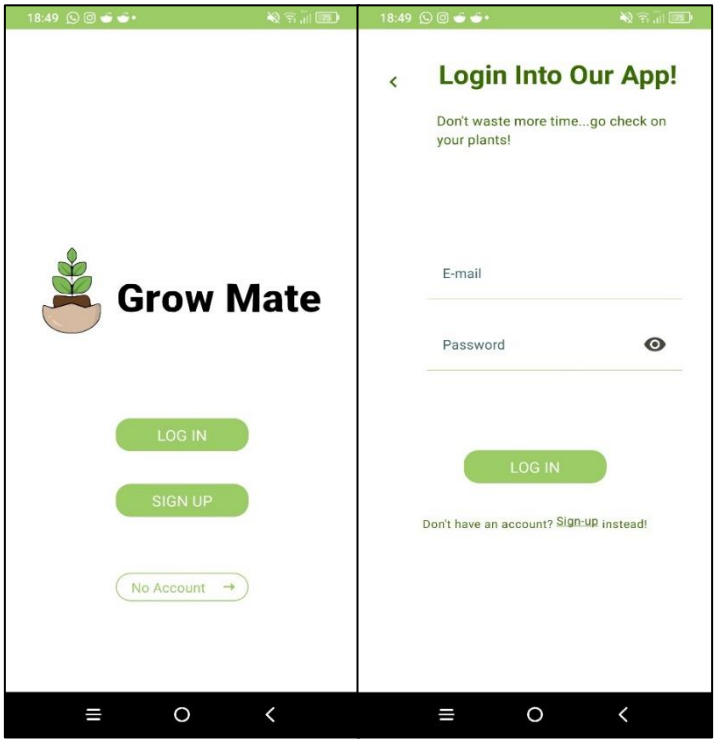


Figure 27 – Initial and login screens.

When users register in the app, besides the usual information required to create a profile, they also have to rate their previous experience with plants, in a scale of 1-5. This will be taken into consideration in some of the algorithms implemented on the backend, such as the new plants suggestions algorithm.

As previously stated there are 3 types of users: Premium, Non-Premium, and Guest users, with different types of functionalities available to them. For guest users, they don't need to login or to register to use the app, being able to access the "Discover new plants" page, described on Chapter 5.1.8, via the "No account" button shown on Figure 27.a).

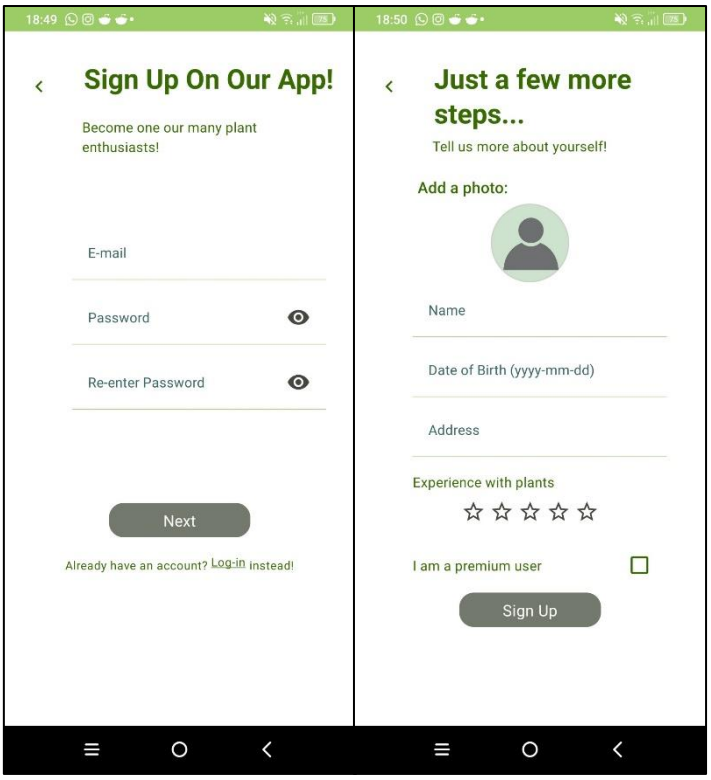


Figure 28 – Profile Registration Screens.

5.1.2. Plant Inventory Screen

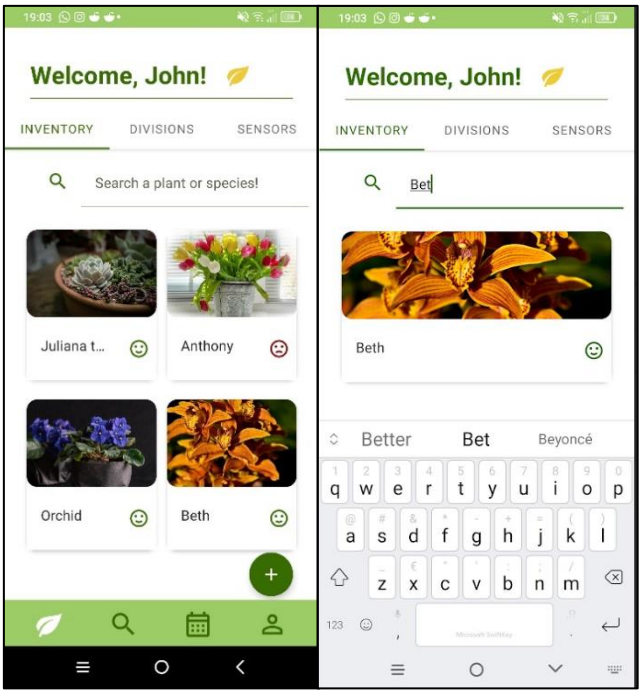


Figure 29 – Plant Inventory page for a Premium Account. Figure b) shows the search for one of the plants in the inventory, by name.

Once an existing user logs into their account, they enter the Plant Inventory page, as shown on Figure 29, which contains an overview of their current inventory of plants. In this section, each card represents one of the plants associated with their profile, with the photos and names given by the user, and with an emoji face symbolizing their current state, calculated via the plant condition algorithm described on Chapter 4.2.4. Pictured in Figure 29, Juliana, Orchid and Beth were in a great condition, with Anthony being in need of some attention. In Figure 29.b) we can see that it's possible to search for the plants in the inventory, by either name or species.

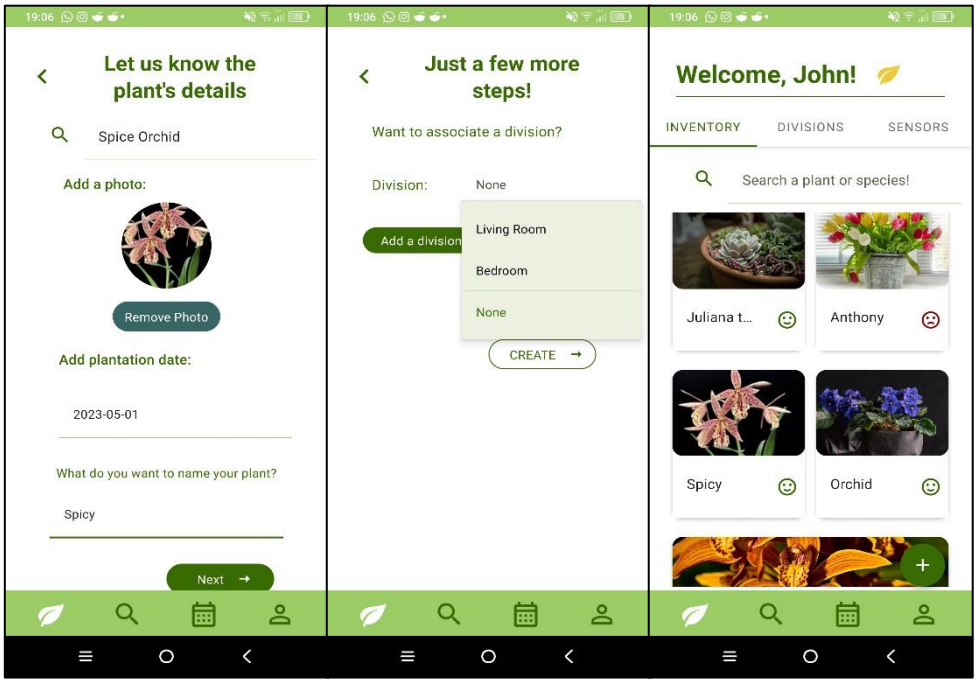


Figure 30 – Add plant screens and workflow.

The ”plus” button on the bottom of the page allows the addition of a new plant, leading into the screens shown on Figure 30. Here, users can select the species the plant belong to, and add the relevant information of their plant. They can also associate the plant to an existing division, or choose not to do so. On figure 30 c) we can see that the newly created plant was added to the inventory.

Given that it still has no sensors associated with it, and that the tasks scheduling are calculated on the moment the plant is added to the inventory, (meaning it's considered they have no overdue tasks at that point), the determined condition for a new plant is always Great.

Note that there are no differences between the Premium and Non-Premium users in these screens, besides the presence of the golden leaf next to the name of the premium users, as shown in figure 30 c).

5.1.3. Divisions Screen

In the division screens, users can manage their inventory of divisions associated with their account, as well as displaying the plants present on each division at the current time.

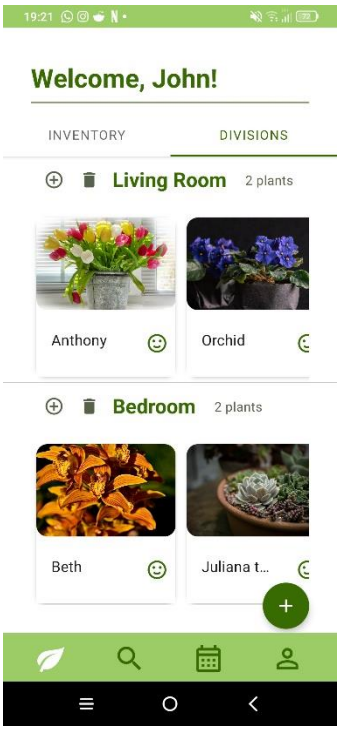


Figure 31 – Division screen of the application. This print belongs to a non-premium account, hence the lack of the "Sensors" tab, and the golden leaf next to the name of the user.

On the "plus" button, users can add a new division to their profile. Besides, when clicking on the plus sign next to the division titles, users can add plants to the respective division, with it being possible for the user to switch plants from a division to another using this feature. Initially, we implemented this screen using a kanban-like slide component that allowed for plants to switch divisions directly on the screen presented in Figure 31. However, this had to be changed due to problems when implementing the connection of the page to the backend. For future work, we would try to refactor the page in order to implement this feature again.

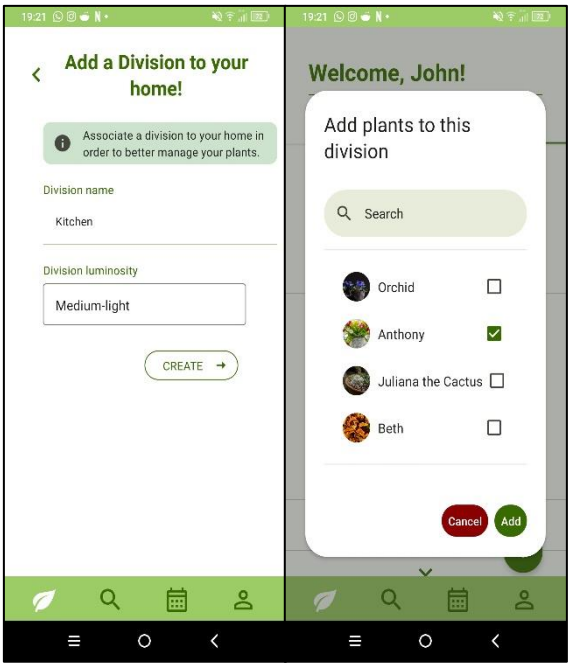


Figure 32 – Add Division and Add Plant To Division screens.

Users are also able to delete divisions. When that happens, the plants belonging to the division they've deleted are disassociated from any division, but are still saved in the database, being presented on the Plant Inventory page.

5.1.4. Sensors Screen

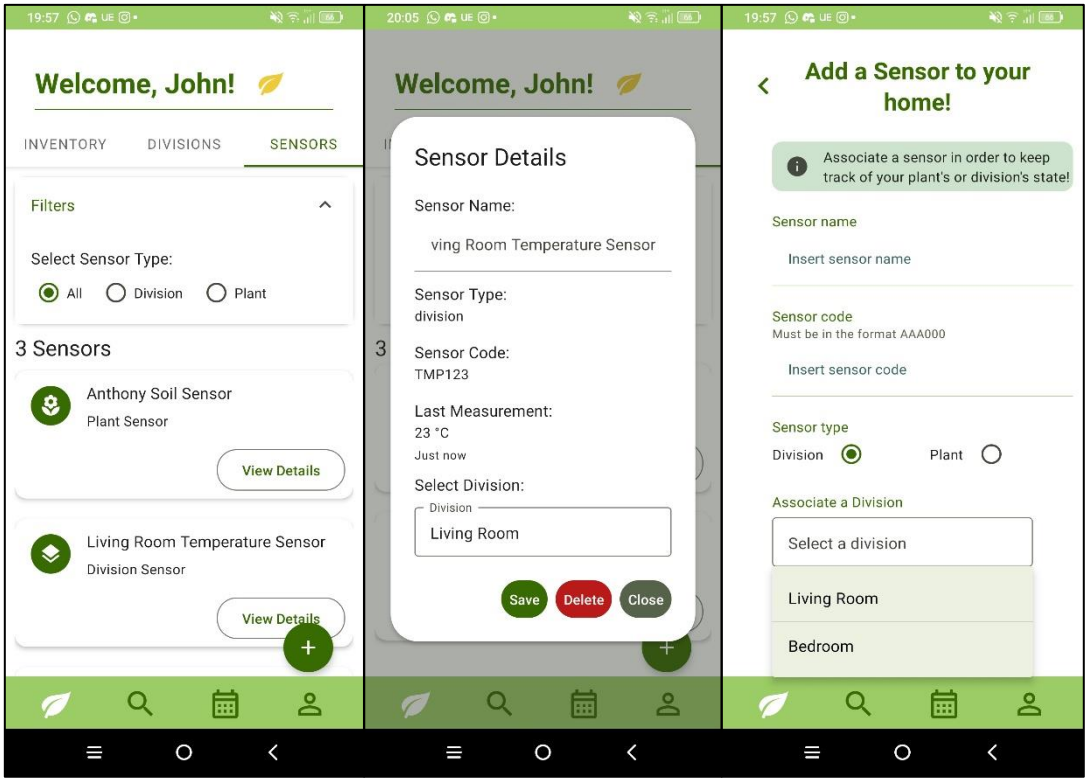


Figure 33 – a) Overview of the Sensors management page; b) Modal showing the details of a chosen sensor; c) Add a Sensor screen

The sensors screen allows the management of the Plant Sensors and Division Sensors associated to the account of an users. As sensors are one of the features reserved only for Premium users, they are not available for non-Premium accounts.

In this page, users can check the list of sensors, and filter based on the plant sensors or division sensors. When they check the details of the sensors, they can view the information regarding this sensor, as well as the latest measurement, and when this measurement was made. Users can also switch sensors to other plants/divisions.

Finally, users can add a new sensor to their account, inserting the necessary information.

Plant sensors, and their measurements, are also shown in the respective plant's page, as shown in Chapter 5.1.6.

5.1.5. Tasks Screen

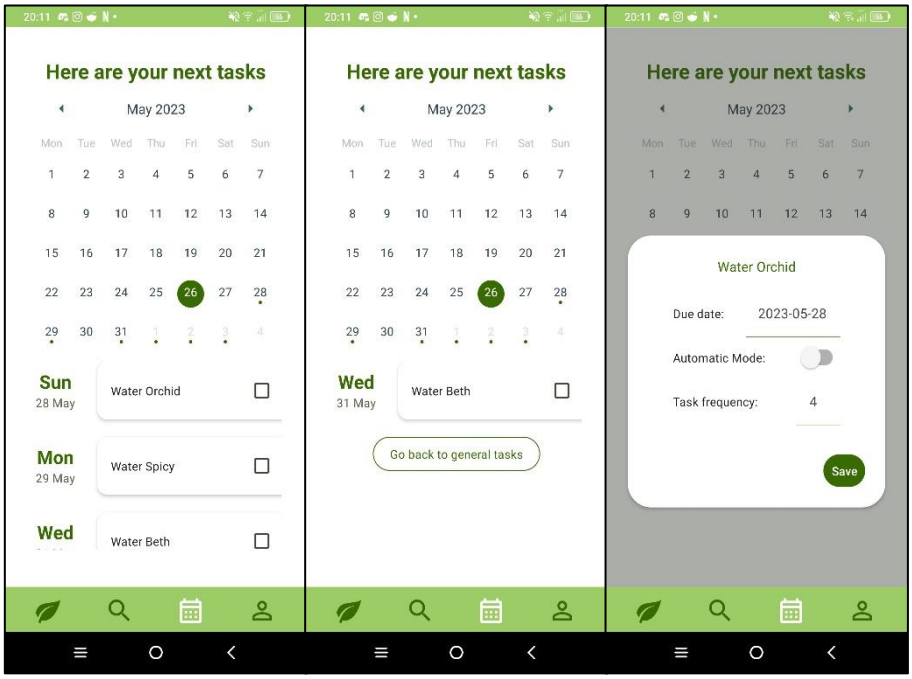


Figure 34 – a) Tasks calendar page, showing all of the tasks associated with the plants in the user's account; b) Filtering the calendar to show only the tasks from May 31st; c) Editing a task's setting.

In the Tasks screen, users can check a calendar and a to do list of all the tasks associated with the plant's in their inventory. The to do list at the bottom of the screen shows all of the tasks, ordered by nearest due date. If a user selects a specific date on the calendar, the list on the bottom is updated to show only the tasks for that day.

As explained on Chapter 4.1.2. and Chapter 4.1.4, each plant has tasks associated with different types, with an algorithm being used to reschedule the tasks due date once the task is completed, based on the plant's conditions and parameters, as well as other conditions such as the current season. If a user selects a task presented in the to do list, they can edit the task's setting, to change if from manual to automatic mode, or vice-versa. In case they change their task to manual mode, they can then define their desired task frequency, overriding the application's algorithm. Users can also freely reschedule a task to another day, without having to change the task's settings.

Once a task is completed, users should register this in the Task screen, by clicking on the box related to the task. As stated, a new task of the same type is scheduled based on the task settings, appearing in the calendar.

5.1.6. Plant Page

Back in the Plant Inventory screen, shown on Figure 29, users can click any of the plants associated to their profile to check information about them and their current condition.

In this page users can check the plant's species, as well as the information related to the species, by clicking the "Check species info", which redirects them to the screen as presented on Figure 40 of the upcoming Chapter 5.1.8.

They can also see the current condition of the plant, calculated by the algorithm previously described, and check and edit the basic plant information.

For premium users, they also get a glance of the latest measurements of the sensors associated with either the plant, or the division the plant is placed in.

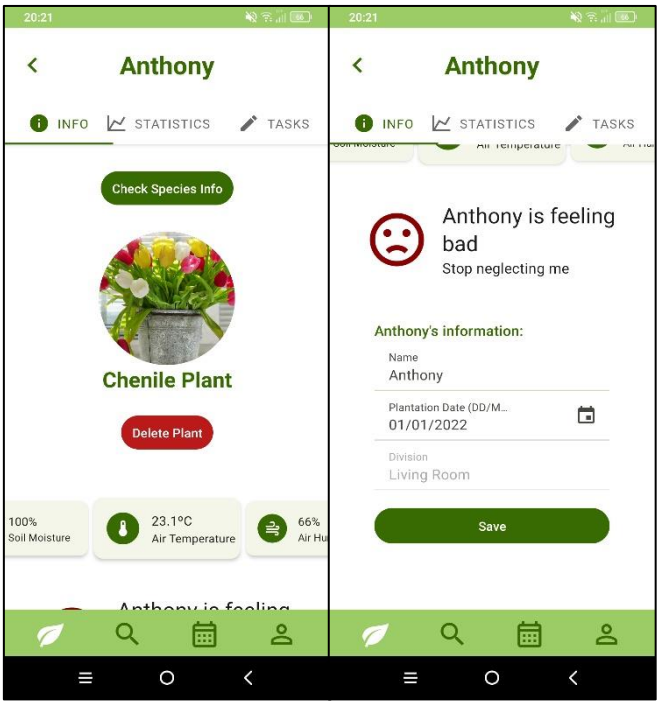


Figure 35 – Plant screen with information about a plant in the inventory, including their latest measurements, current condition, and plant information.

On the statistics tab, available only to premium users, the graphs of the measurements of the sensors associated with the plant in the last 7 days are shown.

On the tasks tab, users get a page like the tasks screen described on the previous section but showing only the tasks related to this specific plant.

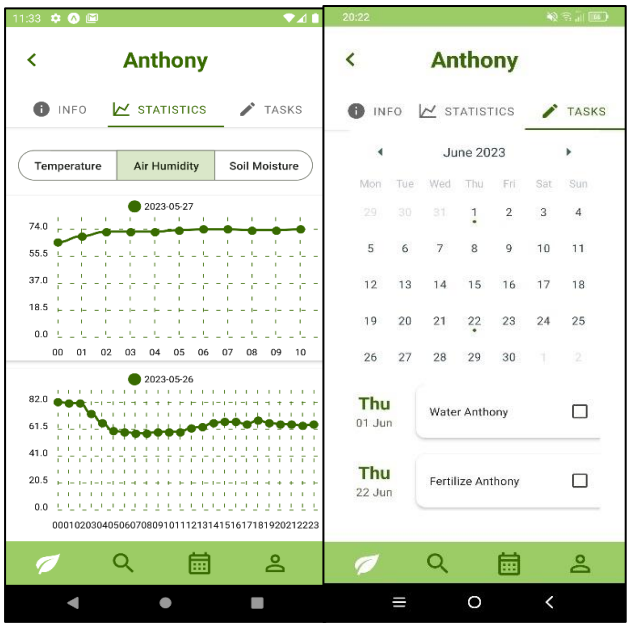


Figure 36 – Statistics and tasks tab on the plant screen.

5.1.7. Profile Page

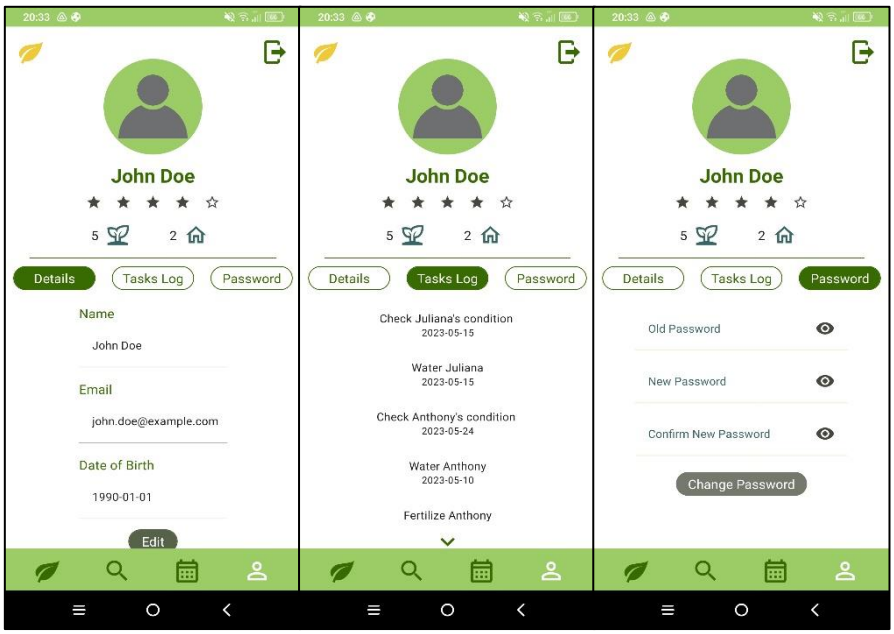


Figure 37 – Profile information screens.

On the profile pages, users can check information about their profile. In the top of the page, the stars represent their current estimated experience with plants, from a scale of 1-5, based not only on the evaluation they make when creating their profile, but also on the plants in their inventory, their associated difficulty, and other parameters, such as the number of plants they have killed.

On the bottom, in the details tab, users can check and edit their profile information. Users can also check the log of tasks they have done previously, as well as change their password.

5.1.8. Discover New Plants Screens

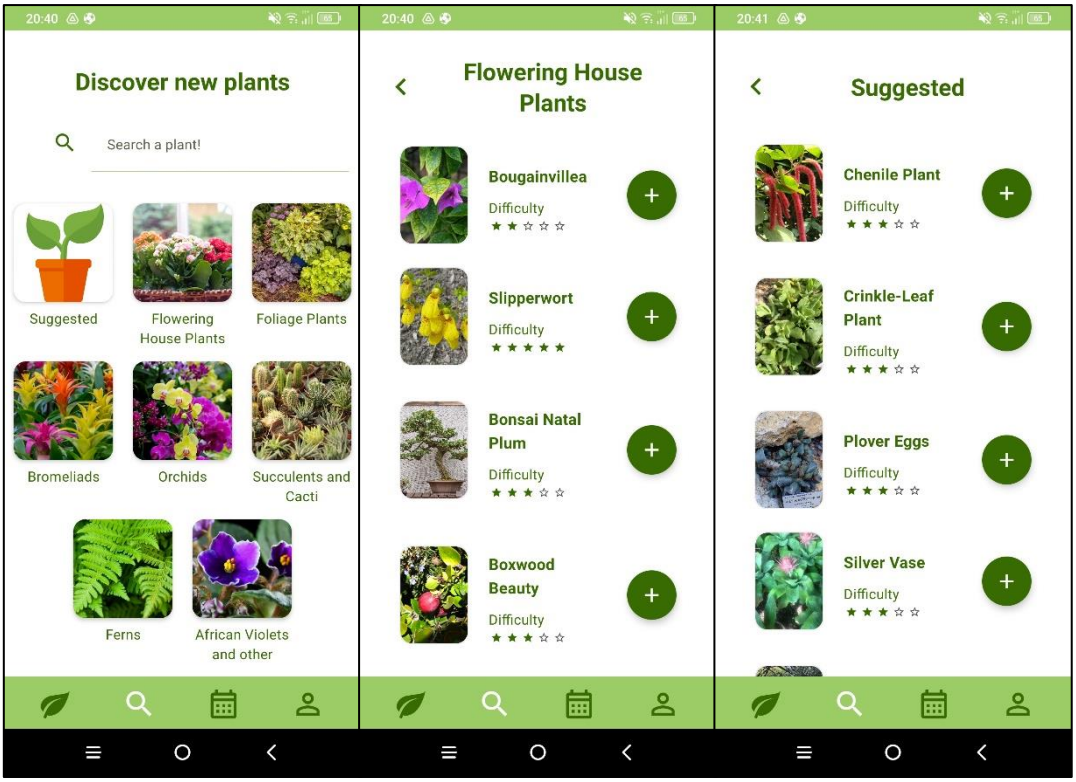


Figure 38 – a) "Discover New Plants" screen, where users can see the species families supported by the application, and search for any specific plant using the search bar, by either common or scientific name; b) The page for the "Flowering House Plants" family; c) The page for the plants suggested by the GrowMate algorithms.

On the "Discover New Plants" screen, users can browse through the catalogue of plants supported by the system. In this page, each of the species' families defined in the domain of the system is presented, as well as a group for plants suggested for the user by the backend algorithms previously described, and a search bar, which allows users to search a specific plant by either their common or scientific names.

If users click on any of the groups, the full list of plants of that family is shown, as well as their estimated difficulty. Users can add a plant to their inventory via these screens.

The suggested group shows plants suggested to the users, taking into consideration their experience and the species' difficulty, as previously explained.

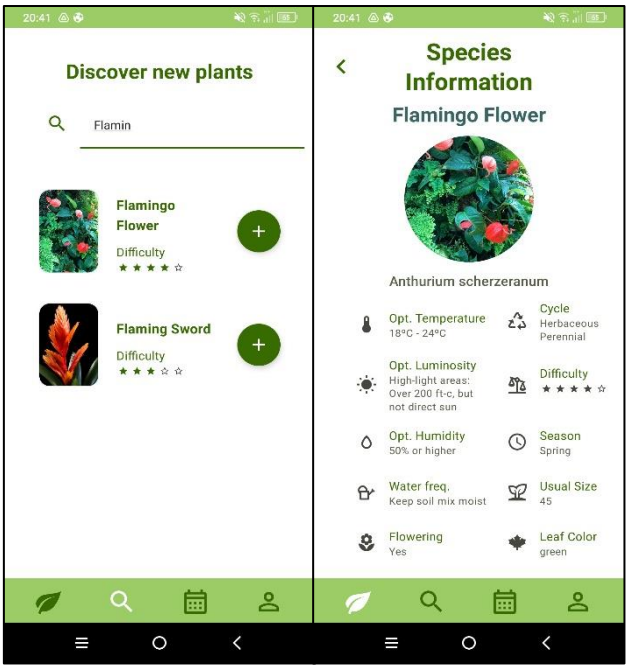


Figure 39 – a) "Discover new plants" screen state change when using the search bar; b) Species information page for a specific plant.

Once a user starts typing on the search bar, the results are filtered to the query made by the user, presenting only the corresponding plants. If users click on a plant, they are taken to their information screen, which state their optimal parameters, as well as some other information regarding the species.

As mentioned before, every user can browse the full catalogue of plants of the app, even unregistered users, meaning that these screens are available for anyone with the GrowMate app. However, for unregistered users the "Suggested" category of plants doesn't appear, as there isn't enough information about them for the algorithm to make any personalized suggestions.

5.2. Usability Tests

Considering that we aim for our application to be used by a vast market segment, composed both by people with or without experience in indoor plants maintenance, and belonging to all age groups and technological backgrounds, one of the biggest focuses on the development of the application was ensuring its usability and intuitiveness, as conveyed in our list of non-functional requirements presented in Chapter 3.1.5.

To guarantee this, we did some informal tests and interviews based on the non-functional and functional prototypes developed during the 3rd milestone of the project, that led to some of the refactoring decisions explained earlier, as well as usability tests made on the final version of the system, to check which areas of the app could have been done better in potential future work.

5.2.1. Conduction of the Tests

For this final usability test, users were asked to do the following tasks, in a pre-existing premium test account, connected to real-life sensors:

- Check the information about a plant in the inventory, including name, species info, optimal parameters and current condition
- Verify the plants located in a given division
- Switching plants from one division to another
- Checking the latest sensor measurements for a plant in the inventory, and historical values for the previous days
- Verifying how many sensors are associated with the account
- Browsing the “discover new plants” section, including a categories page, finding a certain species and checking its difficulty

Users were also asked to create a new non-premium account, on which they did the following tasks:

- Adding a new plant to their inventory
- Registering new divisions, and associating plants to these divisions
- Checking the created tasks and their due dates
- Changing the settings of a specific task from the automatic mode to the manual mode, using a task frequency chosen by them

The tests were conducted by team members. At first, users were able to navigate the app freely to explore its functionalities and the screens. After one minute of this, the team members started the conduction of the test, asking the users to complete each task one by one. Team members did not interfere in the tests, unless the users asked for helped.

While users were doing the tests, team members were completing a forms in which we described if they were successful in doing the tasks, if they found any hardships, and writing down any comments made by the users. At the end, users completed a small questionnaire, rating the application on a scale of 1 (Highly disagree) to 5 (Highly agree) in various questions, encompassing the usability of the app, the flow between screens, the design, the intuitiveness and the usefulness of the functionalities. Users were then

asked to say positive and negative remarks about the app, as well as potential suggestions or any other comments they had to make about the app.

5.2.2. Main Results and Conclusions

The usability tests were conducted with 6 people of varying age groups. However, all of them had never had any contact with a plant management application before.

The full list of tasks, as well as results from the tests, are presented on Appendix D.

All of the tasks were successfully completed by the users, without the need of requesting help from the team members. However, users faced some difficulties in some of the tasks. This was notable in the cases of:

- The biggest gripe users faced was when changing a plant from one division to another, in the screen shown in Figure 31. Five out of the six users tried to swipe a plant from one division to another, only figuring out that they had to click the “Plus” button next to the division in order to edit the plants belonging to it, after they realized that swiping didn’t work. Users suggested implementing the swiping feature in the future.
- The other task that users felt could be shown in a more intuitive way, was while managing the scheduling of tasks, as shown in Figure 34.c). Some of the users tried to immediately switch the task from automatic to manual using the button, before figuring out that they first had to click on the “Edit” button. One user suggested changing the icon of the button to add plants to the division from “plus” to another symbol, as they though that button was to add a new plant to the inventory.

The answers given to the questions asked during the post-questionnaire form are summarized on Table 7.

Table 7 – Average values of the answers given to the Post-Test questionnaire questions by the users. The scale used was from 1 (Strongly Disagree) to 5 (Strongly Agree).

Question	Average Answer (1-5 scale)
It’s easy to understand what each screen and component does	4.2
The colors used for the app allow for easy reading	4.7
The existing information about each plant is enough for me to take good care of them	4.5
The icons used were intuitive	5.0
The overall design of the application is nice	4.3
The navigation between screens is intuitive and easy to execute	4.0
Based on my experience, I believe the use of this app would be easy and not confusing in the future	4.0

On a separate question, users were asked to state their favorite aspects of the application. Among the answers received, were the following:

- Most people found the application very intuitive and easy to learn, citing the flow between the screens as easy to grasp.
- People cited the “traffic-light” emoji system to grasp the plant condition at a glance as very effective.
- Most users highly praised the design and aesthetics of the application.
- Users consider the integration of sensors for plant monitorization and the automatic scheduling of tasks has very positive functionalities.
- Users considered the catalogue of plants, and their information, to be very broad.

When citing the most negative aspects:

- Users mentioned the previously described issues in the Divisions page.
- Some users suggested that in the Plant page, the reasons why the condition of plants in the “yellow” or “red” state should be shown, so they could more easily identify why these plants required attention.

Some users also presented suggestions for possible new features in the future:

- More statistics regarding the sensor’s measurements, such as daily or weekly averages for the observed values.
- Information about relevant pests and diseases that can infect the existing plants, and how these could be treated. This feature had been considered by the team, but was scrapped due to time constraints.

6. Conclusion

The purpose of this project was the development of a proof of concept for a mobile application centered on the care of indoor plants, focusing on overcoming some of the shortcomings of competing similar applications. The goal was to make an application suitable not only for plant enthusiasts, that already have some experience with this hobby, but also for new people that want to start growing plants on their home, but have a hard time figuring out which types of plants are indicated to their level of experience, or tracking down the tasks needed to grow them, and the ideal conditions to maintain them in. Thus, we decided to include some novel features, like the automatic scheduling of tasks for each plant based on their parameters and current condition, or the integration of sensors to measure some of the current plant's parameters, such as their soil moisture, or the air temperature and air humidity of the division they're currently in.

We considered we achieved this goal, ending up with an application that would be very useful for any users wanting to track and monitor their plants' state, and with functionalities that helps experienced and unexperienced user's alike in figuring out what plants need their care the most, when to do certain tasks such as watering the plants or fertilize them, searching for new plants to grow that either fit their experience level, or could provide a new challenge, with a design that is appealing and a flow of execution that is intuitive and easy to figure out to people of varying backgrounds and different levels of experience with technologies, representing the vast potential market segmentation for a product like ours. These conclusions were validated by the results of the usability tests undertaken at the end of the project.

On a personal note, we also considered that this project was very enriching for our development as software engineers, given that we developed the idea and the solution from scratch, and that it was the project with the bigger scale we developed during our degree. Thus, we gained new skills and new hands-on experience with all main steps of the software development process, not only the programming and implementation part, but also the inception of new projects, the research of similar alternatives on the market and how we could improve it, the steps and processes required for efficient requirements gathering and functionalities definitions, the process of designing the architecture and the deployment of a system, and the prototyping of the final implementation, as well as their testing with potential clients and how their feedback along the way can be important to improve the final solution and add value to the development process. We also gained new insights into how to work as a team in a collaborative project, in particular following the agile methodologies, furthering our communication, teamwork and time management skills.

We also widened our experience with the technologies and frameworks used, either by increasing the knowledge of the technologies we had already worked with, such as Spring Boot, or exploring new technologies and solutions that we haven't had much experience with before, such as the implementation and integration of physical sensors or the use of message queues. The scale of the project also allowed us to understand better how the different parts of a software engineering system are interconnected and related to each other, and to understand how to better develop features and requirements that are essential to the use of an application, but that aren't visible for the clients and end users, such as the various algorithms we developed to automate some of the functionalities based on the users profile and their plant inventory.

Along the way, we had some difficulties, in particular caused by the time constraints of the project development, that meant we had to leave out some of the features initially planned for the application. These included, mainly, the user forum section, in which users could share their tips about the plants they've grown, the plant journalling section, and the inclusion of a timeseries database. However, this was also rewarding for us in the long run, as we understood that some of the expectations at the beginning of the project were a bit unrealistic with the time we had for the development, meaning we learned how to better prioritize different tasks, user stories and functionality requirements and to figure out which are really important for the initial proof of concepts of systems, and which can be done at a later stage of development, which will help us in any future projects we undertake.

For future work on this project, we would implement the mentioned forum functionality that we had to scratch from this development cycle. We would also expand on some of the features we have already implemented, potentially reworking them based on the feedback from the usability tests, such as the page regarding the management of divisions and the plants located in each of them. Novel features that would be interesting to implement would include the integration of a system for disease or plague detection on plants, as suggested by some of the testers, via the analysis of their current state and condition via photos, and the consideration of user's statistics to determine the likelihood of different plants showing some of these symptoms. It would also be interesting to include some sort of machine learning model to determine the likelihood of such diseases or symptoms appearing on plants given their characteristics and their conditions.

Performance testing would also be done, to ensure our architecture could handle a bigger and growing number of users, with the implementation of a reverse proxy server to handle load balancing and increased traffic, and a cache on the backend to ease the load on this component, if necessary. Better authentication and authorization mechanisms would be implemented to secure the REST API, with this needing to be done before a public release of the system. We would also update our algorithms, such as the determination of plant's difficulty, or the plant's suggestion algorithms, to consider the statistics and insights obtained from the users of the application over time.

Finally, we would contact experts on the botanical area, to figure out how to improve our current plant catalogue, which other parameters of plants could be monitored, and which other interesting functionalities could be added to a system in this field.

7. Bibliography

1. Gen Z Houseplant Ownership Stems from the Desire to Care for Something Alive - CivicScience. https://civicscience.com/gen-z-houseplant-ownership-stems-from-the-desire-to-care-for-something-alive/.

2. Houseplant Statistics in 2023 (incl. Covid & Millennials) | Garden Pals. https://gardenpals.com/houseplant-statistics/.

3. Lee, M. sun, Lee, J., Park, B. J. & Miyazaki, Y. Interaction with indoor plants may reduce psychological and physiological stress by suppressing autonomic nervous system activity in young adults: a randomized crossover study. *J Physiol Anthropol* **34**, (2015).

4. Darlington, A. B., Dat, J. F. & Dixon, M. A. The Biofiltration of Indoor Air: Air Flux and Temperature Influences the Removal of Toluene, Ethylbenzene, and Xylene. *Environ Sci Technol* **35**, 240–246 (2000).

5. Seven in 10 millennials consider themselves 'plant parents' | by SWNS | Medium. https://swns-research.medium.com/seven-in-10-millennials-consider-themselves-plant-parents-11ef0b34773c.

6. Flower Care - Apps on Google Play. https://play.google.com/store/apps/details?id=com.huahuacaocao.flowercare&hl=en&gl=US&pli=1.

7. Planta - Care for your plants - Apps on Google Play. https://play.google.com/store/apps/details?id=com.stromming.planta&hl=en&gl=US.

8. Pennisi, B. V. *Growing Indoor Plants With Success*. (2022).

9. O que é um banco de dados relacional (RDBMS)? | Google Cloud. *Google Cloud* https://cloud.google.com/learn/what-is-a-relational-database?hl=pt-br.

10. PostgreSQL Advantages: Benefits of Using PostgreSQL. https://www.prisma.io/dataguide/postgresql/benefits-of-postgresql.

11. Free Plant API | Houseplants,Garden,Trees,Flowers,Images & Data. *Perenual* https://perenual.com/docs/api.

12. Getting started | Trefle documentation. *Trefle* https://docs.trefle.io/docs/guides/getting-started/.

13. Find a Plant | North Carolina Extension Gardener Plant Toolbox. https://plants.ces.ncsu.edu/find_a_plant/.

14. Botânico UTAD, J. *Ficha da espécie Aloe arborescens Dados do registo na Flora Digital de Portugal Imagem de capa*. http://jb.utad.pt/termos.JardimBotânicoUTAD:http://jb.utad.ptJBnoFacebook:http://facebook.com/utadjb.

15. Ranzato Filardi, F. L. *et al.* Brazilian flora 2020: Innovation and collaboration to meet target 1 of the global strategy for plant conservation (GSPC). *Rodriguesia* **69**, 1513–1527 (2018).

16. Acalypha hispida (Chenille Plant). https://www.gardenia.net/plant/acalypha-hispida.

17. Adromischus Cristatus: Conheça Essa Interessante Suculenta. https://terragam.com/adromischus-cristatus.html.

18. Adromischus Festivus | Low-Growing Succulent | Planet Desert. https://planetdesert.com/products/adromischus-festivus.

19. Aequimea - Aechmea fasciata - Jardineiro.net. https://www.jardineiro.net/plantas/aequimea-aechmea-fasciata.html.

20. Hortipedia - Aechmea miniata var. discolor. https://en.hortipedia.com/Aechmea_miniata_var._discolor.

21. Aechmea ´Royal Wine´-Herbácea | Paisagismo Digital. https://paisagismodigital.com/item.aspx?id=101531-aechmea--%C2%B4royal-wine%C2%B4.

22. Garden Adventures: Aechmea Royal Wine. https://growerjim.blogspot.com/2011/01/aechmea-royal-wine.html.

23. Aechmea royal wine, catalogue of Bromeliad plants. http://www.lloydgodman.net/Photosynthesis/PHoToS/Aechm/royW.htm.

24. Meu Cantinho Verde: COLUMÉIA-MARMORE - (Aeschynanthus marmoratus). https://www.meucantinhoverde.com/2011/07/columeia-marmore-aeschynanthus.html.

25. NParks | Aeschynanthus pulcher. https://www.nparks.gov.sg/florafaunaweb/flora/1/2/1298.

26. Agave Victoriae Reginae Um Guia Desta Suculenta - Guia das Suculentas. https://guiadassuculentas.com/agave-victoriae-reginae-um-guia-desta-suculenta/.

27. Aglaonema ‘Silver King’ - Planthiza. https://planthiza.com/produto/aglaonema-silver-king/.

28. Chinese Evergreen (Aglaonema ’Silver King’) in the Aglaonemas Database - Garden.org. https://garden.org/plants/view/715791/Chinese-Evergreen-Aglaonema-Silver-King/.

29. Aglaonema ‘Silver Queen’ - Planthiza. https://planthiza.com/produto/aglaonema-silver-queen/.

30. FLOR BOLSA EM MINIATURA Alloplectus nummularia E a umidade melhor umidade para Flor bolsa em miniatura - Alloplectus nummularia? | Como cultivar plantas - Cuidados TIPS. http://world-population.net/house_plants/pt/k20k3.

31. Garden Guides | Life Cycle of an Aloe Vera Plant. https://www.gardenguides.com/137832-life-cycle-aloe-vera-plant.html.

32. Anthurium clarinervium: care & location - Plantura. https://plantura.garden/uk/houseplants/anthurium/anthurium-clarinervium.

33. Anthurium Hookeri Care Explained in Great Detail. https://plantophiles.com/plant-care/anthurium-hookeri/.

34. Asparagus densiflorus ‘Myersii’ (Plume Asparagus). https://www.gardenia.net/plant/asparagus-densiflorus-myersii.

35. FPS051/FP051: Asparagus densiflorus ‘Sprengeri’ Sprengeri Asparagus Fern. https://edis.ifas.ufl.edu/publication/FP051.

36. Asparagus Fern - Care, Growing, Watering, Propagation - Plant Index. https://www.plantindex.com/asparagus-fern/.

37. Asparagus falcatus | PlantZAfrica. https://pza.sanbi.org/asparagus-falcatus.

38. How to Grow and Care for Rex Begonias. https://www.thespruce.com/grow-rex-begonia-1902492.

39. Begônia-cerosa - Begonia semperflorens - Jardineiro.net. https://www.jardineiro.net/plantas/begonia-cerosa-begonia-semperflorens.html.

40. Billbergia, as bromélias mais fáceis de cuidar. https://revistajardins.pt/billbergia-as-bromelias-mais-faceis-de-cuidar/.

41. Billbergia zebrina Lindl. http://www.worldfloraonline.org/taxon/wfo-0000341530;jsessionid=8365B565DC174FEF3ED45366626D24D2.

42. Documentation - 6.2 - Hibernate ORM. https://hibernate.org/orm/documentation/6.2/.

43. 1. Introduction to Spring Framework. https://docs.spring.io/spring-framework/docs/3.2.x/spring-framework-reference/html/overview.html.

44. What is Spring Framework? | Definition from TechTarget. https://www.techtarget.com/searchapparchitecture/definition/Spring-Framework.

45. Spring Boot. https://spring.io/projects/spring-boot.

46. Part 1: RabbitMQ for beginners - What is RabbitMQ? - CloudAMQP. https://www.cloudamqp.com/blog/part1-rabbitmq-for-beginners-what-is-rabbitmq.html.

47. Spring Data JPA - Reference Documentation. https://docs.spring.io/spring-data/jpa/docs/current/reference/html/.

48. 16. Transaction Management. https://docs.spring.io/spring-framework/docs/4.2.x/spring-framework-reference/html/transaction.html.

49. Projections with JPA and Hibernate. https://thorben-janssen.com/projections-with-jpa-and-hibernate/.

50. Hibernate Second-Level Cache Explained | Hazelcast. https://hazelcast.com/glossary/hibernate-second-level-cache/.

51. How to Water and Feed Houseplants - Advice - Westland Garden Health. https://www.gardenhealth.com/advice/indoor-growing/how-to-water-and-feed-houseplants.

52. How to Correctly Fertilize All Your Plants. https://www.bhg.com/gardening/yard/garden-care/why-you-should-fertilize-plants/.

8. Annexes and Appendices

Appendix A – Repository and Other Interesting Links

- GitHub Organization: https://github.com/Sensing-My-Home
  - GrowMate: Solution Repository.
  - Project-Website: Promotional Website Repository.
  - Documentation: Project Documentation Repository.

- Project Documentation Website: https://sensing-my-home.github.io/Documentation/

- Promotional Project Website: https://sensing-my-home.github.io/Project-Website/

- API Documentation: https://documenter.getpostman.com/view/24060738/2s93m8xKtK

- Students@DETI Poster: Poster.pdf

- Promotional Video: https://www.youtube.com/watch?v=uP0-BJbaTyA

Appendix B – How to Run the Solution Locally

All the GrowMate's infrastructure is deployed in a Microsoft Azure's virtual machine which can be always accessed via the following base address: http://13.80.159.172/. Additionally, we generated an Android Package Kit file (APK) to enable the download of the mobile application on Android operating systems through a link which is available on our websites and GitHub repositories.

However, to set up the Growmate's infrastructure and run the app, there are a few steps to follow:

a) Setting up and running the API
   1. Clone the GrowMate's repository.
   2. Install Docker and Docker Compose.
   3. On the main folder terminal run "docker compose up".

b) Setting up the emulator and running the app
   1. Install Node.js and npm
   2. On the "mobileapp" folder terminal run "npm install" and "npm start"
   3. Running the app:
      a. On emulator:
         i. Install Android Studio and create a virtual device (https://developer.android.com/studio/install)
         ii. For better compatibility, create a *Pixel_3a_API_30_x86* device
         iii. Enter the option "a" on the terminal running "npm start" to open the emulator and the app
      b. On Expo Go app:
         i. Install Expo Go app on your phone
         ii. Enter the option "c" on the terminal running "npm start" to print a QR Code
         iii. Scan the QR Code with your phone

Appendix C – Team Contributions

Regarding our team contributions, we divided our efforts for this project as:

- André Butuc
  - Developed the frontend of the mobile application [70%].

- Artur Correia
  - Developed the frontend of the mobile application [30%].
  - Integrated the plant sensors with the backend [100%].
  - Deployed the mobile application [100%].

- Bruna Simões
  - Designed the mobile application [100%].
  - Developed the backend of the mobile application [30%].
  - Populated the plant database [70%].
  - Deployed the API [100%].

- Daniel Carvalho
  - Developed the backend of the mobile application [70%].
  - Performed usability tests [100%].
  - Populated the plant database [30%].

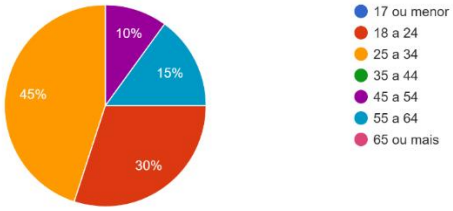
The team contributed equally to all the necessary deliverables, namely, slides for oral presentations, the report, poster, and video.

Overall contribution to the project (percentage wise):

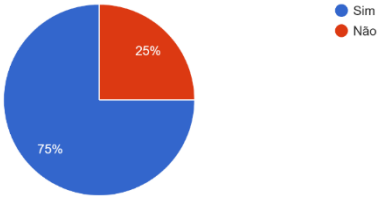
- André Butuc – 25%
- Artur Correia – 25%
- Bruna Simões – 25%
- Daniel Carvalho – 25%

Appendix D – Requirements Gathering Questionnaire – Questions and Results

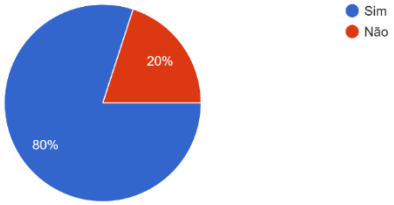
Qual é a sua faixa etária?
20 respostas



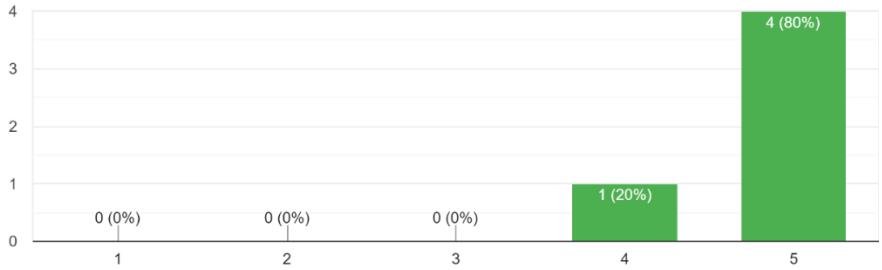
Já teve experiência a cuidar de plantas de interiores?
20 respostas



Estaria interessado em começar a cuidar de plantas de interior no futuro?
5 respostas

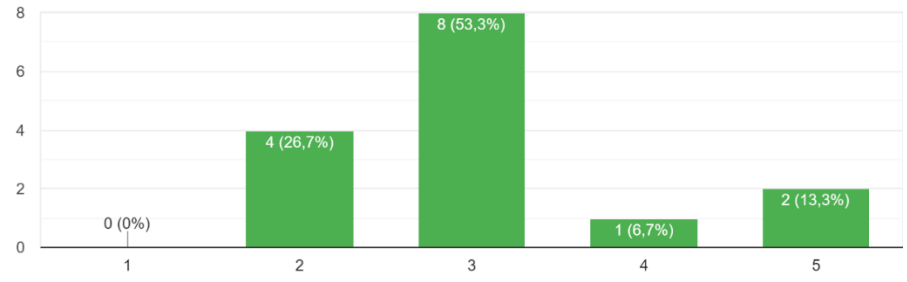


Caso algum dia pretenda crescer plantas, o quão útil seria uma aplicação móvel para monitorizar o crescimento e as tarefas para a manutenção destas?
5 respostas



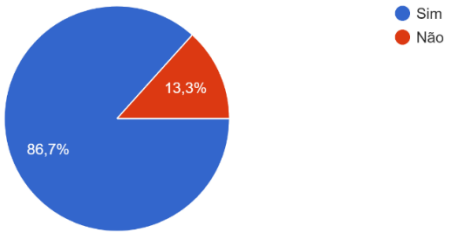
O quão experiente se considera no cuidado de plantas?

15 respostas



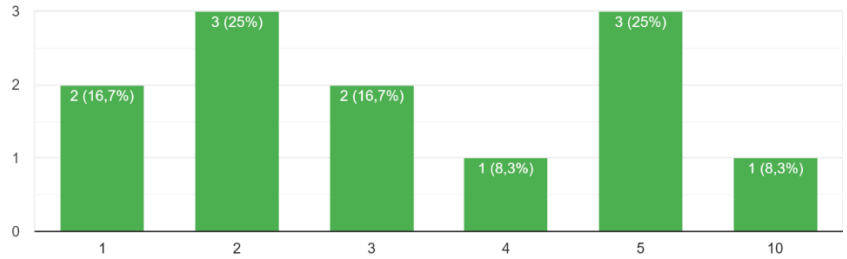
Alguma vez deixou morrer uma planta sua?

15 respostas



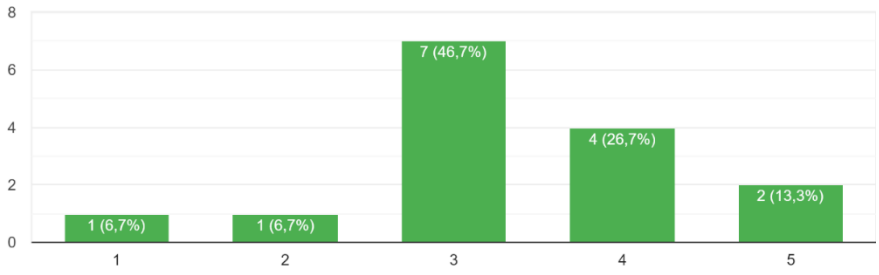
Caso já tenha morto plantas anteriormente, poderia dar uma estimativa de quantas?

12 respostas



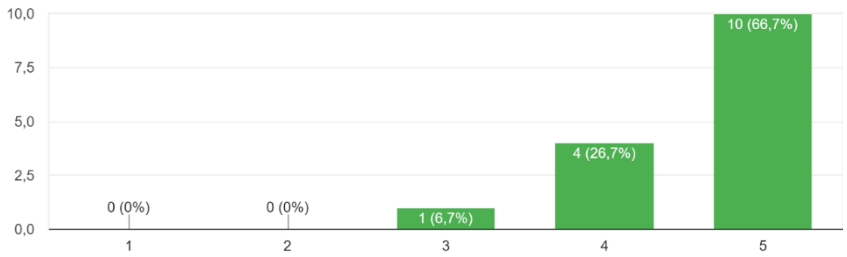
O quão dificil foi manter em mente todas as tarefas de manutenção das plantas?

15 respostas

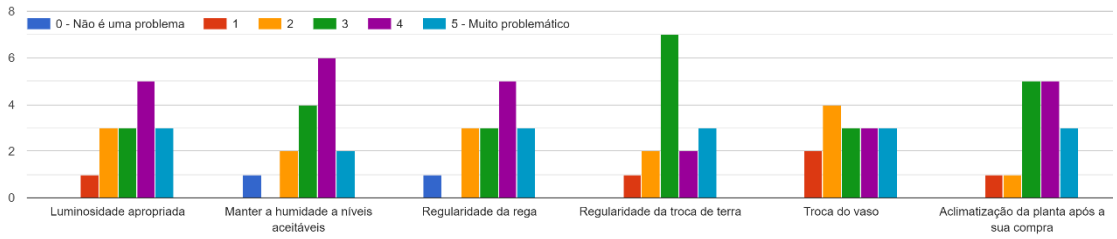


Na sua opinião, o quão útil seria uma aplicação móvel para monitorizar o crescimento das suas plantas e as tarefas para a manutenção destas?

15 respostas



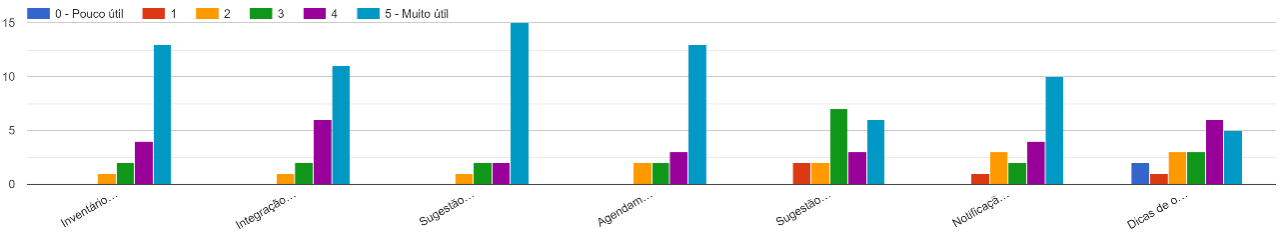
Avalie, segundo a sua experiência, as seguintes dificuldades no crescimento de plantas.



Já sentiu mais alguma dificuldade a cuidar das suas plantas?

- Lembrar-me de cuidar dela
- Necessidade de introdução de nutrientes extra para melhorar o crescimento da planta
- Controlo de pragas
- Saber que fertilizante comprar
- Dificuldade em diagnosticar doenças

Por favor, avalie cada funcionalidade baseada na sua utilidade.



Appendix E – Database Model Diagram

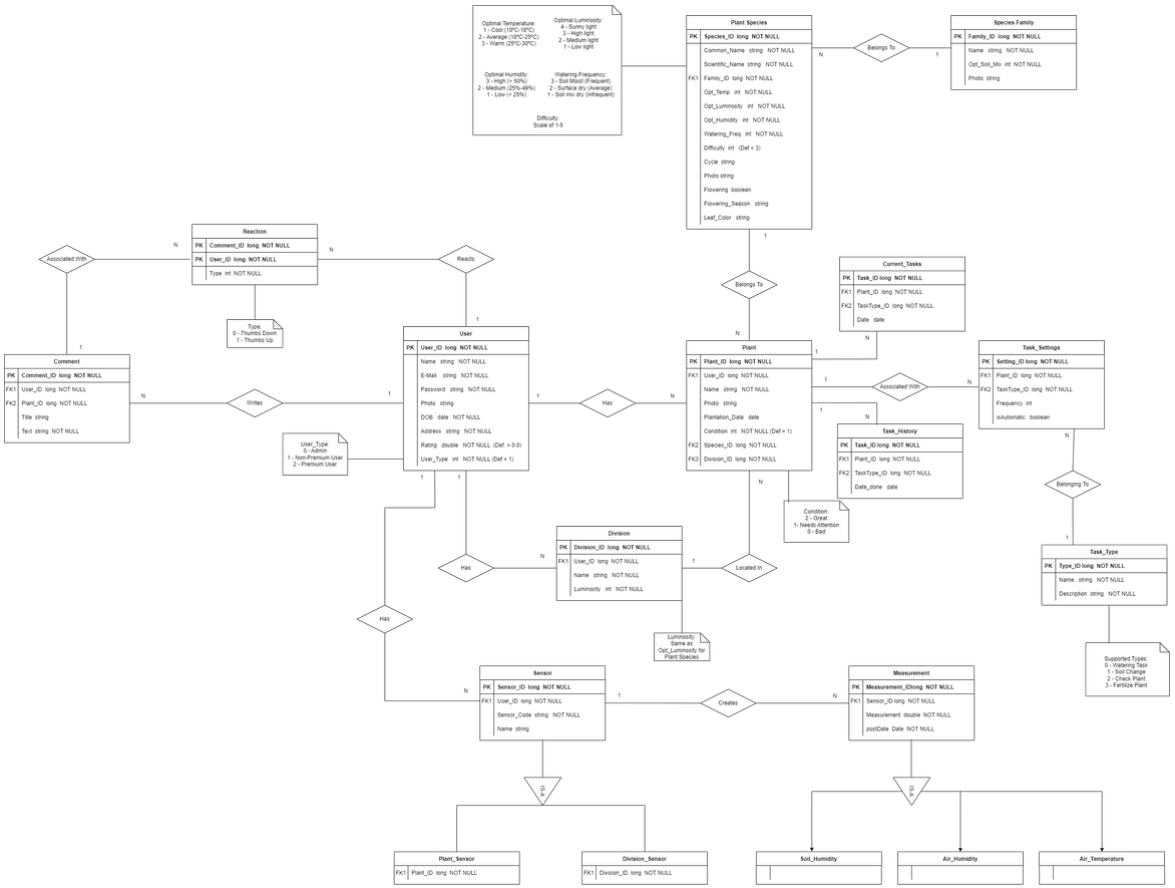
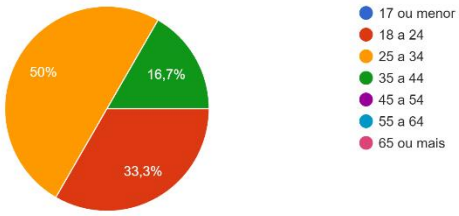


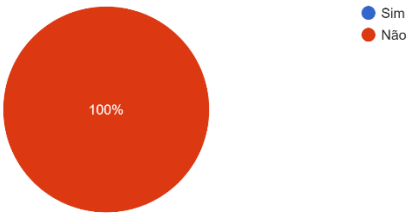
Figure 40 – Database Model Diagram, In case it's not visible, the diagram is available [here](#).

Appendix F – Usability Tests – Questions and Results

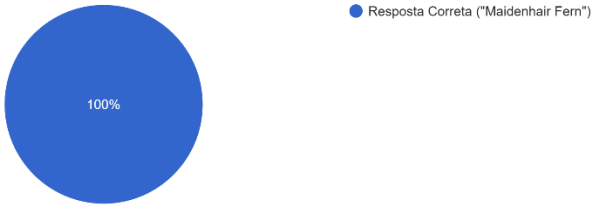
Qual a sua faixa etária?
6 respostas



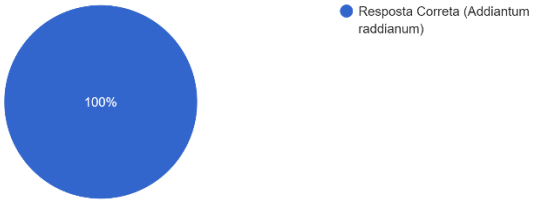
Alguma vez lidou com outras aplicações de gestão de plantas?
6 respostas



1. Qual o nome da espécie a que a Juliana pertence? Regista a opção que o utilizador escolheu.
6 respostas

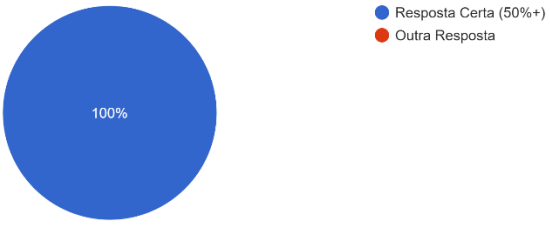


1.1 E qual o seu nome cientifico da espécie a que a Juliana pertence? Regista a opção que o utilizador escolheu.
6 respostas



1.2. Quais são os valores de humidade ótima para a Juliana?

6 respostas

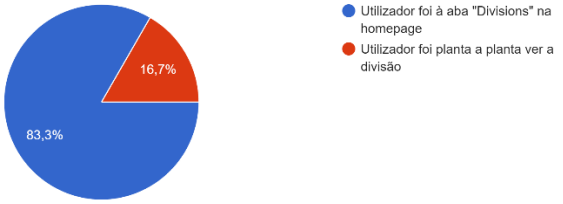


Conseguiu realizar a tarefa anterior (1)?

6 respostas

1. Quais são as plantas localizadas na "Living Room"? Qual o caminho escolhido pelo utilizador?

6 respostas

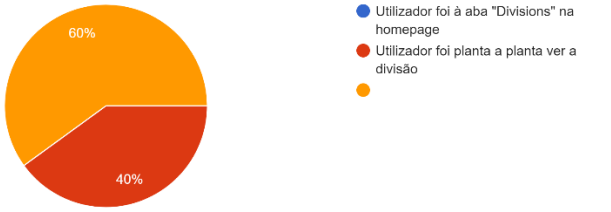


Conseguiu realizar a tarefa anterior (3)?

6 respostas

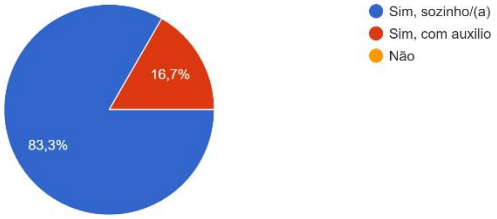
2. Muda a Beth do "Bedroom" para a "Living Room". Qual o caminho escolhido pelo utilizador?

5 respostas



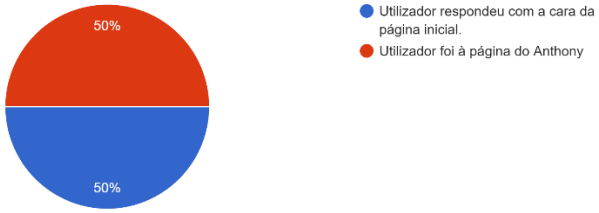
Conseguiu realizar a tarefa anterior (4)?

6 respostas



1. Como se sente o Anthony? Qual o caminho escolhido pelo utilizador?

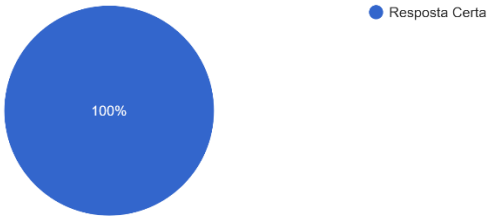
6 respostas



Conseguiu realizar a tarefa anterior (1)?

6 respostas

2. Qual o valor atual da humidade do ar do Anthony? Regista a opção que o utilizador escolheu.
6 respostas



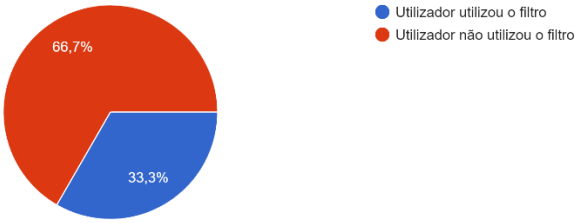
Conseguiu realizar a tarefa anterior (2)?
6 respostas

3. Qual o valor da humidade do solo do Anthony às 19h do dia 23 de Maio? Regista a opção que o utilizador escolheu.
6 respostas

Conseguiu realizar a tarefa anterior (3)?
6 respostas

1. Quantos sensores de plantas estão associados à conta? Qual o caminho escolhido pelo utilizador?

6 respostas

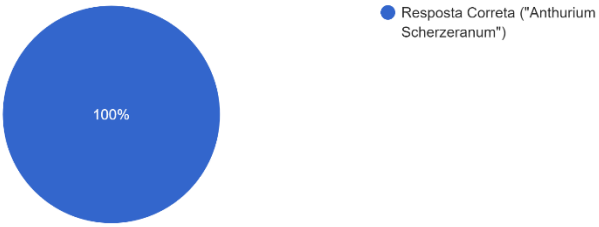


Conseguiu realizar a tarefa anterior (1)?

6 respostas

2.1. Qual o nome cientifico da "Flamingo Flower"? Regista a opção que o utilizador escolheu.

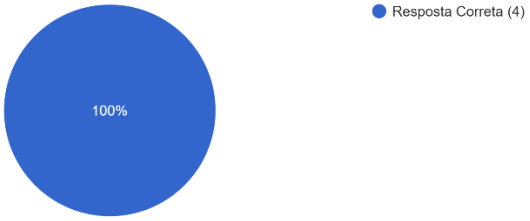
6 respostas



Conseguiu realizar a tarefa anterior (2)?

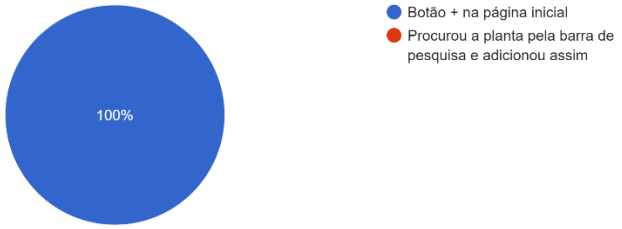
6 respostas

3.1. Qual a dificuldade de crescer estas plantas? Regista a opção que o utilizador escolheu.
6 respostas



Conseguiu realizar a tarefa anterior (3)?
6 respostas

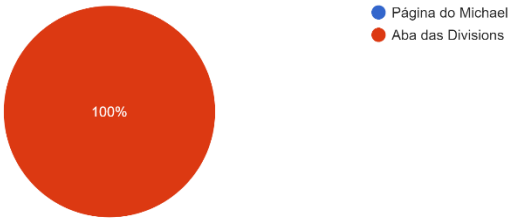
1. Adiciona uma nova planta da espécie "Powder Puff", chamada "Michael", ao teu inventário Qual o caminho escolhido pelo utilizador?
6 respostas



Conseguiu realizar a tarefa anterior (1)?
6 respostas

2. Regista que o "Michael" está localizado na "Cozinha". Deves criar esta divisão. Qual o caminho escolhido pelo utilizador?

6 respostas

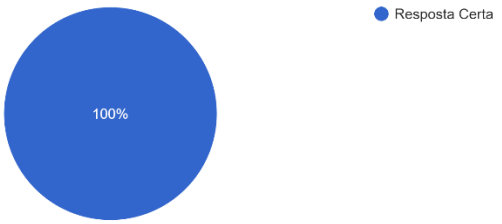


Conseguiu realizar a tarefa anterior (2)?

6 respostas

3. Quais são as tarefas calendarizadas para junho? Regista a opção que o utilizador escolheu.

6 respostas

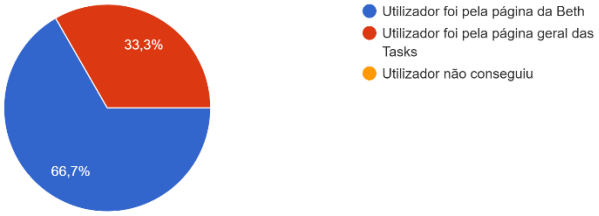


Conseguiu realizar a tarefa anterior (3)?

6 respostas

4. Altera a tarefa "Water Michael" de modo automático para modo manual, e define a sua periodicidade para ser de 3 em 3 dias. Qual o caminho escolhido pelo utilizador?

6 respostas



Conseguiu realizar a tarefa anterior (4)?

6 respostas

Satisfação geral