UNIVERSIDAD DE MÁLAGA

E.T.S. INGENIERÍA INFORMÁTICA
UNIVERSIDAD DE MÁLAGA

Graduado en Ingeniería del Software

Aplicación acompañante para jardines botánicos utilizando balizas IoT con telemetría de servidor.

Botanic garden companion app using IoT beacons with server telemetry.

Realizado por
Adrián Salas Troya

Tutorizado por
Mónica Pinto Alarcón

Departamento
Lenguajes y Ciencias de la Computación
UNIVERSIDAD DE MÁLAGA

MÁLAGA, junio del 2024

Abstract

This final year dissertation presents the analysis, design and development of an Android application for a botanical garden using IoT beacon technology. The proposed application aims to transform and enhance the visitor experience by offering personalized routes, gamification, and interactive content through Bluetooth Low Energy (BLE) beacons carefully placed around the garden.

Botanical gardens, as centers of conservation and environmental education, play a crucial role in the protection of plant species. Thanks to these places, environmental awareness is promoted among the general public. By incorporating the mix of modern technologies such as mobile devices with nature, we bring people closer and encourage them to learn more about nature and bio-conservation.

Furthermore, this project allows the workers of the botanic garden a way to know the visitors' behaviors and patterns by gathering telemetry. Analyzing the flow of visitors, congestion points and ways to alleviate overcrowding can be detected, and plan the placements of plants, the paths between the different zones and where would be better to have bathrooms, water fountains, exhibitions and more.

Keywords: **Beacon, IoT, BLE, Bluetooth, Nature**

Resumen

Este trabajo de fin de grado presenta el análisis, diseño y desarrollo de una aplicación Android para un jardín botánico utilizando tecnología de balizas IoT. La aplicación propuesta tiene como objetivo transformar y mejorar la experiencia del visitante ofreciendo rutas personalizadas, ludificación y contenido interactivo a través de balizas Bluetooth Low Energy (BLE) cuidadosamente colocadas alrededor del jardín.

Los jardines botánicos, como centros de conservación y educación ambiental, desempeñan un papel crucial en la protección de las especies vegetales. Gracias a estos lugares se fomenta la concienciación medioambiental entre el público en general. Al incorporar la mezcla de tecnologías modernas, como los dispositivos móviles, con la naturaleza, acercamos a la gente y la animamos a aprender más sobre la naturaleza y la bioconservación.

Además, este proyecto permite a los trabajadores del jardín botánico conocer los comportamientos y patrones de los visitantes mediante la recogida de telemetría. Analizando el flujo de visitantes, se pueden detectar puntos de congestión y formas de aliviar la masificación, así como planificar la colocación de plantas, los caminos entre las distintas zonas y dónde sería mejor tener baños, fuentes de agua, exposiciones y mucho más.

Palabras clave: **Baliza, IoT, BLE, Bluetooth, Naturaleza**

Contents

1 Introduction

This section outlines the motivation and objectives of this project and describes the structure of the document.

1.1 Motivation

A Botanical Garden provides a perfect space for research, education and leisure for the population, but lacks a modern visitor experience. Identifying this need, this project aims to **improve the experience** through the use of a mobile application with the benefits of the Internet of Things (IoT), specifically IoT beacons. Visitors will be able to obtain more information during their visit and personalise their experience through planned routes and gamified activities thanks to **beacons** placed throughout the garden, while surrounded by the flora of the Botanical Garden.

In addition, the server would capture **telemetry** data in the hope of obtaining important information about visitor behaviour. Through telemetry, the Botanic Garden could improve future planning and resource allocation by analysing data on visitor interactions and patterns.

1.2 Objectives

The objectives of the project are as follows:

1. **Develop an Android app:**
   Develop a user-friendly android application that incorporates the use of beacons to improve the visitor experience. The user will be notified of nearby plants or sites of interest.

2. **Develop a server infrastructure:**
   Create a server infrastructure capable of collecting telemetry from the user and their interactions with the beacons, as well as providing the basic services for the correct

functioning of the mobile app. The server must be able to scale horizontally, be secure while maintaining data integrity and reliability.

3. **Develop an administrator web panel:**
   Develop a basic administrator panel for the management and administration of the application by a worker. In addition this panel will show statistics of the collected telemetry.

4. **Adopt clean architecture principles:**
   Design the whole system taking into account clean architecture principles, to ensure modularity, scalability and maintainability of the software. The use of different layers with clear responsibilities will facilitate maintenance and future enhancements by other developers.

5. **Test the system:**
   Ensure the application undergoes different types of tests to achieve a certain degree of robustness and detect possible bugs early.

6. **Facilitate future maintenance and enhancements:**
   Develop the application with a focus on maintainability and extensibility, using standardised programming, documentation and version control practices. Aiming for the system to be robust and easy to evolve.

1.3 Structure of the document

1. **Introduction:**
   Section explaining the motivation for this project, its objectives and the methodology to achieve them.

2. **Methodologies:**
   Section about the project management methodology employed and the guidelines followed during the development.

3. **Technologies and Tools:**
   A brief description of the technologies chosen for the development of the project is

given, including the beacon technology itself, programming languages, libraries, frameworks, database and development tools.

4. **Requirements and Architectural Design:**
   List of functional and non-functional requirements set prior to the development of the application. And a general look at the architecture and deployment of the application.

5. **Server and Database:**
   This section details the server-side part of the project. The architectural designs of the server are included along with the decisions made to ensure high scalability and performance.

6. **Android Application:**
   This part includes the UI/UX sketches of the Android application, as well as the final design. It also includes how to detect and interact with beacons.

7. **Administrator Web Panel:**
   This part includes the UI/UX sketches of the web panel. This dashboard allows the administrator to create, update and the delete the different features of the application and view statistics of the collected telemetry.

8. **Conclusions and Future Lines of Research:**
   It summarises the main conclusions and results of the project and suggests possible future developments that could be made from this project.

2

Methodology

This section deals with the methodologies used during the project. How the project was organized and the guidelines followed during development.

2.1 Organization and management

Kanban is an agile methodology system widely used in software development. The etymology of the word comes from Japanese *sign* referring to the system of cards used in this methodology. This methodology, by making use of card boards, you can **visually see the status of the project**, the pending tasks, the tasks in progress and the completed tasks. In addition, there are many virtual implementations of the Kanban board that are free to use. Specifically for this project, **Kanboard** has been used as it is **open-source** and can be **self-hosted**.

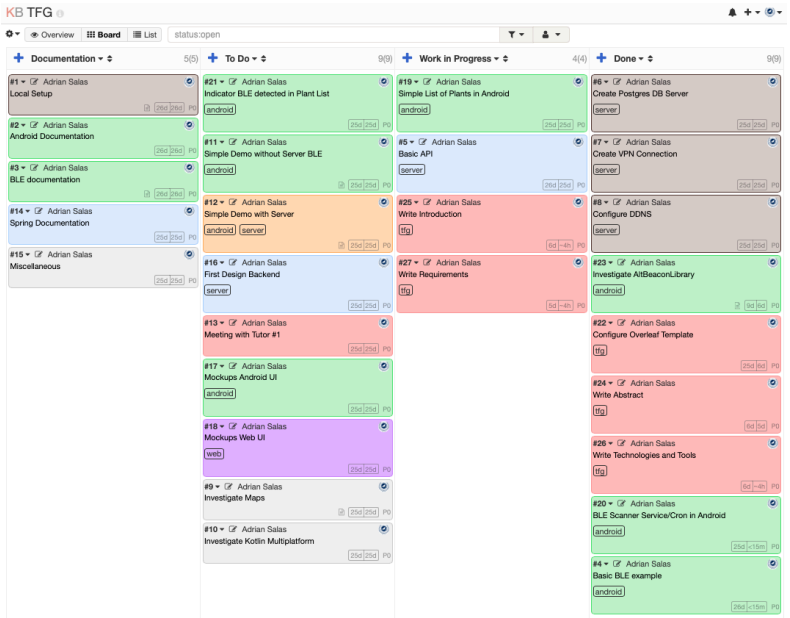

Figure 1: State of the kanban board at the beginning of the project.

With this methodology, a **visual control** can kept of both the status of the project and pending tasks while supporting the **addition of new tasks** that may arise.

2.2 Software development guidelines

Guidelines to follow in the development:

- **Domain-Driven Design** (DDD).

  An approach to software development in which the problem is focused on a "domain" layer, making the software produced **less dependent on technologies** and **easier to maintain** or adapt to future requirements. There are several approaches, including Alistair Cockburn's "Hexagonal Architecture", Jeffrey Palermo's "Onion Architecture" and Robert C. Martin's "Clean Architecture"[1].

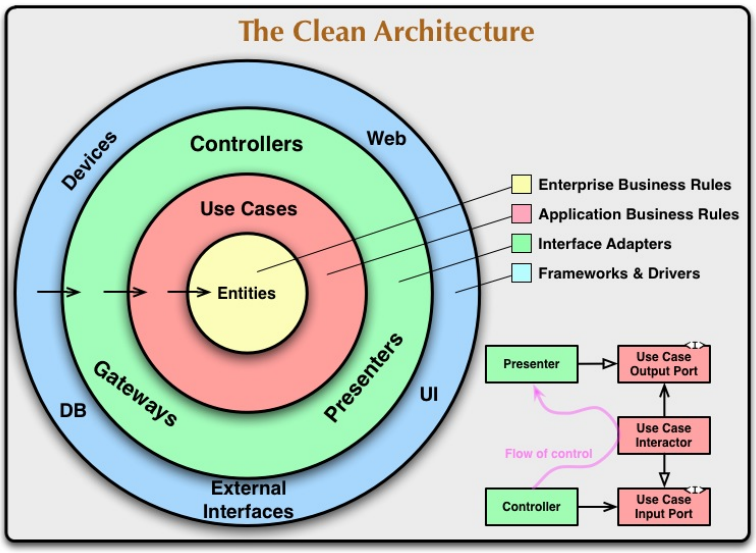

Figure 2: Clean architecture. *Extracted from [1].*

As can be seen in Figure 2, the **clean architecture** consists of the dependencies of the software components going to the interior of the architecture where the use cases and entities are located. This layer called core or **business rules does not depend on higher layers**. Isolating business logic from technology details.

A simpler way to visualize this flow of dependencies is seen in the **hexagonal architecture**. In figure 3 the interfaces are represented by red rectangles, and the implementations by green rectangles. For the controller to call the business logic it has to make use

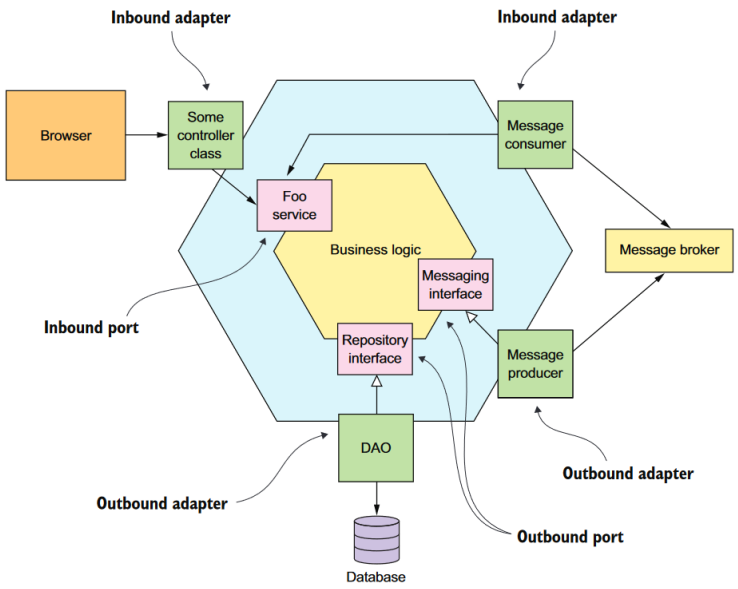

Figure 3: Hexagonal architecture example. *Extracted from [2].*

of an **inbound port**, the business logic does not have dependencies with the external layer, but it has defined what it needs to work (**outbound ports**). These outbound ports are in turn implemented by external layers. This allows for example to **easily test** the business logic without depending on the technologies chosen for the project. Since it depends on an interface and not on an implementation, **switching** from a SQL **database** to a NoSQL database **can be achieved without changing the business logic**.

- **Movel View Controller** (MVC) [3].

  Software architectural pattern used in User Interfaces, that separates the application into three components (see Figure 4): the **Model**, the **View** and the **Controller**. This separation defines a clear separation of concerns, making the application easier to develop, understand and maintain.

  - **Model**: Represents the data of the application. It encapsulates the state of the data to be represented.
  - **View**: Represents the interface itself, it is the element responsible for presenting the Model data to the user.
  - **Controller**: It is the logic of how the View must respond to the user's actions, it is the element in charge of updating the Model and orquestrate the functionality.

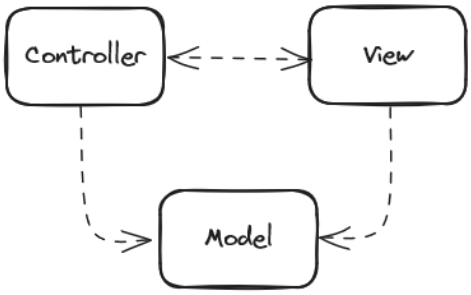

Figure 4: Model View Controller pattern. *Extracted from [3].*

- **Twelve-Factor App** [4].

The Twelve Factors

I. Codebase
One codebase tracked in revision control, many deploys

II. Dependencies
Explicitly declare and isolate dependencies

III. Config
Store config in the environment

IV. Backing services
Treat backing services as attached resources

V. Build, release, run
Strictly separate build and run stages

VI. Processes
Execute the app as one or more stateless processes

VII. Port binding
Export services via port binding

VIII. Concurrency
Scale out via the process model

IX. Disposability
Maximize robustness with fast startup and graceful shutdown

X. Dev/prod parity
Keep development, staging, and production as similar as possible

XI. Logs
Treat logs as event streams

XII. Admin processes
Run admin/management tasks as one-off processes

Figure 5: The Twelve-Factor App [4].

This methodology help the development of a **software-as-a-service application**, by emphasizing practices such as **dependency isolation** and **configuration management**. This way the development and deployment in different environments becomes easier. Figure 5 shows the twelve principles of this methodology. An application that follows these principles must be **configurable** both by files and by **environment variables**. It must be **horizontally scalable**, for that reason it must not have an internal state, but be **stateless**. It must treat external services such as databases as **configurable attached resources**. The application must have the minimum differences between how it is executed during the development and how it is executed in production.

- **Clean Code**.

  Guidelines to generate a code that is **easy to read** by third parties. Emphasising on trying to **avoid generating more complexity than necessary**. These principles are outlined in John Ousterhout's book, "A Philosophy of Software Design" [5] and in Robert C. Martin's book, "Clean Code: A Handbook of Agile Software Craftmanship" [6].

  The main idea of these books and guidelines is to try to **reduce the complexity of the software** and to facilitate the reading of it by third parties. It is to look for the simplest possible code and design to make it **easier to read and test**. Both authors argue that **working code is not enough**, we must try to achieve a code that is **easy to maintain and easy to use**. Examples of principles to achieve clean code are:

  - **Meaningful names.** Use descriptive names for naming variables, functions and classes.
  - **Avoid side effects.** A function should do only what is expected to do.
  - **Keep It Super Simple (KISS)**. Emphasize simplicity to improve readability and maintainability of the code.
  - **Single Responsibility Principle (SRP).** Each function or class should have only one responsibility.
  - **Avoid magic numbers.** Use constants to define what the numbers mean.
  - **Prefer immutability.** Make objects immutable wherever possible.

3

Technologies and Tools

This section describes the technologies and tools used in the development of the project. It includes an overview of the beacon technologies, server technologies, mobile technologies, web technologies and various development tools used.

3.1 Beacon technologies

Figure 6: Different beacons from different vendors. *Extracted from [7].*

3.1.1 Description

Beacons (see Figure 6) are very **simple IoT devices** [7], possibly among the simplest Bluetooth devices in existence. They are small devices between 1cm and 15cm (depending on the size of the battery). With the introduction of Bluetooth 4.0, the concept of **Bluetooth Low Energy** (BLE) was introduced, where devices transfer less information but are much more energy efficient. In particular beacons make use of **broadcast** to transmit their **UUID** to all nearby devices and do not pair with any specific device. This allows a single beacon to be detected by thousands of devices at once.

3.1.2 Standards

There are several standards inside the beacon ecosystem, the three most famous protocols are:

1. **iBeacon:**

   Apple was the forerunner of the beacon technology and the use of iBeacons is natively supported on iOS since iOS 7 [8]. There are 3 main values in its protocol as described in Table 1.

Field	Size	Description
UUID	16 bytes	Application developers should define a UUID specific to their app and deployment use case.
Major	2 bytes	Further specifies a specific iBeacon and use case. For example, this could define a sub-region within a larger region defined by the UUID.
Minor	2 bytes	Allows further subdivision of region or use case, specified by the application developer.

Table 1: iBeacon fields. *Extracted from [8].*

2. **Eddystone:**

   Google developed another protocol called Eddystone [9] for beacons that is quite different from the one designed by Apple. However, Google stopped supporting it in 2018 [10]. It is composed of several parts:

- **Eddystone-UID:** Composed of 10 bytes to identify the namespace and 6 bytes to identify the beacon itself.
- **Eddystone-EID:** Similar to the Eddystone-UID but encrypted with AES 8-bytes with time rotations.
- **Eddystone-URL:** 17 bytes reserved to store an URL, this allows to open specific web pages upon detection of this beacon.
- **Eddystone-TLM:** Provides data on temperature, battery, uptime and number of transmissions made.

3. **Altbeacon:**

   Radius Networks designed this protocol [11] in 2014 because at that time the only protocol was Apple's protocol and it was not opensource. The goal of AltBeacon is to be an open and interoperable specification. It allows a little bit of customization by the vendor in its fields, see Figure 7.

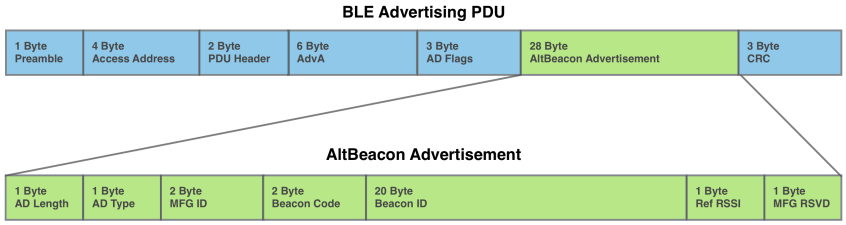

Figure 7: AltBeacon protocol format. *Extracted from [11].*

3.1.3 Use cases

Beacons have a multitude of uses, for example:

- **Locating data accurately**. Especially indoors the use of beacons gives much more accurate location of a person inside a building than a conventional GPS system.
- **Improving the shopping experience**. In shops, beacons can be used to provide product information and offer offers to encourage purchases.
- **Collecting telemetry**. As the behavior of people/customers in a shop or building.
- **Locating objects**. For example Apple's **AirTag** uses this technology.

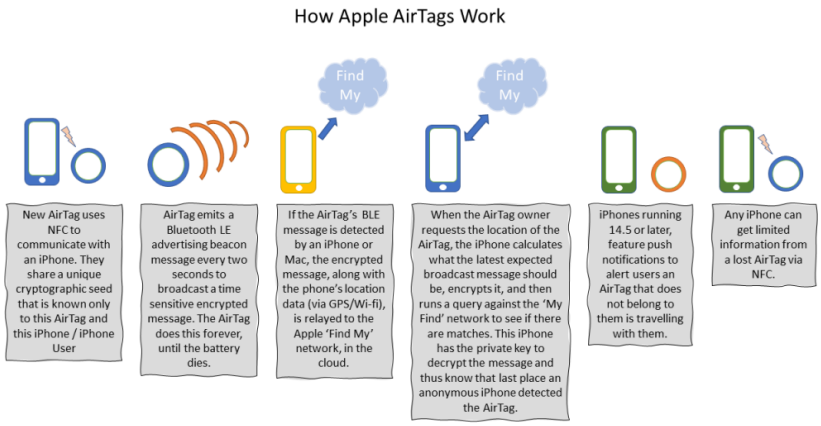

Figure 8: How do Apple AirTags work? *Extracted from [12].*

3.1.4 Challenges

There are different challenges and lines of research currently around this technology [13], the following three can be highlighted:

- **Challenge 1:**
  Currently the applications on the market are only prepared to support one type of beacon, with three different standards it is difficult to develop applications that simultaneously serve all three protocols.

- **Challenge 2:**
  Due to the simplicity of the protocol, security is not its strongest point. Beacons are announced in broadcast without any kind of security, making their UUIDs capturable and replicable by an external agent.

- **Challenge 3:**
  BLE signals are too unstable, and there is too much interference in the average environment to accurately determine the distance to a beacon. Approximate distances can be calculated, but with a significant margin of error.

3.2 Server technologies

Spring Boot [14] was chosen for this project due to the prior experience with the framework. It is one of the most famous and used frameworks in the industry to make **microservices**. It has an extensive collection of libraries for use with **JVM languages**, which make development easier and faster.

Once the framework has been chosen, another of the key pieces for the server would be to choose the database. For this, **PostreSQL** [15] has been chosen, it is free, open source and one of the most used. Although with **Spring Data abstractions** this database could be changed to another without any major impediment. As section 4 of **The Twelve-Factor App** [4] says, the database **should be treated as an attached resource**, and should be easy to change to another one.

To handle possible future **database migrations**, Flyway [16] is included to facilitate database creation/updates.

Traditionally web servers, such as those used in Java EE, made use of **one thread per request** (see Figure 9). They handled each request with a thread and that thread was used until the response was returned. This can be very **inefficient under heavy load**. When a server does **I/O** operations such as file reads/writes or network calls such as database calls, the thread waits idly until the **blocking operation** finishes. This means that the entire thread pool can be consumed, causing bottlenecks and a sudden drop in response times.

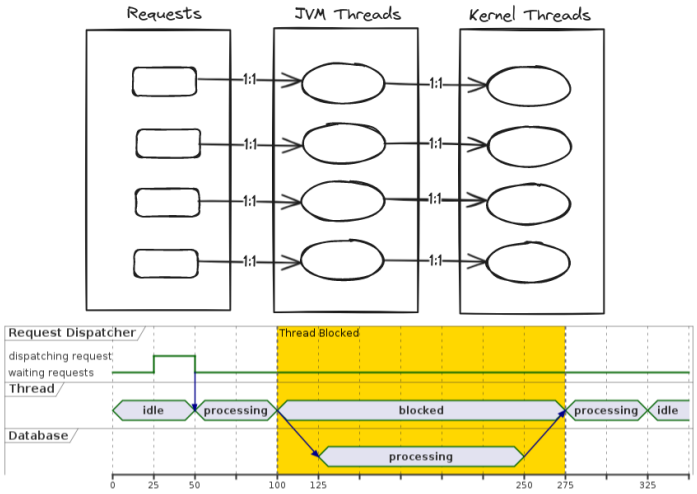

Figure 9: Each request is mapped to a kernel thread, blocking it when waiting I/O.

To solve all the problems caused by blocking APIs, **Spring WebFlux** [17] has been chosen, so that requests are handled in a **non-blocking** way. Webflux is based on Project Reactor [18], which is a Java library for working with **Mono/Flux** (see Figure 10). **Reactive programming** makes it easier to work with asynchronicity, which allows the server a greater flow of requests since a thread will not be exclusively responsible for a single request. A phrase from the Kotlin [19] documentation defines reactive programming very well as: *"everything is a stream, and it's observable"*.

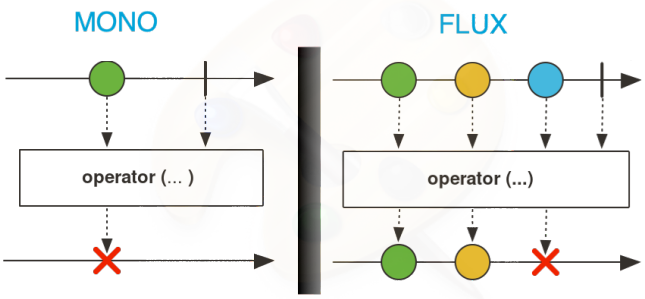

Figure 10: Mono and Flux visualization. *Extracted from [18].*

With Spring MVC, JDBC is used directly to connect to the database, but for this project, WebFlux will be used, requiring the use of **R2DBC** [20] (Reactive Relational Database Connectivity), which provides greater ease in working with data flows without blocking.

Working with this type of programming can be challenging and difficult. To facilitate development, instead of using Mono/Flux directly, **Kotlin** [19] will be used since it offers an alternative way of working with asynchronicity with **coroutines** (see Figure 11). These allow you to work with **non-blocking** code in a **more readable and simple** way than what Project Reactor [18] proposes. In addition, Kotlin is one of the most used languages for making Android applications.

3.3 Mobile technologies

Kotlin [19] is used to develop the Android [21] application to avoid using a language other than that of the server. In this way the cognitive load for the development of this project is reduced. As mentioned before, **Kotlin coroutines** are a very powerful tool for doing **non-blocking tasks** such as a UI, and they are also **cancellable**, which allows for a **fluid user**

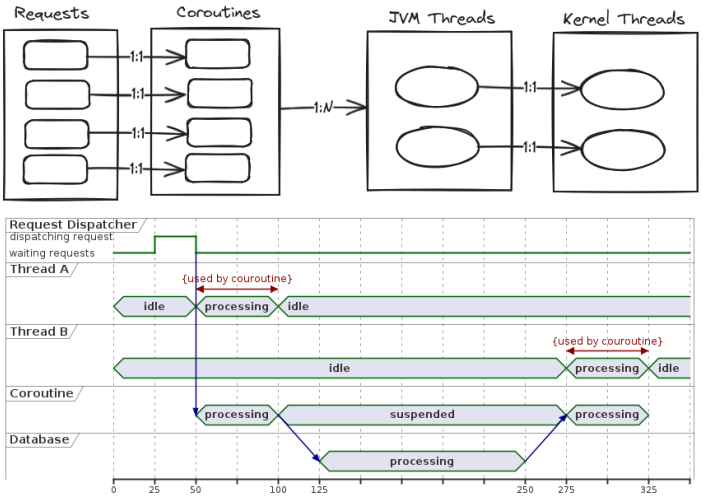

Figure 11: A coroutine can be suspended, freeing the thread to do other tasks.

experience.

To make the UI of the application itself, **Jetpack Compose** [22] will be used. This library is quite recent and allows the UI to be defined **declaratively**. Thanks to Jetpack Compose, **reusable and dynamic components** can be defined, and **Kotlin Multiplatform** [23] is based on Jetpack Compose, which allows **future development of an iOS application** by reusing a large part of the code.

In order to detect the beacons themselves, the **AltBeacon** [24] library will be used, this library greatly simplifies the **detection** of these **BLE devices**. The other library has e**xtra functionalities** such as calculating the approximate distance to the beacons, which, as mentioned previously, is one of the challenges of the beacons.

To connect to the server and make requests, **Retrofit** [25] will be used, a library widely used in Android to make **HTTP calls** to the server. This facilitates the use of the defined Restful APIs. It also has great integration with Kotlin.

3.4 Web technologies

Angular [26] has been chosen for the development of the website. Developed and maintained by Google, it is one of the most used frameworks for developing web applications. Thanks to the use of this framework, the website can be organized into **reusable components**, as well as use **component libraries** that facilitate development such as Angular Material or ngx-charts.

Angular also comes integrated with **reactive libraries** (RxAngular).

Another advantage of using Angular is that by default it uses **Typescript** [27], a JavaScript superset developed by Microsoft that adds typing. Using a strongly **typed language** on the server such as Kotlin, JavaScript development can be chaotic and prone to having unexpected bugs as a result of not having types, the use of a typed language helps detect problems at **transpilation time** and not at execution time.

3.5 Development tools

Git [28] is a distributed **version control** system, it was created by Linus Torvalds and is one of the most used tools for developers. Thanks to the way of controlling changes, it allows you to have a commented history of the entire evolution of a project from its beginning.

One of the most famous **hosting** servers is **Github** [29] where many open-source projects such as Linux itself are hosted. One of the features offered includes extras apart from Git itself, such as a continuous integration and continuous deployment (CI/CD) system, called **Github Actions**, that has been used to automatically execute unit tests.

To write the code itself, different Integrated Development Environments (IDEs) have been used, such as **Android Studio** [30] for the mobile application, **Visual Studio Code** [31] for the web application and **IntellIJ IDEA** [32] for the server. In addition, **DBeaver** [33] has been used to manage the database.

Docker [34] has been used as a tool for containers, and **Nginx** [35] has been used as a load-balancer, reverse proxy and web server. Together with **Cloudflare** [36] as a DNS server.

For diagram tools such as UML, **PlantUML** [37] and **Excalidraw** [38] have been used, both open source and free. All the diagrams in this document have been made with these two tools.

For the task management part and controlling both the times and the scope of the project, **Kanboard** [39], also open source, has been used to apply the Kanban methodology.

4

Requirements and Architectural Design

This section details the requirements, both the functional (FR) and non-functional (NFR), established prior to the development process. It also provides an overview of the system's architectural design.

4.1 Functional requirements

- **FR-1 Show relevant info when approaching a plant or interest area:**
  The application shall display relevant information when the user approaches plants or areas of interest using beacon technology.

- **FR-2 Gather telemetry about visitor behavior:**
  The application should collect data on the visitor's behavior within the botanical garden, in order to later analyze the collected data.

- **FR-3 Botanic Garden Map:**
  Users should be able to consult the botanical garden map from the mobile application.

- **FR-4 Recommended Routes through the Botanic Garden:**
  Administrators will be able to upload routes recommended by the botanical garden, and users will be able to consult and follow these routes from the mobile application itself.

- **FR-5 Custom Routes generated by the plants the user wants to see:**
  The application must allow users to generate customised routes to visit the plants that are of most interest to the user. The application should be smart enough to generate these routes automatically.

- **FR-6 Near plants scanner:**

  The application shall have a nearby plant scanner, this feature shall be able to run in the foreground.

- **FR-7 Plant search:**

  The application will have a search engine for existing plants within the botanical garden.

- **FR-8 Flowering Plants / Blooming Plants:**

  The application will show the plants that are in bloom, so users can easily find out which plants are in bloom.

- **FR-9 News / Alerts in the Botanic Garden:**

  The application should have a section for news and/or alerts related to events, offers, exhibitions or maintenance tasks of the Botanical Garden.

- **FR-10 Gamification:**

  The application will incorporate some element of gamification to offer a different experience to the users, this element of gamification must be based on beacon technology.

- **FR-11 Administrator Web Panel:**

  Allow administrators to perform Create-Read-Update-Delete (CRUD) operations over the different system elements.

  - **FR-11.1 CRUD of Beacons:**

    Allow administrators to manage IoT beacons, including adding, editing, or removing them from the system.

  - **FR-11.2 CRUD of News:**

    Allow administrators to manage news/alerts, including adding, editing, or removing them from the system.

  - **FR-11.3 CRUD of Recommended Routes:**

    Allow administrators to manage recommended routes, including adding, editing, or removing them from the system.

  - **FR-11.4 Activate/Deactivate gamified activities:**

    The administrators should be able to opt-in / opt-out the gamified activities.

- **FR-11.5 Statistics visualizer:**
  The administrators should be able to view and analyze the data recollected by the telemetry.

4.2 Non functional requirements

- **NFR-1 Performance:**
  The application should respond smoothly and not take excessive time to calculate or retrieve data.

- **NFR-2 Security:**
  The application should be secure and protect user data, taking into account the most common OWASP recommendations.

- **NFR-3 Usability:**
  The application should be easy to use by a wide range of users of various ages and different levels of technological knowledge.

- **NFR-4 Scalability:**
  The application must be able to scale to a larger number of users than originally intended and have methods to evolve without having a strong impact on performance.

- **NFR-5 Reliability:**
  The application should be able to work in as many cases as possible, minimizing down-time and errors.

- **NFR-6 Compatibility:**
  The application should try to cover as many devices as possible, both the mobile application, the web administrator panel and the server should be able to work on a variety of devices.

- **NFR-7 Maintainability:**
  Development should follow good development practices within the industry to ensure the maintainability of the code and to allow for future enhancements.

- **NFR-8 Documentation:**

  The completion of this work should generate the necessary technical and manuals documentation.

4.3 Architectural design

The proposed architecture (see Figure 12) consists of an **Android application** and a **web page**, which interact with the backend services through a single point, this **API Gateway** is not limited in technology or requirements, nor in responsibilities. This leaves open the **different implementations** that could be carried out. It is advisable that you have a load-balancer so that **backend services** can be **replicated**. A message queue could be incorporated to handle events.

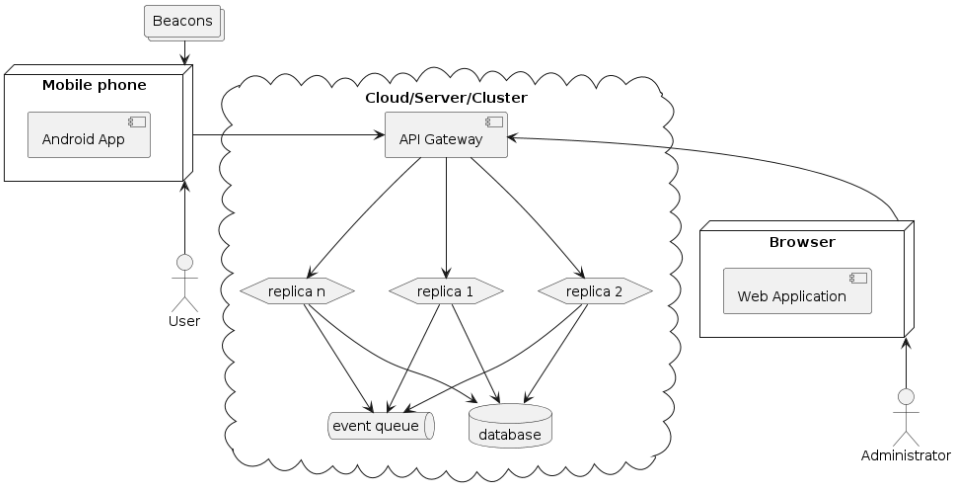

Figure 12: General architectural design for the application.

The backend service is designed to be **stateless**, allowing it to be replicated for greater **scalability** and **availability**, being able to optimize system resources if desired. It has been thought from the first designs that it has **great flexibility to be deployed**, whether on dedicated servers, in containers, in the cloud, with Kubernetes or any desired deployment. It is ensured that the system can be adapted to different needs.

Because the development has been carried out by a single person, microservices have not been used, but rather a **monolith**, **prepared to be separated into microservices** if desired. Section 5 explains in more detail the decisions made regarding the server.

From this proposed design, the example of deployment carried out of the application for development and demos of the same can be derived (see Figure 13). The application has been installed on a local **Debian server**. An **Nginx** container, running on **Docker**, is being used as a gateway. This Nginx container is serving the static files, acting as a **reverse proxy** to the backend services in the `/api` route, and acting as a **load balancer** between the different **replicas of the service**. To connect from outside the local network, **Cloudflare tunnels** are being used, which allow a secure connection without exposing ports on the local network or its IP. All connections are secure using **HTTPS**, from the user's device to the backend services. Finally, the connection to the database is made with **R2DBC** as explained previously in server technologies. Note that the message queue defined in Figure 12 is not being used externally but is being used within each replica internally by Spring Events.

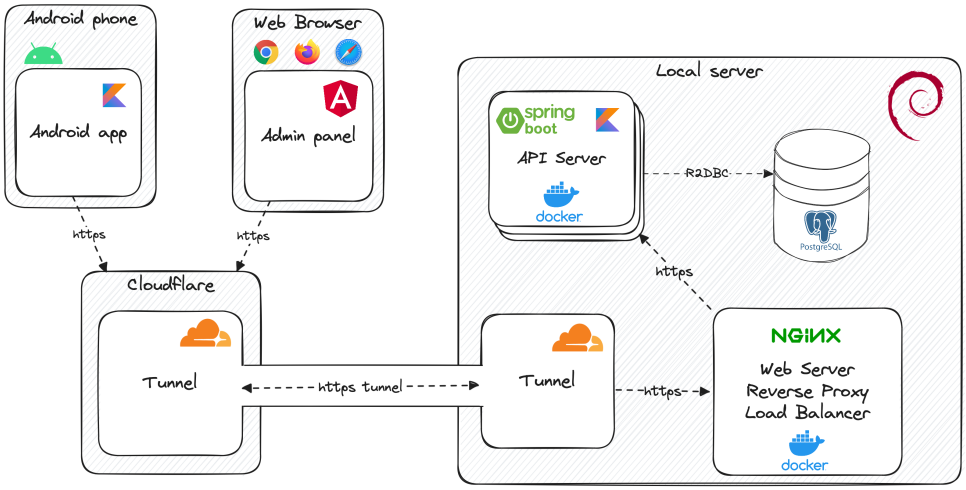

Figure 13: Deployment used during the development.

5

Server and Database

This section describes the server-side architecture and database design. It covers the back-end design decisions aimed at ensuring scalability and performance, as well as the security measures implemented.

5.1 Backend design

There are two main ways to develop server applications: monoliths and microservices. A **monolith** is the more **traditional way** of creating servers, it is a **single application**, where all its components are **tightly coupled** with many parts of the code. **Microservice**s, on the other hand, divide the application into small, **independently deployable services** that communicate via APIs or message queues. **Microservices have become very popular in the industry**, as they allow for greater flexibility, scalability and smoother, more agile development across multiple teams.

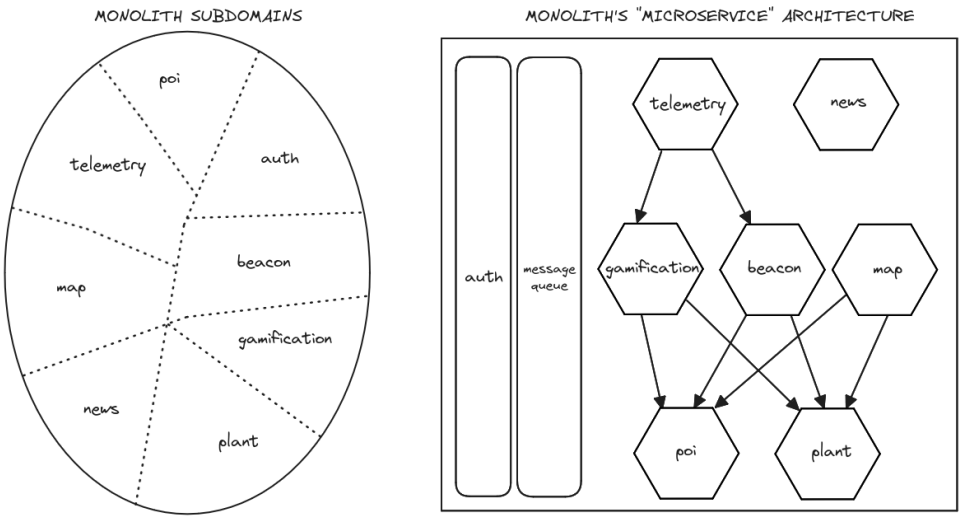

Figure 14: Division of subdomains inside the monolith.

The server application could be defined as a **modern monolith based on microservices principles** following the Microservices Patterns book by Chris Richardson [2]. The monolith has been structured into subdomains (see Figure 14), each one is designed taking into account **domain-driven-development** (DDD), **hexagonal architecture** and **clean architecture** as described in Section 2. With these principles, separation of responsibilities is achieved which facilitates maintenance and establish barriers between responsibilities.

The application itself is **stateless**, that is, it does not have any internal state, allowing it to be scaled to different replicas. It uses Restful APIs, JWT token authentication and is prepared to be used in Docker containers to **facilitate deployment and horizontal scaling**.

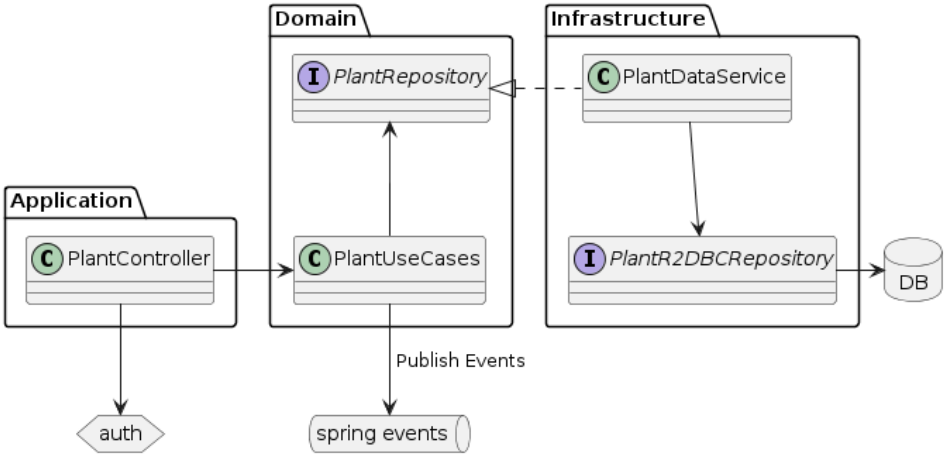

Figure 15: Class diagram of plant subdomain.

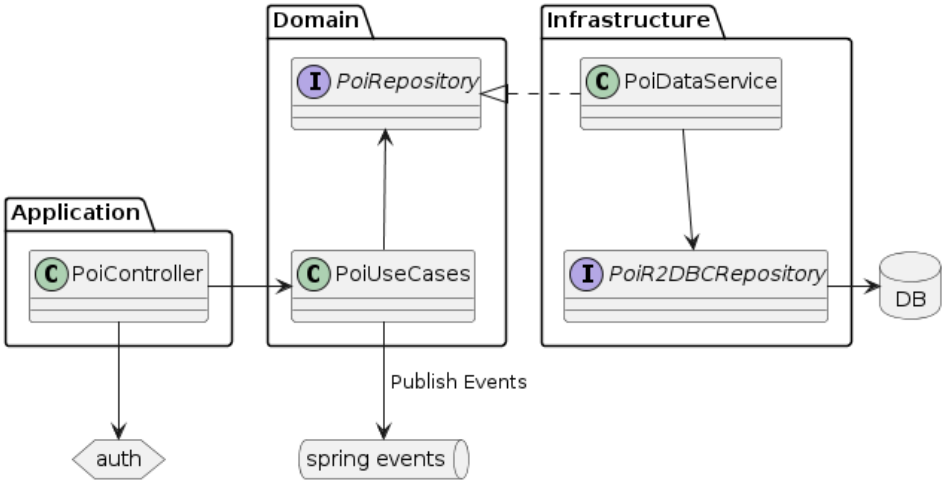

Figure 16: Class diagram of poi subdomain.

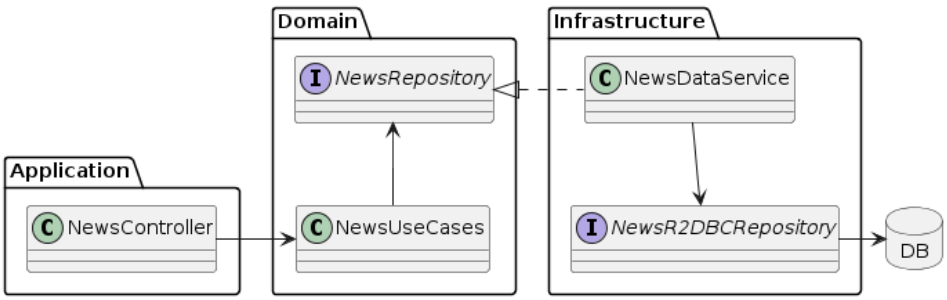

Figure 17: Class diagram of news subdomain.

Following the principles of **12-Factor App** [4], the server attempts to follow best practices to be a robust application, emphasizing codebase integrity, **dependency isolation**, environment-based configuration, and availability. The architecture allows to make several changes to the behavior of the server only by altering **configuration files**, such as pointing to different databases, activating/deactivating SSL or configuring different secrets.

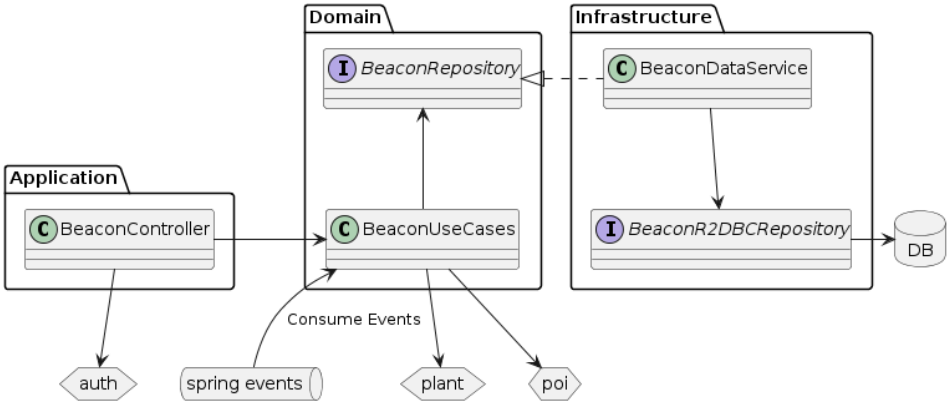

Figure 18: Class diagram of beacon subdomain.

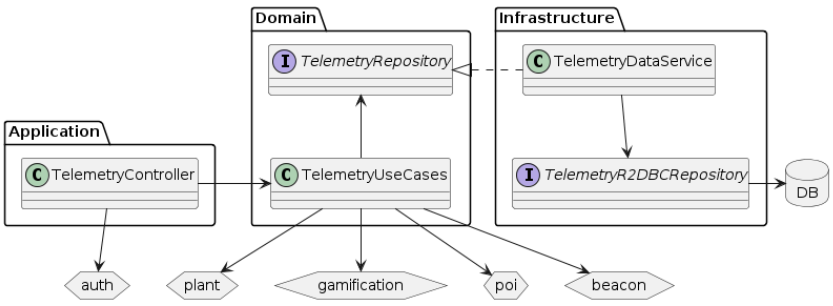

Figure 19: Class diagram of telemetry subdomain.

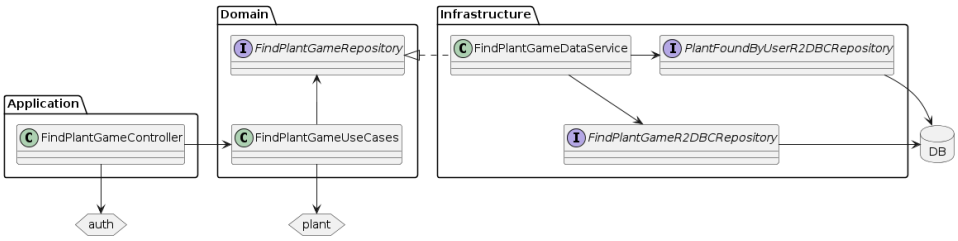

Figure 20: Class diagram of gamification subdomain.

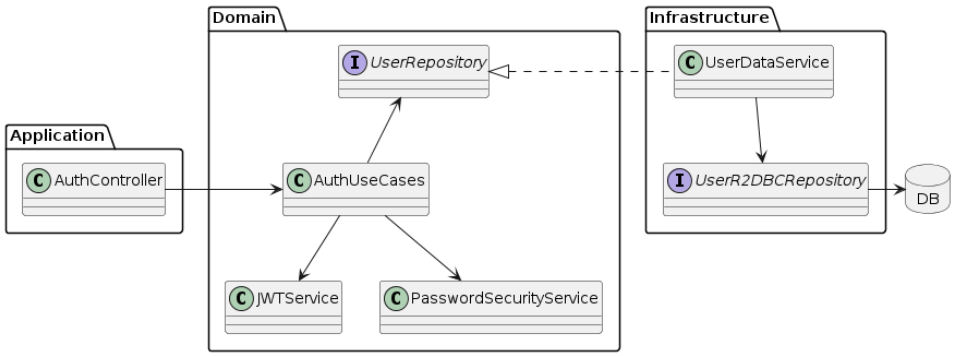

Figure 21: Class diagram of auth subdomain.

Most subdomains are straightforward and primarily involve **basic CRUD operations**, as illustrated in Figures 15, 16, 17, 18, 20, 19 and 21. In the diagrams three layers are observed, being **domain** the internal layer defined in DDD as explained in Section 2. The controllers define the **entry points** of the **application** and are the implementation of REST endpoints. They call the use cases within the **domain**. The use cases depend only on interfaces defined within the domain to interact with a database abstraction called repository. Repository interfaces are implemented by the **infrastructure** layer by services that use R2DBC repositories generated by **Spring Boot**. These are the ones that implement the communication with the database.

With these designs, applications are **decoupled from** both the **input** implementation (REST APIs through HTTP) and the **output** (R2DBC connection to a database). In addition, each diagram incorporates whether it communicates with another subdomain either by direct calls or by using an event queue.

The map subdomain, however, presents a **higher level of complexity** due to its inherently more intricate logic, as seen in Figure 22. It also follows the same layers (application, domain and infrastructure) but unlike the other subdomains it contains **four controllers**

which in turn call different classes of use cases. Most of the use cases like `RouteUseCases` or `PointUseCases` are very similar to the CRUDs that are defined in the rest of the subdomains. The `UserRouteUseCases` class makes use of auxiliary services for the calculation of optimal routes.

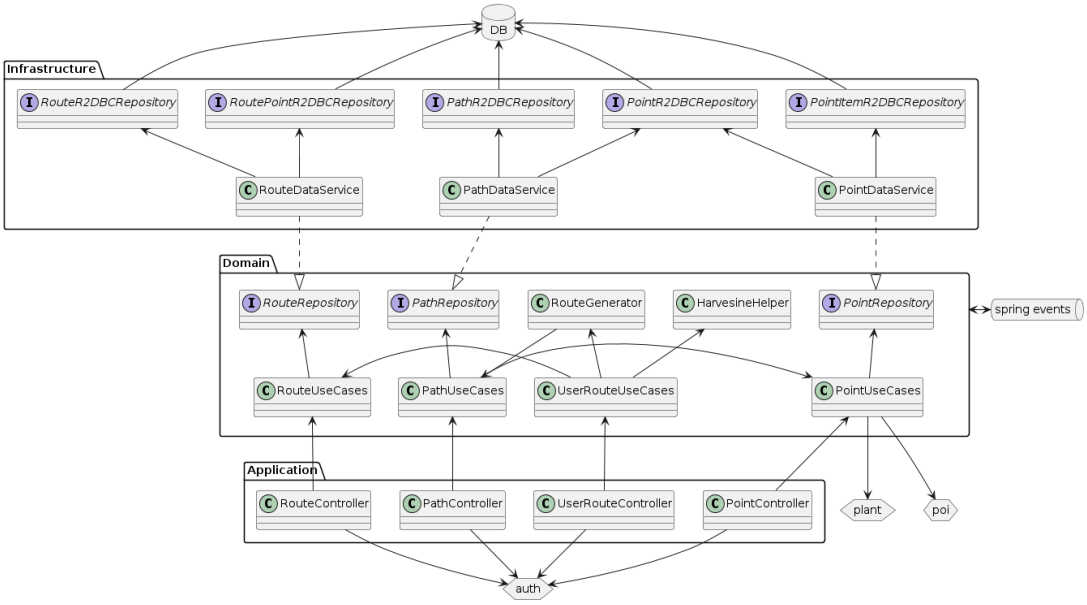

Figure 22: Class diagram of map subdomain.

As previously explained in Section 3, all subdomains are using a **non-blocking** concurrency model, that is, when a task does an I/O task, this task is **suspended** and the thread would be free to be used by another. This achieves **high performance** and **high concurrency** when performing tasks that involve doing networks calls as database queries.

5.2 Database

In **Domain-Driven Design** (DDD) [1] and the **12-Factor App** methodology [4], the database is considered an **external component** that is not part of the core. This perspective comes from the fact that the main focus of the application is the domain logic and business rules. Therefore, the database is treated as a simple external implementation detail, **secondary to the domain**. Thanks to this, **greater flexibility** is achieved to evolve the application since it is **not tightly coupled** to a specific technology or database.

In Figure 23, we can see the Entity-Relationship diagram of the database, the different tables have been grouped in the previously defined subdomains, in a microservice architec-

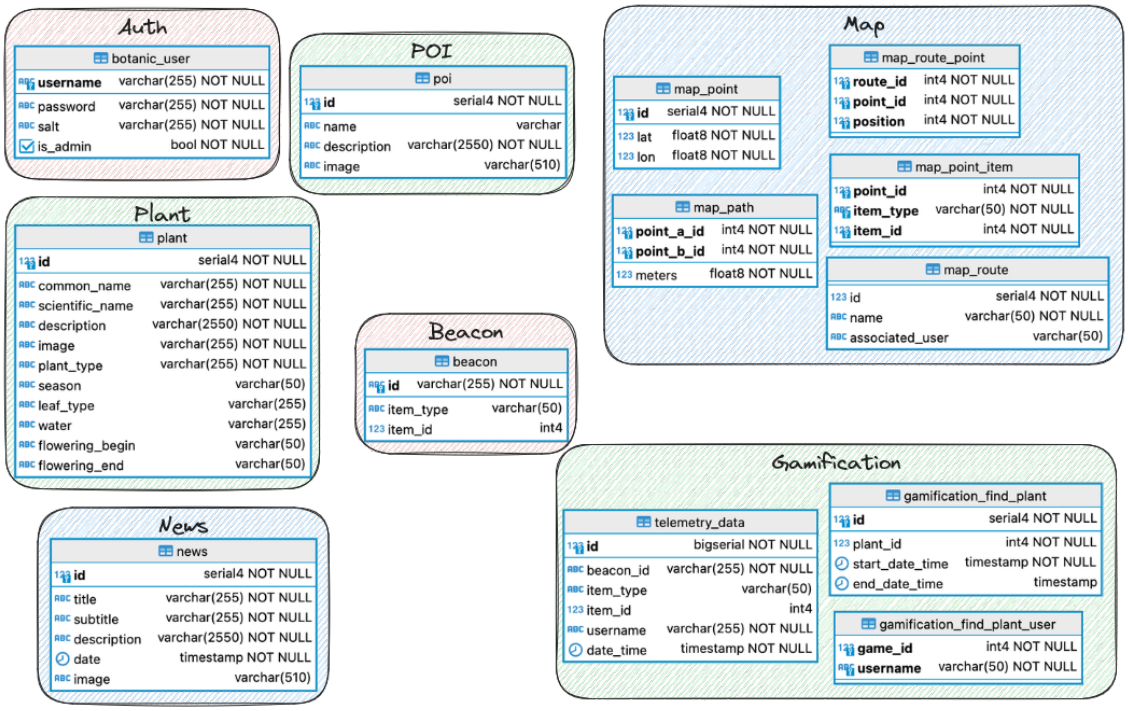

Figure 23: Entity relationship diagram of the database.

ture, **each microservice has its own database**, therefore the tables would be in different databases. This database structure is **automatically generated** by **Flyway** [16] when the application is initialized.

Additionally, in a microservices architecture [2], **each microservice would have its own database**. This concept helps that each microservice can be deployed independently of the rest. Due to this, foreign keys (FKs) cannot be used to maintain relationships between tables from different subdomains, with this we achieve a **loose coupling** and data consistency is maintained with inter-process mechanisms such as message queues or APIs.

5.3 Security

Following 12-Factor-App the server is configurable and there is **no key or secret written in the code**, but rather they are configurable through environment variables.

Firstly, the APIs go through **HTTPS**, so that network packets cannot be sniffed. Additionally, many endpoints require a **JWT token** to be used, some endpoints will require administrator permissions and others are only applicable to your own user (read from the JWT

token).

To avoid **SQL injection**, Spring repositories from Spring Data R2DBC [14] are used and the few handwritten queries use SQL parameters.

Passwords are not kept in the clear but are hashed with **SHA-512** following OWASP's **salt-and-pepper** recommendations [40].

5.4 Code patterns

The following are several code fragments that show the use of different design patterns.

- **Value object** [41]:

  In DDD, the concept of **value-object** (see Listing 1) is usually used to describe an aspect of the domain without having its own entity, for example, a field to be validated or normalized could be a value-object.

```
class BeaconId(id: String) {
    val value: String = id.lowercase()
}
data class Beacon(
    private val beaconId: BeaconId,
    val item: Item? = null
) {
val id: String
    get() = this.beaconId.value
}
```

Listing 1: Value object to normalize an id.

- **Command Query Responsibility Segregation (CQRS)** [42]:

  In clean architectures, when the writing data model is different from the reading data model, the CQRS pattern is usually used. This pattern defines a distinction between the writing domain (command) and the reading domain (query) (see Listing 2).

```
data class CreatePathCommand(
    val pointAId: Int,
```

```
    val pointBId: Int
)
data class Path(
    val pointA: Point,
    val pointB: Point,
    val meters: Double
)
```

Listing 2: Command to save a path and its query model.

- **Data Transfer Object (DTO)** [43]:

  The object defined in the domain is not always returned, or it is not related one to one with the database. For this, the DTO pattern is used to act as an intermediate object, for the transformations Kotlin extension functions are used to act as **mappers** (see Listing 3).

```
fun Poi.toDto(): PoiDto {
    return PoiDto(
        id = this.id,
        name = this.name,
        description = this.description,
        image = this.image
    )
}
```

Listing 3: Extension function to map a domain object to a dto.

- **Domain events**:

  This pattern consists of sending events when an action is performed to notify possible consumers, it can be written to a message queue as Kafka. In this case, **Spring events** (see Listing 4) are used as it is a **monolith**. With this we avoid cyclical dependencies.

```
// PlantUseCases.kt
@Transactional
suspend fun delete(id: Int) {
```

```
    repository.delete(id)
        .also { publisher.publishEvent(PlantDeletedEvent(
   this, id)) }
}
// BeaconUseCases.kt
@EventListener(PlantDeletedEvent::class)
fun handleEvent(event: PlantDeletedEvent) = runBlocking {
    findAll()
        .filter { it.item == Item(ItemType.PLANT, event.id
   ) }
        .collect { repository.update(it.copy(item = null))
    }
}
```

Listing 4: Domain event of a plant deleted.

5.5 Algorithm to calculate optimal routes

The first thing to be able to calculate optimal routes is to know the distance of the different paths in the botanical garden. To calculate the distance between two geopoints, previously created by the administrator, the Haversine formula [44] is used.

$$d = 2r \cdot \arcsin\left(\sqrt{\sin^2\left(\frac{\varphi_2 - \varphi_1}{2}\right) + \cos(\varphi_1) \cdot \cos(\varphi_2) \cdot \sin^2\left(\frac{\lambda_2 - \lambda_1}{2}\right)}\right) \tag{1}$$

where:

- d is the distance between the two points (along the surface of the sphere),
- r is the radius of the Earth,
- φ_1 and φ_2 are the latitudes of the two points in radians,
- λ_1 and λ_2 are the longitudes of the two points in radians.

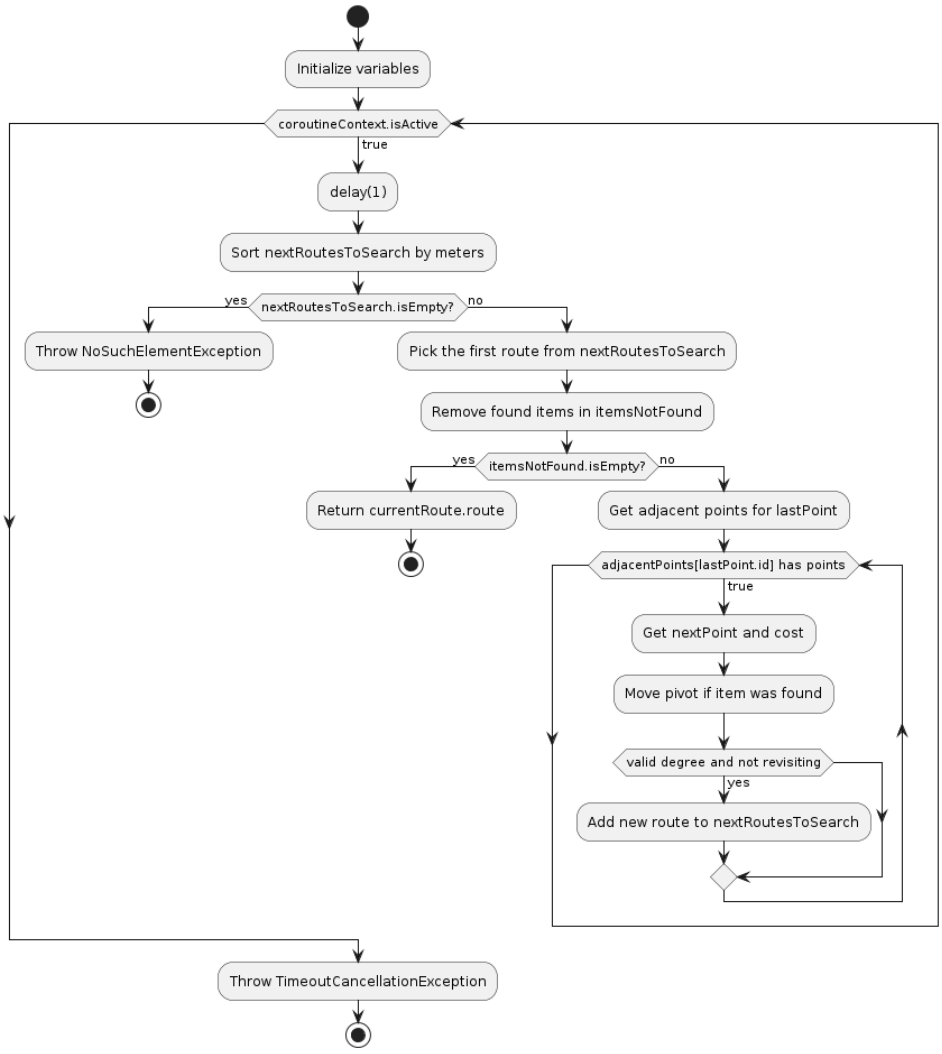

Figure 24: Activity diagram of the custom route algorithm.

Thanks to having the measurement of the botanical garden **paths**, we can apply different **graph algorithms** to calculate routes, such as A*, Dijkstra, Floyd-Warshall among others.

One of the **requirements** defined for this work was the generation of a route that would cover all the plants that the user wanted to visit, for this a **custom algorithm** has been designed to optimally cover all the points desired by the user. **At first** a **Breath First Search (BFS) with weights** that was capable of going backwards was designed. This, as expected, presented **several problems**, among which the following stand out:

- If **two points** are **very close** to each other, the algorithm spends a lot of time generating possible routes that bounce between those two points without exploring more paths.

- If there is a **cycle** in the graph, that cycle could be repeated several times.

These two problems can be avoided with these **two solutions**:

- Only explore unvisited nodes, until a searched plant is found, then be allowed to explore nodes previously visited to the found plant.
- The degree of a node cannot be exceeded.

These two ideas partially solve the problems, however, since it is a BFS algorithm, we **cannot ensure** that a path is **found in a reasonable time** without saturating the server. For this reason, a **timeout** has been added to cancel the Kotlin's coroutine if it exceeds a configured time.

In Figure 24, we can see the algorithm in question. While the Kotlin coroutine is not cancelled (it is **cancelled if** the algorithm **exceeds thirty seconds**), it will try to search for the most optimal route following a **flow similar to BFS**.

1. We **initialize** the algorithm by defining at which point it starts searching (the first point containing an item to be found), a relation of **adjacent points**, and an empty **list of tuples** of **routes** with **items to search** for and a **pivot**. Then all the following possible routes given the adjacent points are added to the list.
2. **The route with the smallest distance will be chosen**, the items found in the last node will be removed from its tuple. **If all the searched points have been found**, the algorithm **will finish returning this route**. Otherwise the algorithm will continue.
3. The cost of visiting **the adjacent points** which fulfill the conditions will be calculated. Afterwards, they will be **added to the list of tuples together** with the remaining items to be found and the pivot. The **pivot will move** only **if an item was found** in the last node, the pivot serves as a condition to **discard routes where it would go backwards**. Also as a condition no nodes are added when they have been visited as many times as the **degree of the node**.

5.6 Testing

Three types of tests have been used in the development:

1. **Unit tests:**

   For this, the **MocKK library** was used, where everything surrounding the class intended for testing was mocked, and tests were conducted solely on that piece without external effects.

```
@Test
fun delete(): Unit = runBlocking {
    every { publisher.publishEvent(any()) }.returns(Unit)
    coEvery { useCases.delete(ID) }.returns(Unit)
    val eventCaptor = slot<PlantDeletedEvent>()

    useCases.delete(ID)

    verify { publisher.publishEvent(capture(eventCaptor)) }
    assertThat(eventCaptor.captured.id).isEqualTo(ID)
}
```

Listing 5: Unit test of deleting a plant.

2. **Integration tests:**

   With the Spring's annotation `@DataR2dbcTest` and an **embedded H2** database, tests have been carried out to check the integration with the database of different subdomains of the application.

```
@BeforeEach
fun setUp(): Unit = runBlocking {
    beaconDataService.insert(BEACON)
}

@AfterEach
fun tearDown(): Unit = runBlocking {
    beaconDataService.findAll()
        .collect { beaconDataService.delete(it.id) }
```

```
}

@Test
fun findAll(): Unit = runBlocking {
    val list = beaconDataService.findAll().toList()

    assertThat(list).hasSize(1)
    assertThat(list[0].id).isEqualTo(BEACON.id)
    assertThat(list[0].item).isEqualTo(BEACON.item)
}
```

Listing 6: Integration test of deleting a plant with embedded database.

3. **E2E Tests:**

   During the development of the application, a **Postman collection** [45] was generated that covers all endpoints and allowed more realistic tests to be done in an environment with replicas of the service and a real PostgreSQL database.

On the Github server, there is a pipeline of **Github Actions** that **runs automatically** when uploading changes to the main branch, which runs both the unit tests and the integration tests.

6

Android Application

This section of the document presents the design and functionality of the Android application. It includes UI/UX design sketches, as well as screenshots of the application.

6.1 UI/UX design

At the beginning several designs were thought of how the application menu would be structured (see Figure 25). The objective was that the user experience (UX) would be common to other applications and would be intuitive for a **basic mobile phone user**.

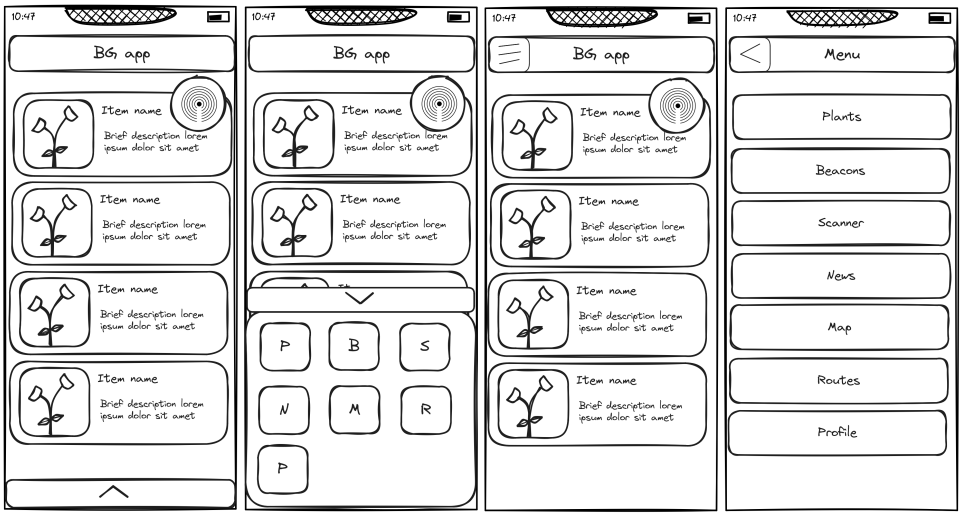

Figure 25: Initial designs for the android application.

Finally, a design was chosen that would have few options, and all of them would be accessible at the bottom of the screen **at the reach of the thumb** (see Figure 26).

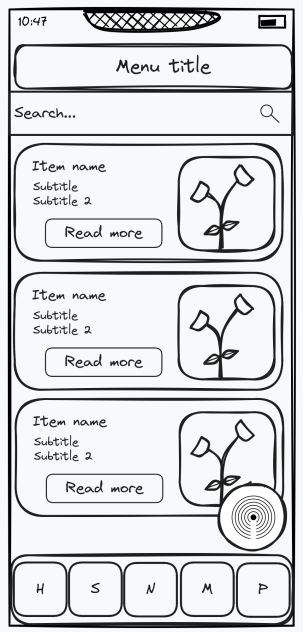

Figure 26: Final design for the android application.

6.2 Software design

As previously explained in Section 2, the android application follows the **MVC pattern**. Where we try to make a clear distinction between the data model, the controllers and the view. In Figure 27 it can be seen the organization of packages in the mobile application, the **pages** would be the "view" of the MVC pattern, representing the entire screen visible to the user in each functionality. While the **components** are the smaller parts that compose each page. These components can **be reused** by different pages. The user would interact with the pages, these would trigger actions in the **services**, which would be equivalent to the "controllers" of the MVC pattern. And the models are the representation of the different data of the application.

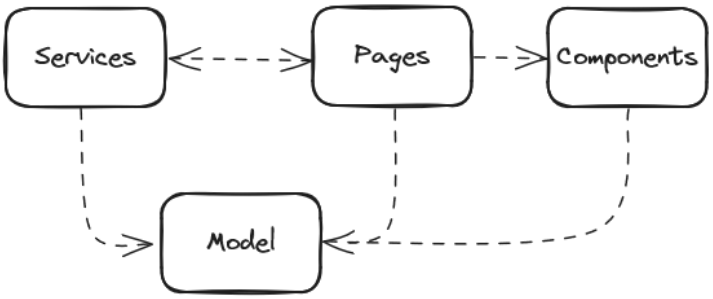

Figure 27: Packages of the android application.

Thanks to the components, the code is **reusable** and thanks to the implementation of the MVC pattern, a **separation of responsibilities** is achieved, making the code easier to mantain in the future.

6.3 Plants

The plant page (see Figure 28) has a **search input** field, which searches by both the common name and the scientific name of the plant. It also has a filter to show only **flowering** plants. Clicking on a plant will display a screen with more information about the selected plant.

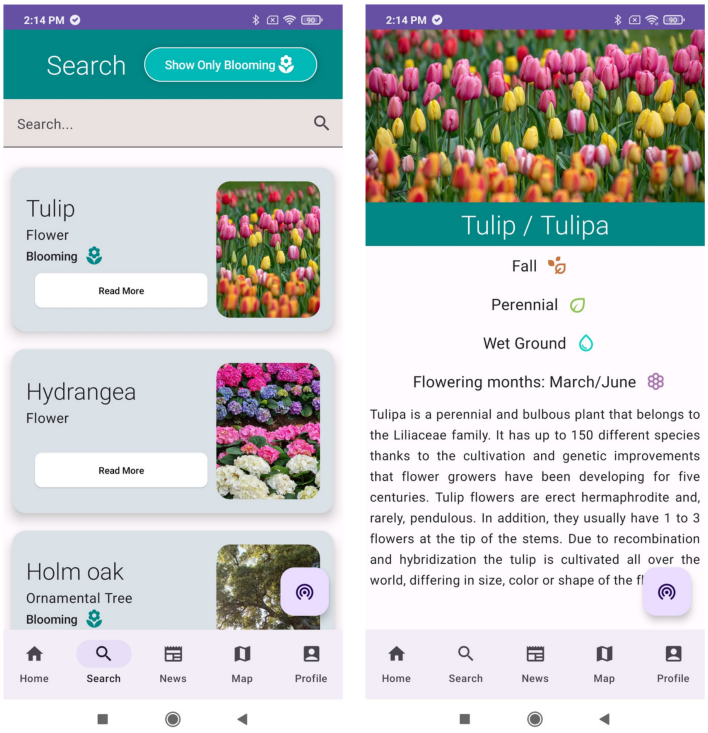

Figure 28: Page for plants in the android application.

6.4 Scanner

If the floating beacon icon is selected anywhere in the application, the scanner menu will open (see Figure 29). This menu has a toggle to activate or deactivate the scanner, in order to activate the scanner it is necessary that the user gives permissions for the use of **Bluetooth** and that it is active. Once enabled, several features of the application are activated:

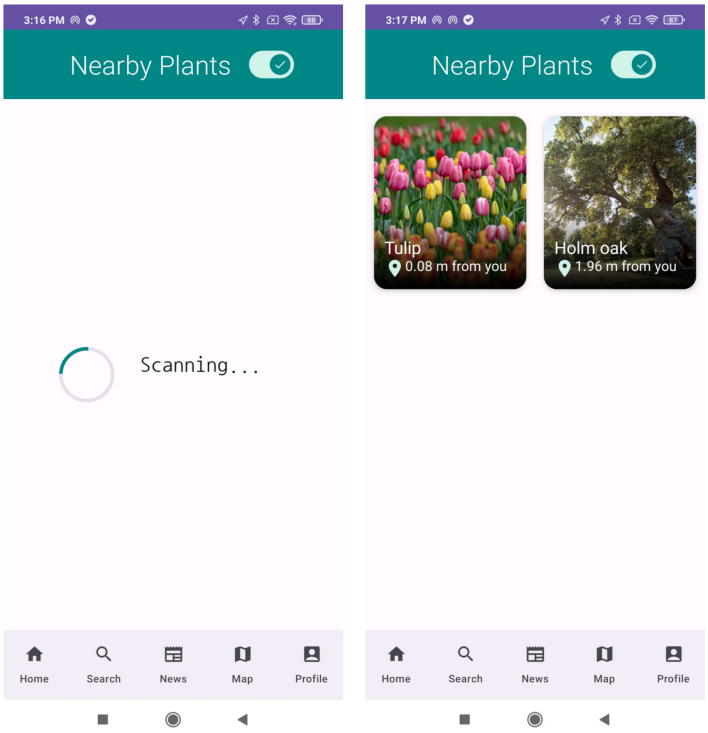

Figure 29: Plant scanner in the android application.

1. This screen will show the **nearby plants** and the **distance** to them.

2. In the routes menu, **routes detected** by the beacons will be notified.

3. **Telemetry** is activated.

4. **Gamification** activities are activated.

In addition, as it happened in the plants menu, when selecting a plant, information about it will be opened.

During the development, a **different behavior** of the **simulator** with respect to the **real beacons** was detected. A beacon simulator emits a constant signal while **a real beacon emits pulsations** of the signal (see Figure 30), this causes the beacons to seem to appear and disappear. To solve this problem in the `NearbyBeaconService` file a mechanism was defined to have a **time window** to keep remembering the beacons seen (see Listing 7).

```
fun getNearbyBeacons(): List<Beacon> {
    return beaconSeen.filter { it.wasSeenInTimeWindow() }
```

```
}

private fun Beacon.wasSeenInTimeWindow(): Boolean {
    return (System.currentTimeMillis() - this.
   lastCycleDetectionTimestamp) < TIME_WINDOW_MILLIS
}
```

Listing 7: Time window for beacons.

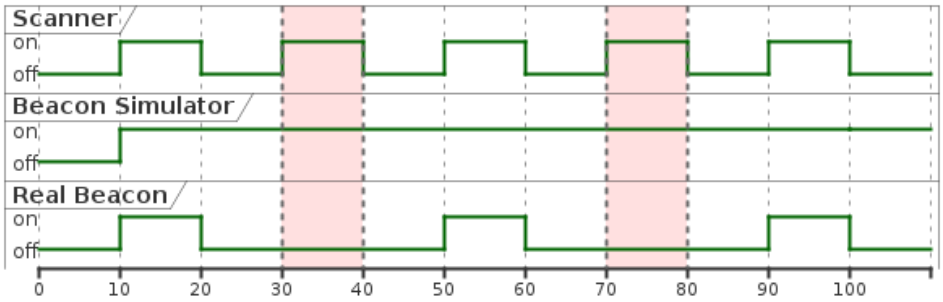

Figure 30: Real beacons do not emit continuous signals. Red areas are areas where a real beacon has not been detected but the simulator has.

6.5 News

The **news page** is a list of news items (see Figure 31). Selecting a news item will open a new page with more details of it, the date of the news item may be in the future, this can be used to notify of events taking place in the botanical garden.

6.6 Maps

The main screen of the map menu (see Figure 32), contains a button to generate custom routes, the previously **customized route** for that user if it exists, and a **list of routes** generated by the botanical garden administrators. An icon is included on **nearby routes** that contain a nearby plant or point of interest detected by a **beacon**.

If a route is selected from the list, it can be viewed on a map (see Figure 33), in addition to all paths in the botanical garden previously defined by an administrator.

Figure 31: Page for news in the android application.

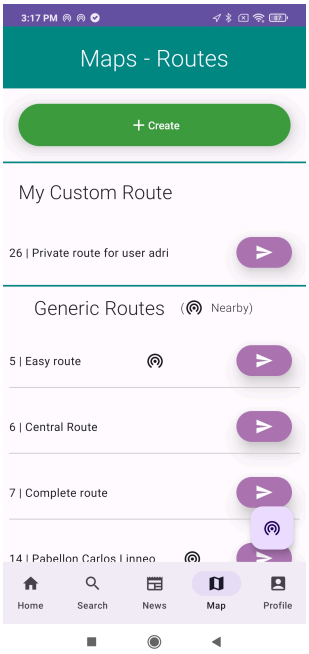

Figure 32: Page of map routes, with the recommended routes and the custom route of the user.

By accessing the **create route** option (see Figure 34), the user will be shown the plants and points of interest in the botanical garden to select what they want to see during their visit. If

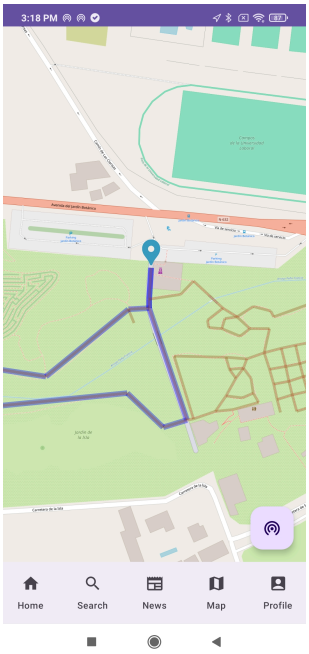

Figure 33: A route visualized in a map.

the route is successfully generated, this new route will replace the previously generated route for that user, each user could have a maximum of one personalized route. And as explained in Section 5, there can be no assurance that it will be possible to find a route that satisfies the requirements of the visitor in a reasonable amount of time.

6.7 Profile

In the user menu tab (see Figure 35) users can see which **gamification activity** is currently active. In addition, the amount of **points** the user has earned to date is displayed. Also included is a button to change the password and another button to log out of the application.

6.8 Performance

In order for the application to respond quickly and smoothly to the user and for the user not to feel that the application is lagging, **several tasks are performed asynchronously**. With the Kotlin **coroutines** (previously explained in the Section 3), the interface is not blocked while making network calls, if the user navigates between pages without having loaded the content, **the coroutine is canceled freeing resources**.

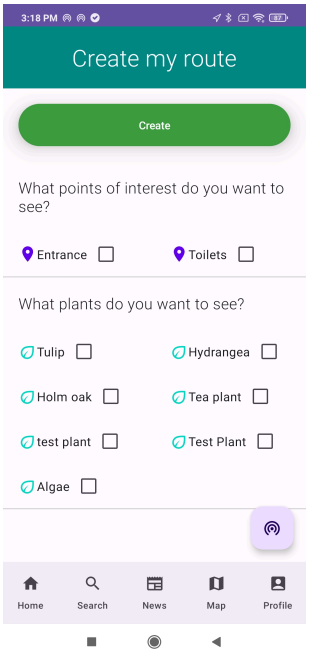

Figure 34: Form to generate a custom route.

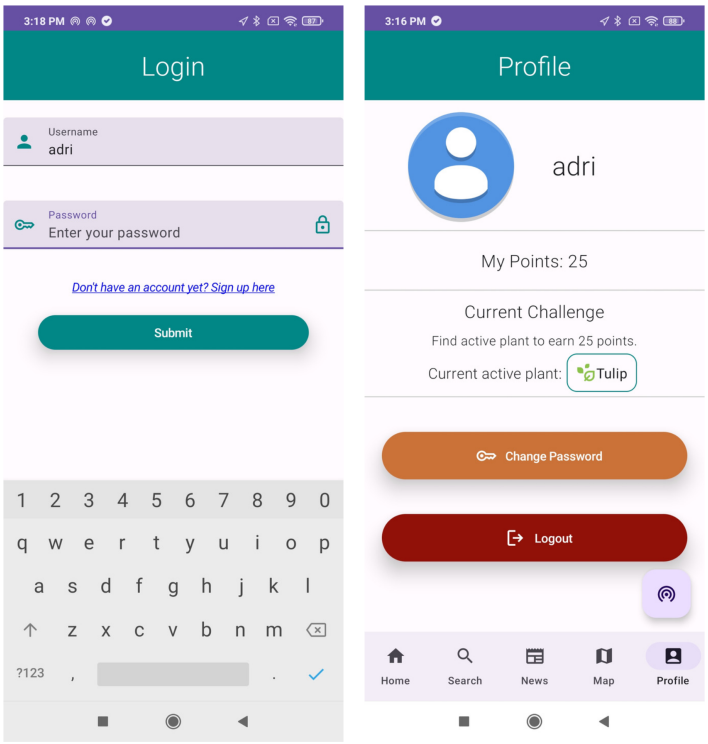

Figure 35: Login screen and profile of an user.

In addition, a **time cache** has been implemented for GET calls, for example to obtain the plants, during a certain time the call will be cached to avoid making unnecessary calls.

7

Administrator Web Panel

This section of the document presents the design and functionality of the web-based administrative panel. It includes UI/UX design sketches, as well as screenshots of the application.

7.1 UI/UX design

The UX/UI design of the web has as main objective to be **intuitive** and **easy to use**. The design has been focused on the use by **web browsers** in a resolution of 16:9, even though by using **Angular Material** components could be used in smaller devices such as mobile, however it is beyond the scope of this project that the web administration is responsive.

Figure 36: Design of the dashboard pages.

As seen in Figures 36, 37 and 38, it has been thought to have a **side menu** on the left with the different pages of the application. For consistency, pages that do similar things follow very **similar layouts** to decrease the **cognitive load**.

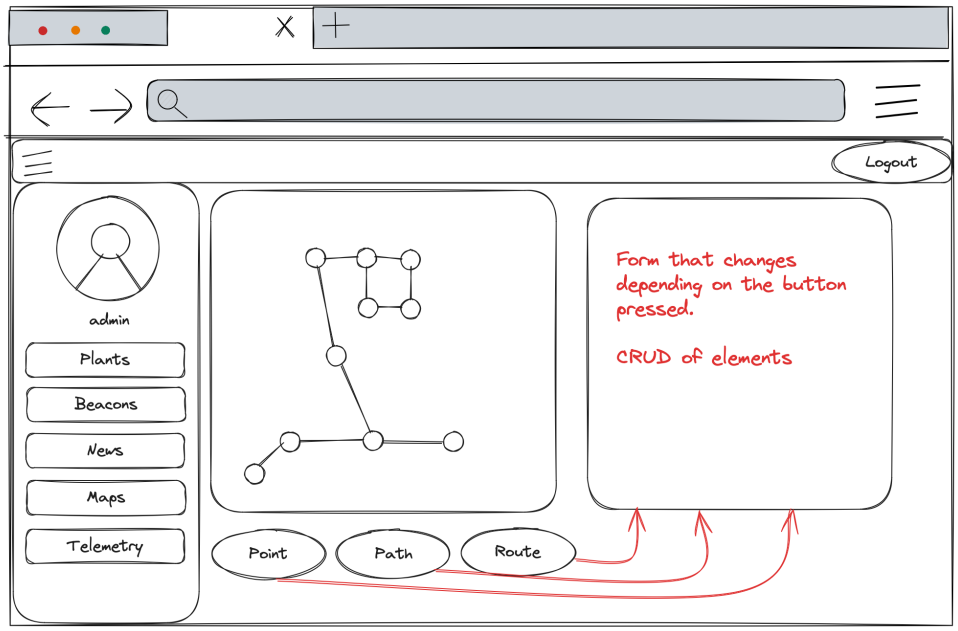

Figure 37: Design of the map page.

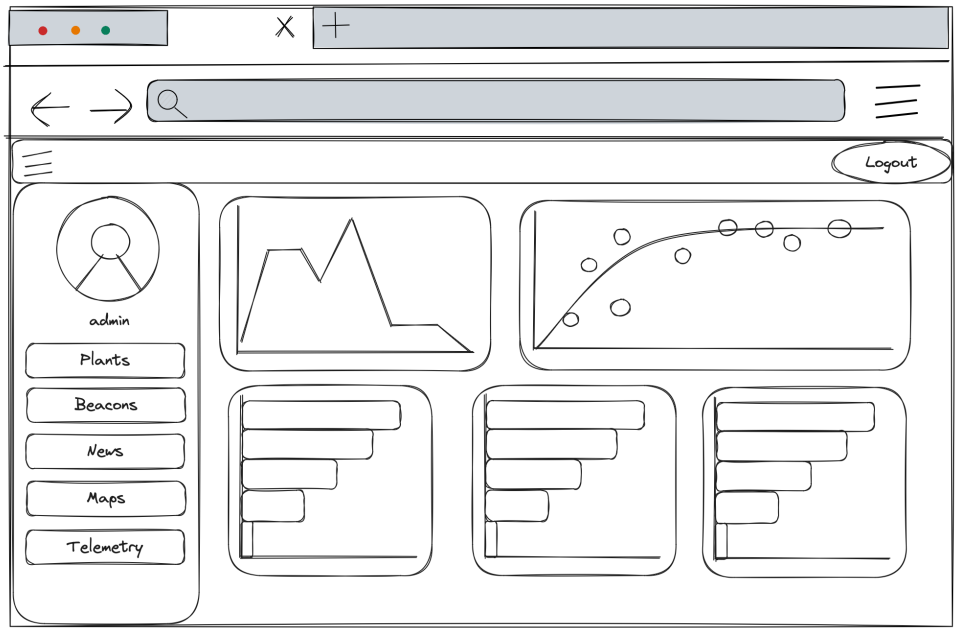

Figure 38: Design of the telemetry page.

7.2 Software design

The web application adheres to the **MVC pattern**, as previously discussed in Section 2. This pattern achieves a **separation of responsibilities** between the view (pages and controllers), the model and the controller (services). With a similar approach to the android application, **the same packages are defined** (see Figure 39). This helps to **reduce the cognitive load** between the mobile application and the web application **even though** they are **built on different technologies**.

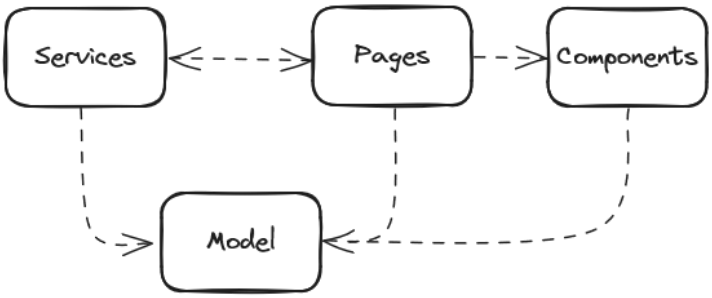

Figure 39: Packages of the web application.

This **similarity to the android design simplifies the development**, since when changing between the development of both applications many concepts are maintained and are transferable between them regardless of the framework or language used.

7.3 CRUD dashboards

Most of the application features can be summarized in a **CRUD** (Create, Read, Update, Delete) of elements. For this we have different **dashboards** to perform these operations:

- Dashboard for plants (see Figure 40).
- Dashboard for news (see Figure 41).
- Dashboard for beacons (see Figure 42).
- Dashboard for points of interest (see Figure 43).

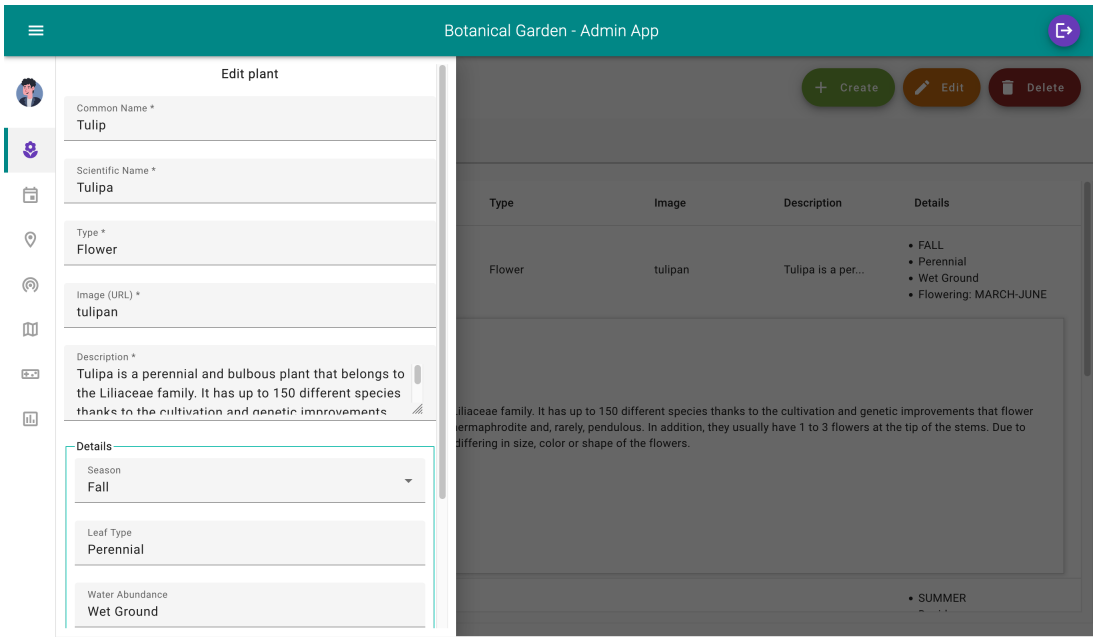

Figure 40: Dashboard page for plants while editing a plant.

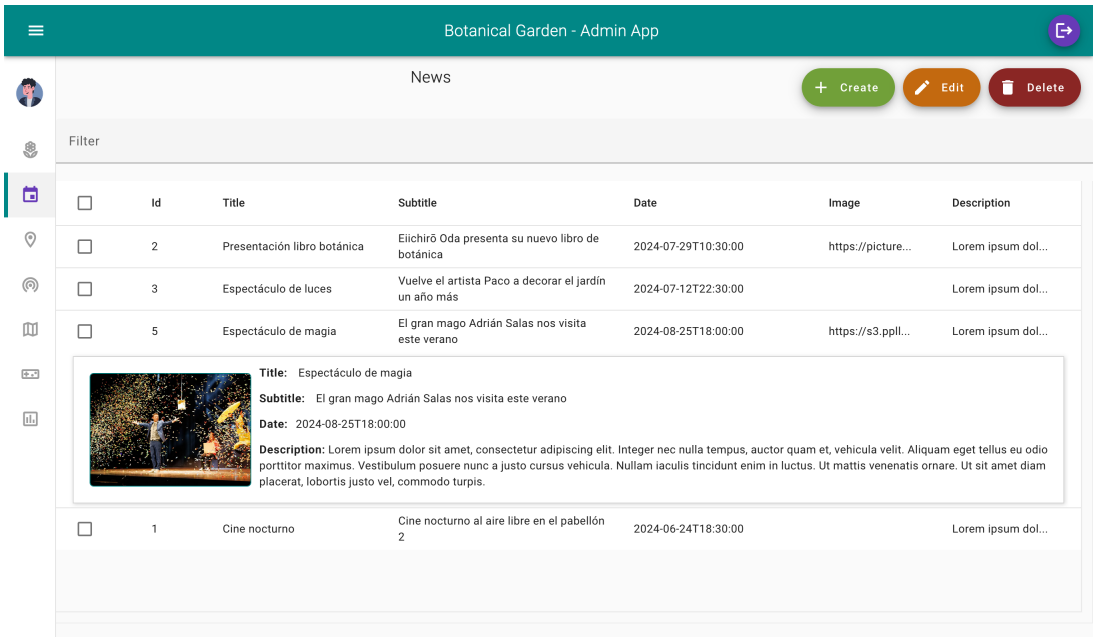

Figure 41: Dashboard page for news.

7.4 Gamification

In the gamification section (see Figure 44), we have added the option to **activate a plant** for users to earn points when they find it for the first time, you can also disable the activity, as well as check the points earned by all users of the application. To deactivate the plant finding

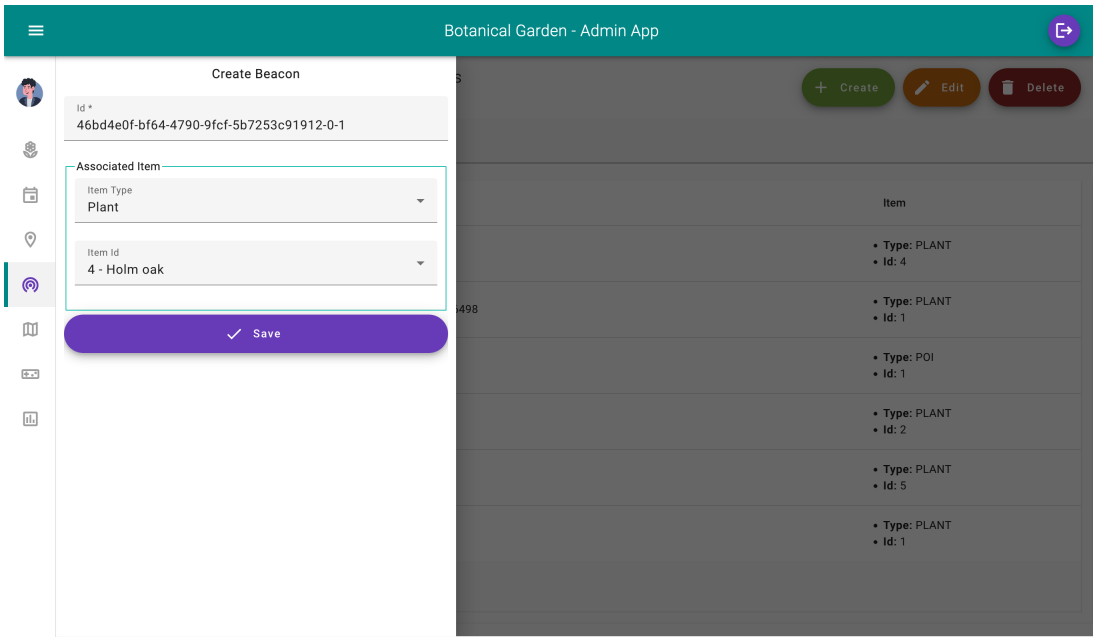

Figure 42: Dashboard page for beacons while creating a new beacon.

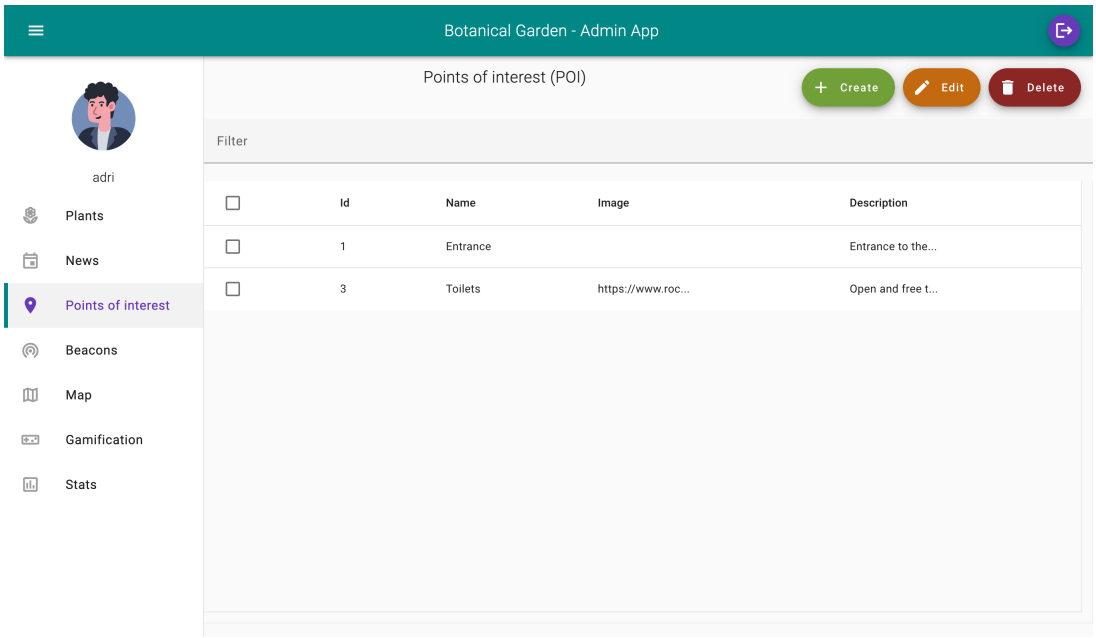

Figure 43: Dashboard page for points of interest.

activity, simply select that there is no active plant.

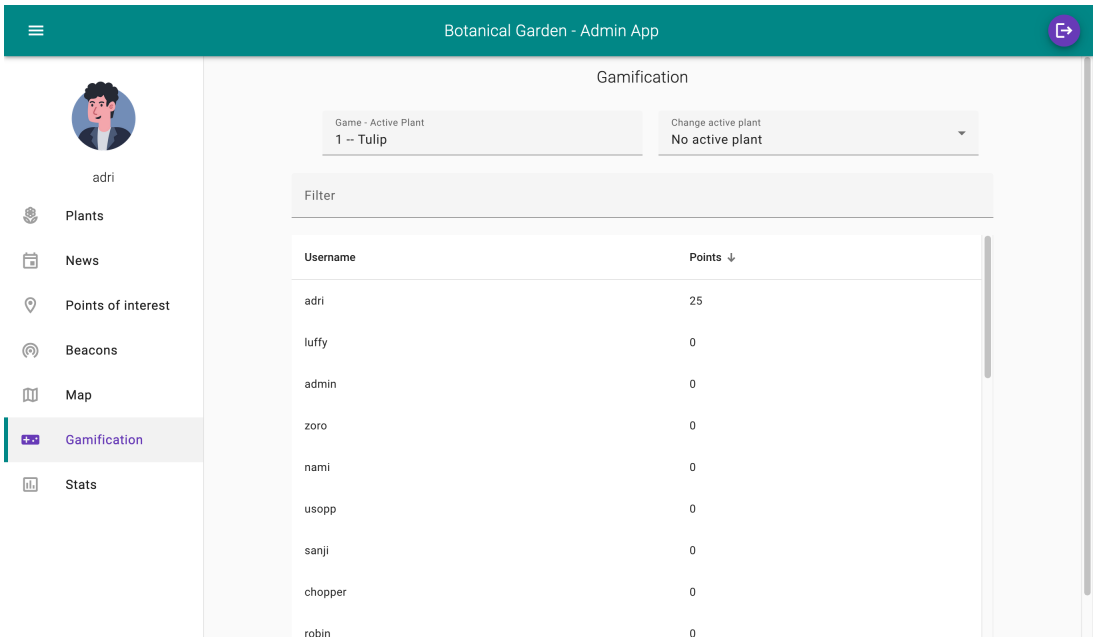

Figure 44: Gamification page displaying an activity and the leaderboard of points.

7.5 Telemetry

In this part of the application we can see the **value** that beacons can bring **to the botanical workers**, beyond improving the visitor experience. By including **metrics** of **data** collected thanks to the **beacons**, it is possible to extract several data about the **behavior of users**. Figures 45 and 46 show the different graphs of visitors per day, visitors per hour, most visited plants and most visited point of interest.

7.6 Maps

One of its features is the ability to add, edit and delete **points** (see Figure 47), each point can be assigned both a list of plants and a list of points of interest that exist at that point. To edit a point simply select it on the map, and if you want to change the longitude or latitude of the point you can click on an empty space on the map to autofill those two fields.

Another feature is the creation or deletion of **paths** (see Figure 48), to create a path you select the two desired points in the drop down menu and click on create, automatically you will see the new path on the map. To delete a path you can delete it by selecting it from the list of paths and confirming the action. To facilitate this procedure it is allowed to hover over

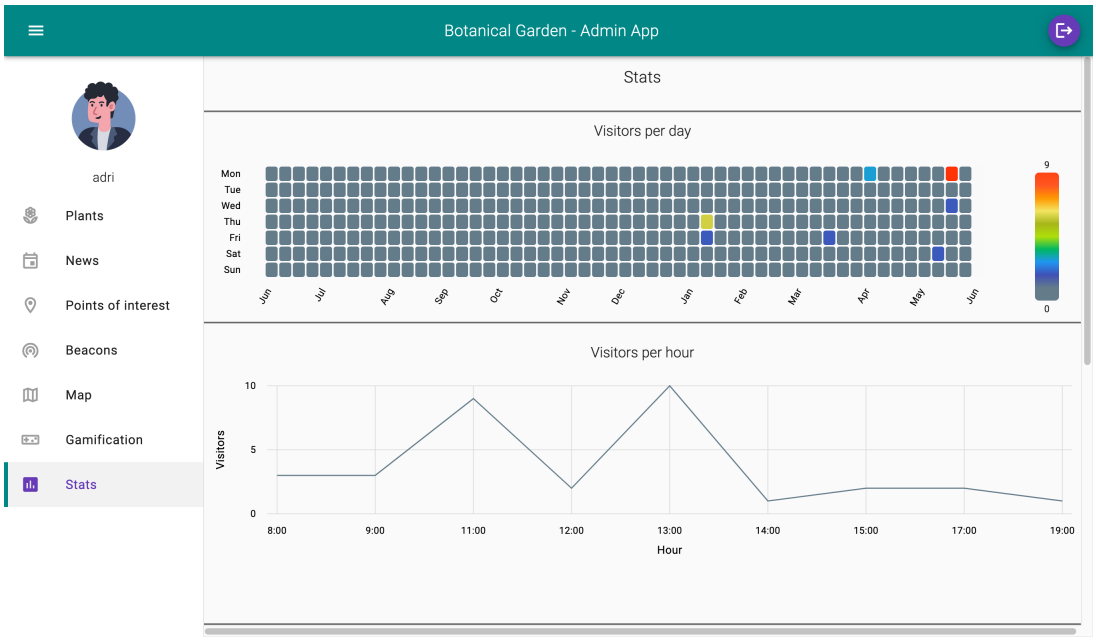

Figure 45: Telemetry page displaying statistics of visitors per day and hour.

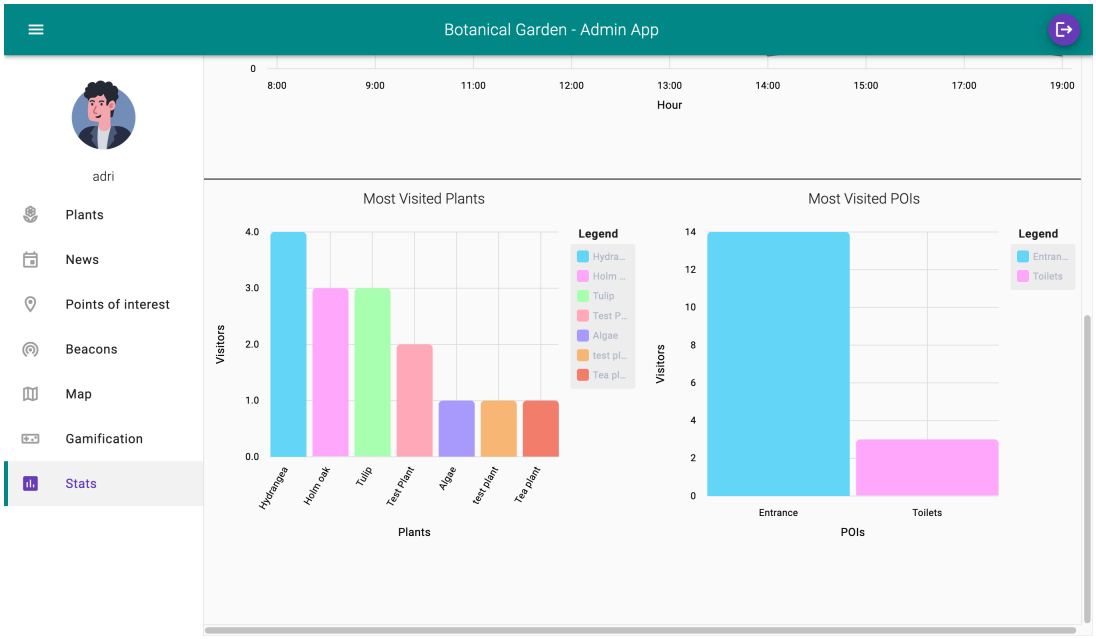

Figure 46: Telemetry page displaying statistics of most visited plants and points of interest.

the points on the map to see their id.

Another feature is to create **routes** (see Figure 49), and to visualize routes (see Figure 50), the routes are created in the order of the selected points and all the created routes would be visible by the users of the mobile application.

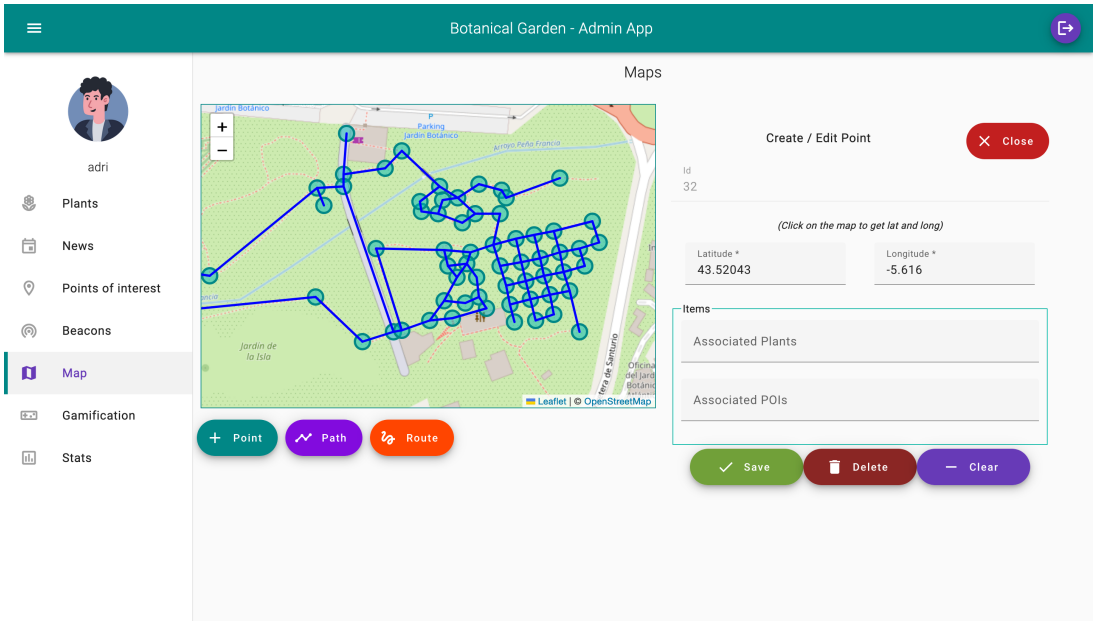

Figure 47: Map page with the option to create, edit or delete a point.

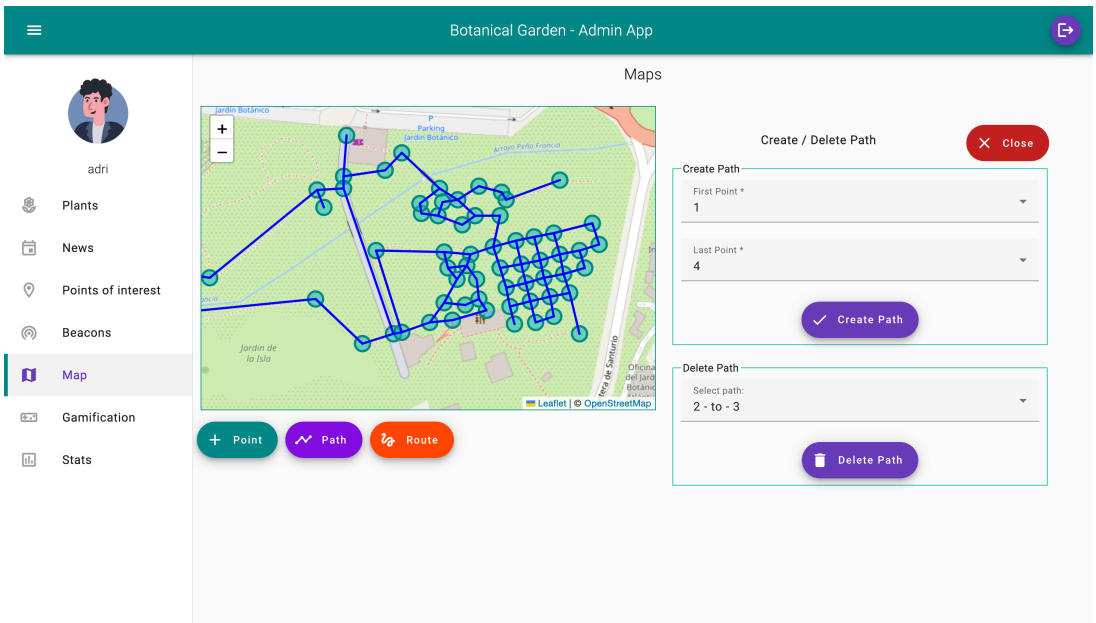

Figure 48: Map page with the option to create or delete a path.

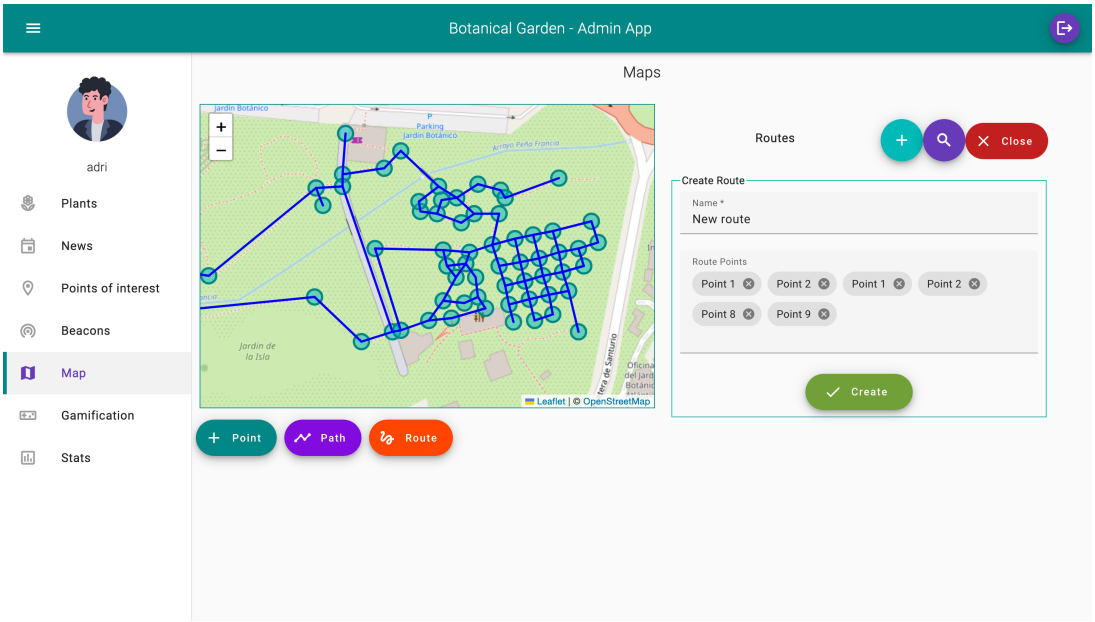

Figure 49: Map page with the option to create a route.

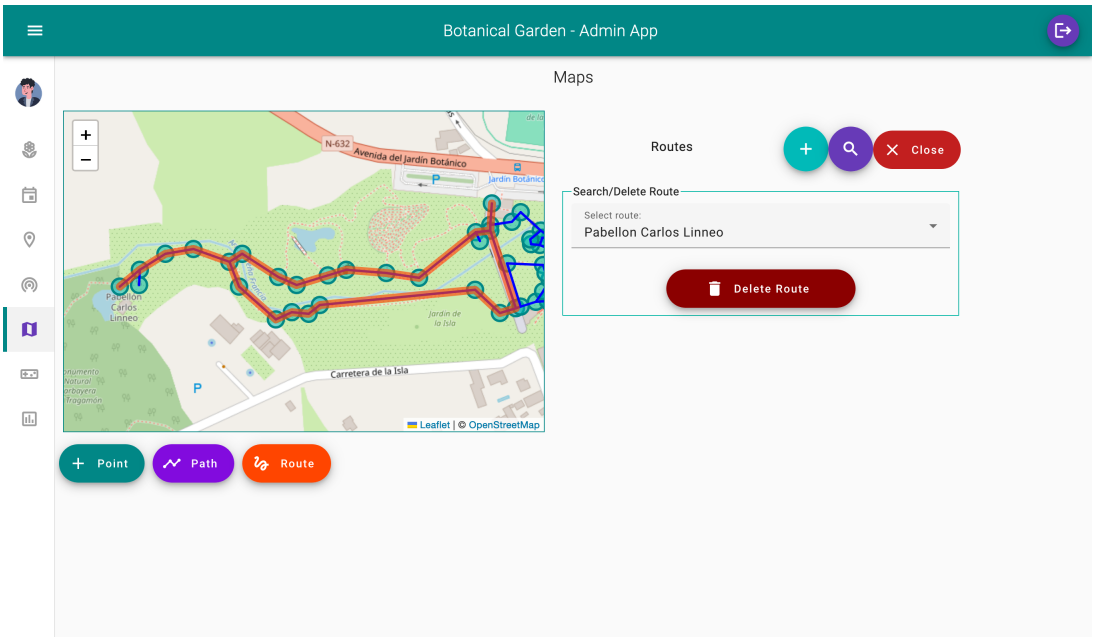

Figure 50: Map page with the option of displaying a route.

8

Conclusions and Futures Lines of Research

8.1 Conclusions

This thesis has shown **the design, development, and implementation** of a project for a botanical garden, comprising **a server**, **an Android application**, and **a web application** for administrators. This project comprises the delivery of three distinct applications while ensuring that they adhere to best practices, are secure, and are production ready.

The project responds to the need to **enhance** the botanical garden **visitor experience** by offering innovative ways to discover and explore the garden. **Using beacons**, the user experience can be personalized, either by showing nearby plants, recommending routes or through gamification activities.

In addition to enriching the visitor experience, the system provides **value to the botanical garden staff** by **collecting data from beacons** detected by mobile devices. **The resulting server** is **stateless**, robust and easily **scalable**, designed with software development industry **best practices** in mind to generate **clean architectures** and code that is **easy to maintain and test**. The web application provides administrators with a simple tool to manage the various features.

Each component (server, mobile application and web application) has been designed to be **easily extendable and scalable.** For example, the server has been developed with future microservices development in mind, the mobile app is prepared for a future iOS version and the web app is built using a modern component-based framework. This significant effort ensures that the entire system **is not only functional,** but also **optimized for future growth.**

In conclusion, this project delivers **production-ready software** that is poised to provide significant value to both visitors and botanical garden owners. The applications have been conceived and designed to evolve over time, addressing future needs that botanic gardens may encounter. This forward-looking approach ensures that **the system can evolve and improve over time** without resulting in high development costs.

8.2 Future lines of research

As has been mentioned numerous times throughout the document, the developed applications have not been conceived and designed to meet only the objectives set out at the beginning, but to be applications that can evolve over time and adapt to the future needs that botanical gardens may have.

The server could be expanded to **microservices**, with service discovery, with its own gateway and using a **time series database to collect telemetry** such as InfluxDB [46].

The mobile application with **Kotlin Multiplatform** [47] could be **migrated to iOS** as well as continue to explore ways to squeeze the use of beacons as it could be:

- **Position triangulation** [48] thanks to the beacon distance.
- **More gamification activities** like random quizzes when finding plants.
- **Collection of feedback** on certain zones when approaching beacons.
- **Crowd management**, to have real time control of where visitors are in the park and which areas could be congested.

The web application could also be expanded by adding more **statistics visualizations**, more gamification tasks, a dashboard with more options to manage users and a news editor with more options to edit the text and content of the news.

9

Conclusiones y Líneas Futuras

9.1 Conclusiones

Esta tesis ha mostrado **el diseño, desarrollo e implementación** de un proyecto para un jardín botánico, compuesto por **un servidor**, **una aplicación Android** y **una aplicación web** para administradores. Este proyecto comprende la entrega de tres aplicaciones distintas al tiempo que garantiza que se adhieren a las mejores prácticas, son seguras y están listas para la producción.

El proyecto responde a la necesidad de **mejorar** la **experiencia del visitante** del jardín botánico ofreciendo formas innovadoras de descubrir y explorar el jardín. El **uso de balizas** permite personalizar la experiencia del usuario, ya sea mostrando plantas cercanas, recomendando rutas o mediante actividades de ludificación.

Además de enriquecer la experiencia del visitante, el sistema **aporta valor al personal del jardín botánico** al **recoger datos de las balizas** detectadas por los dispositivos móviles. **El servidor resultante no tiene estado**, es robusto y fácilmente **escalable**, diseñado teniendo en cuenta las **mejores prácticas** del sector de desarrollo de software para generar **arquitecturas limpias** y un código **fácil de mantener y probar**. La aplicación web ofrece a los administradores una herramienta sencilla para gestionar las distintas funciones.

Cada componente (servidor, aplicación móvil y aplicación web) ha sido diseñado para ser **fácilmente ampliable y escalable.** Por ejemplo, el servidor se ha desarrollado teniendo en cuenta el futuro desarrollo de microservicios, la aplicación móvil está preparada para una futura versión de iOS y la aplicación web se ha construido utilizando un marco moderno basado en componentes. Este importante esfuerzo garantiza que todo el sistema **no solo sea fun-**

cional, sino que también **esté optimizado para su futuro crecimiento.**

En conclusión, este proyecto entrega **software listo para la producción** que está preparado para proporcionar un valor significativo tanto a los visitantes como a los propietarios de jardines botánicos. Las aplicaciones han sido concebidas y diseñadas para evolucionar con el tiempo, atendiendo a las futuras necesidades que puedan encontrar los jardines botánicos. Este enfoque con vista al futuro garantiza que **el sistema pueda evolucionar y mejorar con el tiempo** sin que ello suponga elevados costes de desarrollo.

9.2 Líneas futuras

Como se ha mencionado en numerosas ocasiones a lo largo del documento, las aplicaciones desarrolladas no han sido concebidas y diseñadas para cumplir únicamente con los objetivos planteados en un principio, sino para ser aplicaciones que puedan evolucionar en el tiempo y adaptarse a las necesidades futuras que puedan tener los jardines botánicos.

El servidor podría ampliarse a **microservicios**, con descubrimiento de servicios, con pasarela propia y utilizando una **base de datos de series temporales para recoger telemetría** como InfluxDB [46].

La aplicación móvil con **Kotlin Multiplataforma** [47] se podría **migrar a iOS** así como seguir explorando formas de exprimir el uso de beacons como podría ser:

- **Triangulación de posición** [48] gracias a la distancia de los beacons.
- **Más actividades de ludificación** como concursos aleatorios al encontrar plantas.
- **Recogida de feedback** sobre determinadas zonas al acercarse
- **Gestión de multitudes**, para tener un control en tiempo real de dónde se encuentran los visitantes en el parque y qué zonas podrían estar congestionadas.

La aplicación web también podría ampliarse añadiendo más **visualizaciones de estadísticas**, más tareas de ludificación, un panel de control con más opciones para gestionar a los usuarios y un editor de noticias con más opciones para editar el texto y el contenido de las noticias.

References

[1] R. C. Martin, *Clean architecture*. Prentice Hall, 2017.

[2] C. Richardson, *Microservices patterns: with examples in Java*. Simon and Schuster, 2018.

[3] *Model view controller*, Accessed: 2024-05-18, Martin Fowler. [Online]. Available: https://martinfowler.com/eaaCatalog/modelViewController.html.

[4] *The twelve-factor app*, Accessed: 2024-05-18, Adam Wiggins. [Online]. Available: https://12factor.net.

[5] J. K. Ousterhout, *A philosophy of software design*. Yaknyam Press Palo Alto, CA, USA, 2018, vol. 98.

[6] R. C. Martin, *Clean code: a handbook of agile software craftsmanship*. Pearson Education, 2009.

[7] S. Statler, A. Audenaert, J. Coombs, *et al.*, *Beacon technologies*. Springer, 2016.

[8] "Getting started with ibeacon version 1.0." Accessed: 2024-05-18, Apple. (2014), [Online]. Available: https://developer.apple.com/ibeacon/Getting-Started-with-iBeacon.pdf.

[9] *Eddystone*, Accessed: 2024-05-18, Google. [Online]. Available: https://github.com/google/eddystone.

[10] *Discontinuing support for android nearby notifications*, Accessed: 2024-05-18, Google. [Online]. Available: https://android-developers.googleblog.com/2018/10/discontinuing-support-for-android.html.

[11] *Altbeacon, the open and interoperable proximity beacon specification*, Accessed: 2024-05-18, AltBeacon. [Online]. Available: https://altbeacon.org.

[12] *How do apple airtag work?* Accessed: 2024-05-18, Msctek. [Online]. Available: hhttps://www.msctek.com/how-apple-airtags-work/.

[13] S. D. Padiya and V. S. Gulhane, "Iot and ble beacons: Demand, challenges, requirements, and research opportunities-planning-strategy," in *2020 IEEE 9th International Conference on Communication Systems and Network Technologies (CSNT)*, IEEE, 2020, pp. 125–129.

[14] *Spring boot*, Accessed: 2024-05-18, VMware Tanzu. [Online]. Available: https://spring.io/projects/spring-boot.

[15] *Postgresql*, Accessed: 2024-05-18, The PostgreSQL Global Development Group. [Online]. Available: https://www.postgresql.org.

[16] *Flyway*, Accessed: 2024-05-18, Redgate. [Online]. Available: https://flywaydb.org/.

[17] *Spring webflux*, Accessed: 2024-05-18, VMware Tanzu. [Online]. Available: https://docs.spring.io/spring-framework/reference/web/webflux.html.

[18] *Project reactor*, Accessed: 2024-05-18, Broadcom, Inc. [Online]. Available: https://projectreactor.io/.

[19] *Kotlin*, Accessed: 2024-05-18, Jetbrains. [Online]. Available: https://kotlinlang.org/docs/getting-started.html.

[20] *R2dbc*, Accessed: 2024-05-18, R2DBC. [Online]. Available: https://r2dbc.io/.

[21] *Android developers*, Accessed: 2024-05-18, Google. [Online]. Available: https://developer.android.com/.

[22] *Jetpack compose ui app development kit*, Accessed: 2024-05-18, Google. [Online]. Available: https://developer.android.com/develop/ui/compose.

[23] *Kotlin multiplatform*, Accessed: 2024-05-18, Jetbrains. [Online]. Available: https://kotlinlang.org/docs/multiplatform.html.

[24] *Altbeacon, beacon library*, Accessed: 2024-05-18, AltBeacon. [Online]. Available: https://altbeacon.github.io/android-beacon-library/.

[25] *Retrofit*, Accessed: 2024-05-18, Square, Inc. [Online]. Available: https://square.github.io/retrofit/.

[26] *Angular*, Accessed: 2024-05-18, Google. [Online]. Available: https://angular.io/.

[27] *Typescript*, Accessed: 2024-05-18, Microsoft. [Online]. Available: https://www.typescriptlang.org/.

[28] *Git*, Accessed: 2024-05-18, Software Freedom Conservancy, Inc. [Online]. Available: https://git-scm.com.

[29] *Github*, Accessed: 2024-05-18, Github, Inc. [Online]. Available: https://github.com.

[30] *Android studio*, Accessed: 2024-05-18, Google. [Online]. Available: https://developer.android.com/studio.

[31] *Visual studio code*, Accessed: 2024-05-18, Microsoft. [Online]. Available: https://code.visualstudio.com.

[32] *Intellij idea ce*, Accessed: 2024-05-18, Jetbrains. [Online]. Available: https://www.jetbrains.com/idea/.

[33] *Dbeaver*, Accessed: 2024-05-18, DBeaver Corporation. [Online]. Available: https://dbeaver.io.

[34] *Docker*, Accessed: 2024-05-18, Docker, Inc. [Online]. Available: https://docs.docker.com/get-docker/.

[35] *Nginx*, Accessed: 2024-05-18, Nginx, Inc. [Online]. Available: https://nginx.org/en/.

[36] *Cloudflare*, Accessed: 2024-05-18, Cloudflare, Inc. [Online]. Available: https://www.cloudflare.com/.

[37] *Plantuml*, Accessed: 2024-05-18, Arnaud Roques. [Online]. Available: https://plantuml.com.

[38] *Excalidraw*, Accessed: 2024-05-18, Excalidraw. [Online]. Available: https://excalidraw.com/.

[39] *Kanboard*, Accessed: 2024-05-18, Frédéric Guillot. [Online]. Available: https://kanboard.org.

[40] *Password storage - owasp*, Accessed: 2024-05-18, OWASP Foundation. [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html.

[41] *Value object*, Accessed: 2024-05-18, Martin Fowler. [Online]. Available: https://martinfowler.com/bliki/ValueObject.html.

[42] *Cqrs*, Accessed: 2024-05-18, Martin Fowler. [Online]. Available: https://martinfowler.com/bliki/CQRS.html.

[43] *Data transfer object*, Accessed: 2024-05-18, Martin Fowler. [Online]. Available: https://martinfowler.com/eaaCatalog/dataTransferObject.html.

[44] C. C. Robusto, "The cosine-haversine formula," *The American Mathematical Monthly*, vol. 64, no. 1, pp. 38–40, 1957.

[45] *Postman*, Accessed: 2024-05-18, Postman, Inc. [Online]. Available: https://www.postman.com/.

[46] *Influxdb*, Accessed: 2024-05-18, InfluxData. [Online]. Available: https://www.influxdata.com.

[47] *Kotlin multiplatform*, Accessed: 2024-05-18, Jetbrains. [Online]. Available: https://kotlinlang.org/docs/multiplatform.html.

[48] J. Kleinberg, A. Slivkins, and T. Wexler, "Triangulation and embedding using small sets of beacons," in *45th Annual IEEE Symposium on Foundations of Computer Science*, IEEE, 2004, pp. 444–453.

Appendix A
Installation Manual

The applications in this project are designed to be installed in a variety of different ecosystems, without being especially restricted to a single form of installation.

A simple container-based installation is provided in this appendix with two replicas for the API. Neither containers nor Nginx are necessary, it is only an example of installation and can be easily changed by other technologies.

1. Get a PEM certificate:
   - **Option A:** Get it from a CA.
   - **Option B:** Generate a self-signed a certificate.

     ```
     openssl req -newkey rsa:4096 -new -nodes -x509 -days
       36500
     -keyout "bgserver-key.pem" -out "bgserver-cert.pem"
     -addext "subjectAltName=DNS:${fill_this},IP:${fill_this
       }"
     ```

2. Generate the executables/statics files for each component:
   - Server.
     - Generate docker image

       ```
       ./gradlew bootBuildImage --imageName=salastroya/
         bgserver
       ```

     - Save image

       ```
       docker save salastroya/bgserver | gzip > bgserver.
         tar.gz
       ```

     - Load image

```
docker load < bgserver.tar.gz
```

- Android.
  - Set this environment variable
    - `API_ADDRESS` *(ex. api.example.org)*
  - *(Optional)* Add your public key

    If your server uses a self-signed certificate, add the public key in `./app/src/main/res/raw/cert.pem`.
  - Generate APK in Android Studio

    Build > Build Bundle(s)/APK(s) > Build APK(s)
- Admin web dashboard.
  - Install Angular CLI

    To install Angular CLI, execute the following command:

    ```
    npm install -g @angular/cli
    ```
  - Generate static files.

    To generate statics, execute the following command:

    ```
    ng build
    ```

    Your static files will be inside `./dist/bg-admin-app/browser/`.

3. Set these environment variables:
   - `POSTGRES_HOST` *(ex. localhost:5432)*
   - `POSTGRES_DATABASE` *(ex. test)*
   - `POSTGRES_USERNAME` *(ex. admin)*
   - `POSTGRES_PASSWORD` *(ex. password)*
   - `PEPPER_SECRET` *(ex. secretWordForPepper)*
   - `JWT_SECRET` *(ex. secretWordForJWT)*
   - `CERT_PATH` *(ex. /etc/ssl/bgserver-cert.pem)*
   - `PRIVATE_KEY_PATH` *(ex. /etc/ssl/bgserver-key.pem)*

4. Copy this nginx.conf

```
user              nginx;
worker_processes  auto;

error_log         /var/log/nginx/error.log notice;
pid               /var/run/nginx.pid;

events {
    worker_connections  1024;
}

http {
    include             /etc/nginx/mime.types;
    default_type        application/octet-stream;

    log_format  main  '$remote_addr - $remote_user [
  $time_local] "$request" '
                      '$status $body_bytes_sent "
  $http_referer" '
                      '"$http_user_agent" "
  $http_x_forwarded_for"';

    access_log          /var/log/nginx/access.log  main;
    server_tokens       off;
    sendfile            on;
    keepalive_timeout   65;
    gzip                on;

    ssl_session_cache   shared:SSL:10m;
    ssl_session_timeout 10m;
```

```
    server {
        listen 80 default_server;
        server_name _;
        return 301 https://$host$request_uri;
    }

    server {
        listen       443 ssl default_server;
        listen  [::]:443 ssl;
        server_name  _;

        ssl_certificate     /etc/ssl/private/bgserver-cert
   .pem;
        ssl_certificate_key /etc/ssl/private/bgserver-key.
   pem;

        location / {
            root   /usr/share/nginx/html;
            index  index.html index.htm;
        }

        location /api/ {
            proxy_pass https://bgserver:8080/api/;
        }

        error_page   500 502 503 504  /50x.html;
        location = /50x.html {
            root   /usr/share/nginx/html;
        }
}
}
```

5. Copy this docker-compose.yml

```
name: bgserver
services:
  nginx:
    image: nginx
    volumes:
      - ./nginx.conf:/etc/nginx/nginx.conf:ro
      - ./static/:/usr/share/nginx/html/:ro
      - ${CERT_PATH}:/etc/ssl/private/bgserver-cert.pem:ro
      - ${PRIVATE_KEY_PATH}:/etc/ssl/private/bgserver-key.
   pem:ro
    ports:
      - 80:80
      - 443:443
    networks:
      - default
  bgserver:
    deploy:
      replicas: 2
    environment:
      - POSTGRES_HOST=${POSTGRES_HOST}
      - POSTGRES_DATABASE=${POSTGRES_DATABASE}
      - POSTGRES_USERNAME=${POSTGRES_USERNAME}
      - POSTGRES_PASSWORD=${POSTGRES_DATABASE}
      - PEPPER_SECRET=${PEPPER_SECRET}
      - JWT_SECRET=${JWT_SECRET}
      - CERT_PATH=/bgserver/certs/bgserver-cert.pem
      - PRIVATE_KEY_PATH=/bgserver/certs/bgserver-key.pem
    volumes:
      - ${CERT_PATH}:/bgserver/certs/bgserver-cert.pem:ro
      - ${PRIVATE_KEY_PATH}:/bgserver/certs/bgserver-key.
```

```
    pem:ro
    image: salastroya/bgserver
    networks:
      - default
networks:
  default:
    name: bgserver_default
```

6. Run the containers:

```
docker compose up -d
```

If the steps were followed correctly, one nginx container and two API containers should be running on the server. Nginx will be in charge of both acting as a load-balancer and reverse proxy for the API, as well as serving the static files.

Appendix B
API Endpoints

This appendix lists all **API endpoints** with their **security measures**.

During the development of this project, a Postman collection was produced; a complete set of predefined requests, making it easy to **interact and test the API**. Keep in mind that for some routes you need the **JWT token** that you get when you log in with a user. You can obtain the **Postman collection** corresponding to this API at the following url: github.com/adrisalas/botanic-garden.

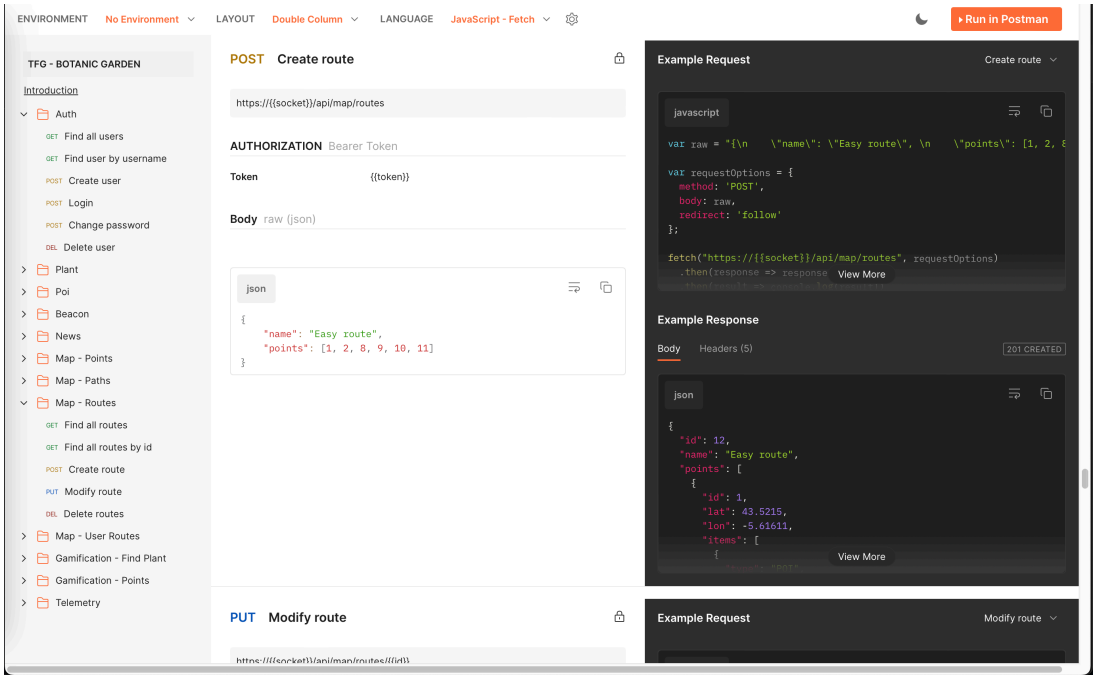

Figure 51: Postman Collection.

HTTP Method	URL	JWT Token	Only admin
POST	/api/auth/login		
GET	/api/auth/users	✓	✓
GET	/api/auth/users/{{username}}	✓	✓
POST	/api/auth/users		
POST	/api/auth/users/{{username}}/password	✓	
DELETE	/api/auth/users/{{username}}	✓	✓

Table 2: Auth endpoints.

HTTP Method	URL	JWT Token	Only admin
GET	/api/plants		
GET	/api/plants/{{id}}		
POST	/api/plants	✓	✓
PUT	/api/plants/{{id}}	✓	✓
DELETE	/api/plants/{{id}}	✓	✓

Table 3: Plants endpoints.

HTTP Method	URL	JWT Token	Only admin
GET	/api/poi		
GET	/api/poi/{{id}}		
POST	/api/poi	✓	✓
PUT	/api/poi/{{id}}	✓	✓
DELETE	/api/poi/{{id}}	✓	✓

Table 4: Points of Interest endpoints.

HTTP Method	URL	JWT Token	Only admin
GET	/api/beacons		
GET	/api/beacons/{{id}}		
POST	/api/beacons	✓	✓
PUT	/api/beacons	✓	✓
DELETE	/api/beacons/{{id}}	✓	✓

Table 5: Beacons endpoints.

HTTP Method	URL	JWT Token	Only admin
GET	/api/news		
GET	/api/news/{{id}}		
POST	/api/news	✓	✓
PUT	/api/news/{{id}}	✓	✓
DELETE	/api/news/{{id}}	✓	✓

Table 6: News endpoints.

HTTP Method	URL	JWT Token	Only admin
GET	/api/map/points		
GET	/api/map/points/{{id}}		
POST	/api/map/points	✓	✓
PUT	/api/map/points/{{id}}	✓	✓
DELETE	/api/map/points/{{id}}	✓	✓

Table 7: Map - Points endpoints.

HTTP Method	URL	JWT Token	Only admin
GET	/api/map/paths		
GET	/api/map/paths?pointId={{idA}}		
POST	/api/map/paths	✓	✓
DELETE	/api/map/paths/{{idA}}/{{idB}}	✓	✓

Table 8: Map - Paths endpoints.

HTTP Method	URL	JWT Token	Only admin
GET	/api/map/routes		
GET	/api/map/routes/{{id}}		
POST	/api/map/routes	✓	✓
PUT	/api/map/routes/{{id}}	✓	✓
DELETE	/api/map/routes/{{id}}	✓	✓

Table 9: Map - Routes endpoints.

HTTP Method	URL	JWT Token	Only admin
GET	/api/map/my-route	✓	
POST	/api/map/my-route	✓	

Table 10: Map - User routes endpoints.

HTTP Method	URL	JWT Token	Only admin
GET	/api/gamification/find-plant		
POST	/api/gamification/find-plant	✓	✓
DELETE	/api/gamification/find-plant	✓	✓

Table 11: Gamification - Find plant endpoints.

HTTP Method	URL	JWT Token	Only admin
GET	/api/gamification/my-points	✓	
GET	/api/gamification/all-users	✓	✓

Table 12: Gamification - Points endpoints.

HTTP Method	URL	JWT Token	Only admin
POST	/api/telemetry	✓	
GET	/api/telemetry/most-visited-plants	✓	✓
GET	/api/telemetry/most-visited-pois	✓	✓
GET	/api/telemetry/visitors-per-hour	✓	✓
GET	/api/telemetry/visitors-per-day	✓	✓

Table 13: Telemetry.

UNIVERSIDAD DE MÁLAGA | uma.es

E.T.S de Ingeniería Informática
Bulevar Louis Pasteur, 35
Campus de Teatinos
29071 Málaga

E.T.S. DE INGENIERÍA INFORMÁTICA