Digital Travellers

An application for recurrent travellers

Miquel de Domingo i Giralt

TBD

Contents

List of Figures

List of Tables

1. Introduction

1.1 Project introduction

Recurrent travel is a master's thesis for the Architecture and Design of Software of the La Salle URL university.
The project is being developed in the context of the increasing demand for air transportation and the digitalization of the travel industry. This project aims to offer an innovative solution for those users who frequently travel for work or personal reasons, through the creation of an application that allows them to be notified of opportunities on their regular routes.

This masters' thesis is framed in the ambit of project planning and management, software design and clean software architectures, databases, testing, user interface design and user experience.

1.2 Project structure

Even though the project was a team effort, this thesis will primarily focus on the frontend development of the project. I have actively worked on and been involved in the decisions regarding the frontend application development.

However, the thesis will also cover certain aspects of the backend, including key concepts, a brief explanation of its architecture, and how the frontend utilizes the API.

1.3 Context

Throughout the past decades, the European air transportation sector has undergone an unprecedented growth, primarily propelled by the democratization of pricing and the digitization of airlines. This expansion has resulted in a noticeable increase in the number of users who utilize both domestic and international flights.

In 2019, the volume of work flights increased to 1.28 billion dollars[1] and, even with the COVID-19 pandemic, the volume kept going up[2]. Business trips are essential for both interpersonal connection development and building trust between different companies[3]. Business trips are often conducted to visit the client, with over 44% of the business trips in Europe being made for client meetings and 32% for visiting the company offices in other cities[4]. Moreover, 30% of business travellers in Europe travel once a month, 62% once a year, and 5% between 21 and 40 times a year[4].

Group business trips are also a commonality, with over 50% of such being made by two or more workers[4]. When choosing a flight, 26% of the travellers in Europe will consider direct connections as the most important aspect of a flight, 19% will consider the price of the flight, 23% schedule coordination, and airport location 20%[4].

Taking into account the data before the COVID-19 pandemic, it is starting to be more common that employees are responsible for the booking of their own travels, with up to a 59% of them travelling in the United States, being also responsible to book the hotel[5]. Around the 69% of the travellers states that they book all the reservations, regardless of the type. Additionally, 79% of the business traveller have booked a trip through their mobile devices, indicating the increased use of mobile technology for this purpose[6].

In the post-pandemic work (2023), commercial aviation continues to be efficient, resilient and a key component to the development of the modern world. According to the annual report by

IATA[7], it is expected that by the end of 2023, all regions will have surpassed pre-pandemic flight demand. Additionally, the charts how an average positive variation of 5.4% points in real GDP in the coming years.

- For 2023, it is projected that the number of passengers will exceed pre-COVID-19 levels, reaching up to 105%.
- By 2030, it is estimated that the number of global passengers will grow to 5.6 billion.

Consequently, medium-term growth forecasts for commercial aviation indicate a promising and expansive future for the sector.

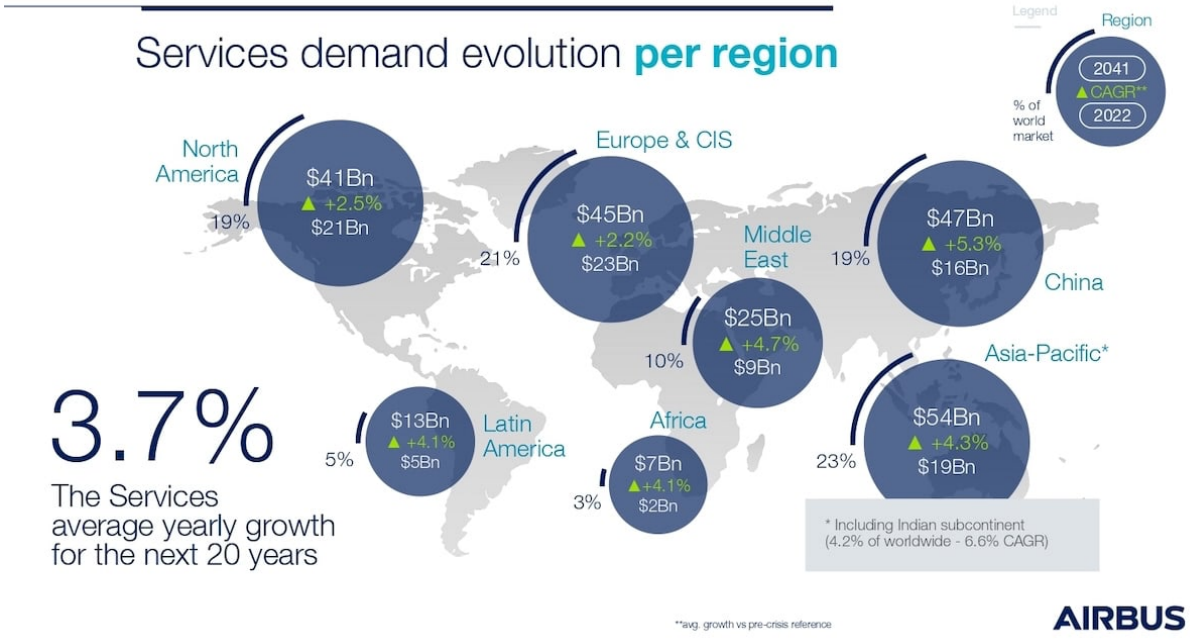

Figure 1.1: Evolution of flight demand of Airbus flights in the pre-pandemic world vs the post-pandemic.

Within this context of growth, it has appeared another type of user: recurrent travellers. These travellers, whether due to work, family or personal reasons, have incorporated the use of planes into their routine periodically throughout the year.

Recurrent trips are characterized by sharing the same point of origin and destination, similar departure and arrival times on all routes, and a stable repetition frequency throughout the year. Although airline digitalization and the ease of purchasing plane tickets online have greatly simplified the process, recurrent travellers still face difficulties when it comes to finding opportunities and optimizing their itineraries. The lack of specific solutions for this group of users has generated a growing demand for an application that notifies them of opportunities on their recurrent trips, facilitating the planning and management of their habitual travels.

Given the importance and frequency of business travel and the growing trend of employees taking care of their own reservations, it is crucial to address the challenges faced by these recurrent travellers.

1.4 Solution proposal

Throughout the project, we will focus on critical topics in software engineering. The hexagonal architecture will be addressed due to its ability to facilitate the development maintenance,

and modification of software by separating responsibilities into logical layers, thus promoting efficient code maintenance for the product.

Domain-Drive Design (DDD) is an architecture pattern that emphasizes in solving specific business problems, improving communication and the efficiency of the software model. Applying DDD will help us create software that is more understandable and effective by focusing on the different domain concepts of Recurrent Travel.
Furthermore, using DDD ensures the incorporation of oblique language, which utilizes domain-specific terminology that helps when planning a design in which stakeholders, business representatives and other team members are involved. Oblique language facilitates a clearer communication, a shared understanding, and the breakdown of complex problems, enable the creation of a more understandable and effective software that aligns with the domain concepts.

Additionally, the SOLID principles will be used for designing a modular, understandable, and extensible software. These principles are essential for establishing quality and maintainability standards in software component design. It also ensures the creation of a robust, reliable, and easily maintainable solution.

To ensure reliability and proper functioning of our application, testing and validation techniques will be implemented. Continuous Integration (CI) and Continuous Development (CD) will also be taken into account as part of the development cycle. Automation and version deployment will also take a spot in the cycle.

Furthermore, the integration with external systems will also be studied. This can improve the efficiency of our software solution by leveraging the functionality of other existing services in the market.

Finally, the importance of good practices in software development will be strongly emphasized, focusing on resource management, risk mitigation, effective communication, and delivering an MVP that meets customer expectations.
Ultimately, this master's thesis project aims to offer an effective and efficient solution to enhance the experience of recurrent travellers, enabling them to optimize their trips and facilitate the planning and management of their regular travels.

1.5 Topics and concepts

Before moving into the details of the project, it is fundamental to be familiarized with some key concepts regarding software architecture and design, which will be addressed throughout this project. In this regard, we will distinguish between general concepts, independent of the technology and environment in which they will be applied, and more specific concepts for the backend and frontend development.

1.5.1 General concepts

Before starting, it has to be taken into account the concepts about software architecture and design in the different applications to develop. As explained in the goals of the project, the software to be developed should be scalable, maintainable and robust.

In first place, it is essential to consider the meaning of software design. Software design refers to the creation of a logical and organized structure taking into account from the architecture, to modularity, reusability and readability of the code.

Therefore, as a team we opted to apply hexagonal architecture as the foundation of our ap-

plications. Hexagonal architecture is a software pattern whose goal is to separate between business logic of its technical concerns. In order words, it separates the domain of an application from the logic it may have. It also promotes a greater independence between the components and simplifies the adaptation to underlying technological changes. Additionally, it promotes modularity and code reusability. The clear separation of responsibilities of each layer and component, it facilitates the maintenance and evolution of applications over time. Another important aspect about hexagonal architecture is its focus on interfaces and contracts. The definition of such interfaces and contracts ensures a more efficient communication and facilitates the integration with other services or external systems.
Alongside hexagonal architecture, the team has opted to apply domain driven design in our projects. DDD is a methodology that it is focused in modelling the core business, focusing on specific concepts and rules of the domain. In this project, DDD allows to precisely define entities, value objects and aggregates from the current domain. Furthermore, the domain driven design allows the team to establish a common language between the members of the team, providing a more effective communication and a shared knowledge of functionalities and requirements.
It is also important to note the well-known SOLID principles, which have been used to develop a better software, at a lower level than the aforementioned concepts. Although not strictly an architectural concept, SOLID principles are equally important in terms of development, as they promote code modularity, extensibility and maintainability.

Regarding the deployment of the applications that will conform the system, the team has opted for cloud solutions. The usage of a cloud environment offers a set of significant advantages in terms of scalability, availability and flexibility, which are properties that are wanted for the project.
The cloud, also known as *cloud computing*, refers to the management of the infrastructure and computing services over the internet, instead of hosting the applications in local servers.
Furthermore, deploying the applications in the cloud frees the team from maintaining the underlying infrastructure. The cloud service provided will take care of the server management, networking, and other required resources, allowing the team to focus on the development.

Once the architecture and deployment aspects have been explained, it is crucial to incorporate an element that strengthens the robustness of the system: testing. In this project, two types of tests are distinguished: unit tests and functional or acceptance tests. It is worth mentioning that testing is not limited to these two types, many more types of testing exist, but such will not be covered in this specific project.
On the one hand, unit testing aims to validate the correct functioning of individual units or functional pieces of code. While the main objective is to ensure the expected functioning of such elements, their true value emerges in the future when modifications need to be made to old or legacy code. When making changes to existing code, developers may not take into account all the potential side effects that may arise from such changes. Thus, unit testing provides validation and security to the developer, ensuring that their changes do not break any existing functionality.
On the other hand, it is also essential to include functional tests alongside unit test. These tests aim to validate the overall behaviour of a system by attempting to recreate as close as possible the production environment. The focus relies on component interaction and complete workflows or user flows. Functional tests are particularly useful for detecting potential issues in the integration of different modules or components of the system. By simulating the real usage, it becomes easier to identify problems in communication between components, errors in business logic, or differences in user experience.
The combination of both type of tests, unit and functional, provides a comprehensive testing approach that covers both the internal validation of each component and the overall verification

of the system as a whole.

1.5.2 Backend concepts

The backend for the Recurrent travel application is the core of data processing in order to simplify the user experience in terms of finding travels. This application layer it is crucial for its functionality and system performance. However, it is not directly exposed to the client, meaning that it will require of a frontend.
When developing the frontend, it is really important to keep scalability and performance as the two main factors of the applications.

1.5.2.1 Gradle

Gradle is an advanced and flexible build automation system. Unlike other systems, it uses a Groovy-based language for its configuration file, which is more readable and easier to understand.

- Dependency Management: It allows for efficient and flexible dependency management, which is vital in a project of this scale.
- Automation: It is used to automate the build and deployment process, making updates easier and ensuring that all parts of the system are synchronized.
- Versatility: It supports both Java-based projects and other languages like Kotlin, making it extremely flexible for any language change or future additions. This ensures that this project has the freedom to adapt and evolve over time.
- Performance: It is designed to perform well in large-scale projects, thanks to features like caching and incremental execution. This means that only the parts of the project that have changed since the last build will be compiled, saving time and resources. This is especially valuable for this project, which can grow in complexity over time.
- Integration with IDEs and CI/CD: It integrates well with popular integrated development environments (IDEs) like IntelliJ IDEA and Eclipse. This means that developers can use and manage Gradle directly from their IDE. It also integrates with continuous integration/delivery (CI/CD) tools like Jenkins and Travis CI, facilitating automatic deployment and updates.

Therefore, the choice of Gradle for Recurrent travel not only facilitates dependency management and project automation but also provides a solid and flexible foundation for the project's evolution and growth in the future.

1.5.2.2 Java

Java is a widely used programming language known for its portability, efficiency, and support for object-oriented programming.

- Portability: Java is platform-independent, allowing the backend code to run on any operating system that has the Java Runtime Environment (JRE) installed, facilitating deployment and scalability.
- Maturity and Community Support: It has been a staple in the software industry for over two decades, which means it has a wealth of libraries, frameworks, and community support.

- Reliability and Stability: Java has a long track record of reliability and stability, making it a safe choice for critical and large-scale applications. Its strong exception handling system and automatic garbage collector help minimize errors and prevent memory leaks, improving overall application robustness.

- Security: Java was designed with security in mind. Its security features include a strong type system, bytecode verification before execution, and absence of pointers, helping prevent common errors that can lead to security vulnerabilities.

- Support for Enterprise Development: It provides a rich ecosystem of frameworks and libraries for enterprise development, such as Spring, Hibernate, and Apache Camel, among others. These tools simplify the implementation of a variety of backend functionalities, from database access and security to integration with other systems and services.

Therefore, choosing Java as the programming language for the backend of Recurrent travel provides a solid and robust foundation for building a secure, scalable, and high-performance application.

1.5.2.3 Spring boot

Spring Boot is a framework that simplifies the configuration and deployment of Spring-based applications.

- Rapid Development: It provides default configuration that accelerates development by eliminating the need for extensive manual configuration.

- Integration: It integrates with many tools and libraries, such as Hibernate for data persistence and Spring Security for security, making it easy to implement these features.

- Integration with DevOps: It integrates well with DevOps practices and tools. For example, it integrates with Jenkins for continuous integration and with Docker for containerization, facilitating development, testing, and deployment of applications.

In summary, choosing Spring Boot for the development of this project enables quick startup, scalability, and easy maintenance. Its wide range of features and integrations facilitate the creation of a high-quality application that meets the demands and expectations of frequent travellers.

1.5.2.4 Unit test

The implementation of testing is a crucial component to ensure software quality and user satisfaction in this project. In the context of unit testing, technologies like JUnit and Mockito are used.

JUnit is the most widely used unit testing framework for Java that helps developers design and execute tests to verify the behaviour of small units of code, such as individual methods or classes. In this project, the usage of JUnit provides the following benefits:

- Quality Assurance of Code: By using JUnit for unit testing, it can be ensured that each unit of code functions correctly before integration. This helps detect and correct errors early in the development cycle.

- Facilitates Updates and Maintenance: Unit tests with JUnit facilitate updates and maintenance of the application. Whenever a change is made to the code, unit tests can be executed to ensure that the changes haven't introduced new errors.

Mockito is a popular Java mocking framework used to mimic objects and behaviours of the test class. For the project, Mockito can be used to simulate application logic that is not directly related to the code unit being tested, such as database dependencies. The benefits of Mockito in this context include:

- Isolation of Code Units for Testing: It allows creating simulated objects (mocks) of external dependencies, meaning that unit tests can focus on the code under test without worrying about the behaviour of dependencies. This makes the tests more reliable and easier to write.
- Simulation of Diverse Test Scenarios: It is possible to simulate a variety of test scenarios, such as the behaviour of the application when an external service fails or returns unexpected data. This helps ensure that Recurrent travel can properly handle these situations in production.

In summary, the combination of JUnit and Mockito enables comprehensive and effective unit testing in "Recurrent Travel," resulting in more reliable and maintainable software.

1.5.3 Frontend concepts

In terms of frontend development, it is important to consider the same concepts mentioned in the first section. Although more details will be covered in future sections, unlike server repositories, the client repository has been developed as a mono-repository. There is a difference between a mono-repository and a multi-repository, such as:

- *Multirepo*. The multirepo approach means to have multiple applications in different repositories. The main benefits of such approach are the fact that teams can separately work in the repository while at the same time the repository is kept smaller and cleaner.
- *Monorepo*. The monorepo approach is the opposite of the multirepo: all the applications are developed from the same repository. Such approach allows maintaining build and deployment patterns altogether. However, application versioning may be harder to manage.

When aiming to develop a client-scalable application while maintaining a single shared domain layer, the mono-repository architecture has been preferred.

1.5.3.1 Nx

One of the main challenges that a team can face when working with a mono-repository is its maintenance. Maintenance includes tasks such as library updates and managing CI/CD pipelines, if they exist.
Fortunately, there are tools that simplify this maintenance, known as build tools. In the JavaScript world, a tool called Nx has gained significant popularity. Translated from its documentation: *Nx is a next-generation build system with first-class support for mono-repositories and powerful integrations.*[8]

The use of such a tool significantly simplifies the maintenance of JavaScript and/or TypeScript mono-repositories. Some of its other strong points are:

1. The same developers of Nx maintain plugins and similar tools that are easily integrated with an existing repository.
2. Instant generation of internal libraries.

3. The ability to run tests and deployments only on the affected parts, that is, on the modified code, allows limiting the necessary work in Continuous Integration (CI) environments. This involves running tests for the modified code and its dependencies.

4. All created libraries and applications include all the necessary dependencies, scripts, and tools for serving, testing, building, and deploying in a streamlined manner.

5. It provides a rich ecosystem of plugins and utilities within the same base library.

Furthermore, it provides extensive support for the most commonly used libraries and frameworks in frontend development, such as Next.js, the chosen library for developing the frontend application.

Figure 1.2: Nx logo

1.5.3.2 TypeScript and JavaScript

Currently, JavaScript is the most widely used language worldwide for many reasons. However, due to its lack of types, some applications become harder to debug and more error-prone. That's one of the reasons why Microsoft developed TypeScript, which is a strict syntactical superset of JavaScript. Code written in TypeScript is transcompiled to JavaScript during the build time.
By using TypeScript, we provide a highly productive environment when developing the different applications. It not only reduces the number of hard-to-detect errors caused by type issues but also provides all the benefits of ECMAScript.

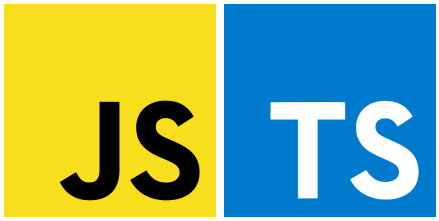

Figure 1.3: JavaScript and TypeScript logos

1.5.3.3 Next.js

As mentioned before, Next.js has been chosen as the front-end framework to build the applications with. This framework is built on top of Node.js, which enables React-based web application functionalities such as server-side rendering (SSR) and static websites. On the Next.js homepage, they provide the following description: *Used by some of the world's largest companies, Next.js enables you to create full-stack Web applications by extending the latest React features, and integrating powerful Rust-based JavaScript tooling for the fastest builds*[9].

Development with Next.js involves structuring the code in a specific way so that the compiler can generate packages in the most efficient manner, avoiding unnecessary code being sent to the client and thus reducing page load time. This feature is known as code-splitting.
React is a framework designed to simplify the construction of web components. As it gained popularity, developers started building full-fledged applications based on React. However, this

meant serving a large amount of JS code to the client, which slowed down webpage loading. Next.js automatically solves this problem without the need for any configuration.

Figure 1.4: Next logo

Development with Next.js involves structuring the code in a specific way so that the compiler can generate packages in the most efficient manner, avoiding unnecessary code being sent to the client and thus reducing page load time. This feature is known as code-splitting.
React is a framework designed to simplify the construction of web components. As it gained popularity, developers started building full-fledged applications based on React. However, this meant serving a large amount of JS code to the client, which slowed down webpage loading. Next.js automatically solves this problem without the need for any configuration.

1.5.4 Unit and acceptance testing

The importance of testing has already been mentioned, and the client is no exception. Testing the client brings the same benefits as testing the server, as it is the part that will be used by the end user. Nx provides plugins to enable testing with different libraries.

Figure 1.5: Jest logo

At the unit testing level, both for the domain code and the component code, the Jest library has been used. Jest is a unit testing and mocking library developed and maintained by Facebook. Currently, it is one of the most famous, if not the most famous, libraries for testing. It provides a powerful CLI with the ability to run tests on modified code. It has also gained popularity for its easy integration with all types of repositories, as its configuration is straightforward.
However, Jest does not provide all the tools to, for example, test component code in an isolated environment. In conjunction with Jest, a library has been used that extends Jest's capabilities. This library is *testing-library*, and its specific plugins for React have been used.

Figure 1.6: Cypress logo

At the acceptance testing level, the repository is set up to run acceptance tests using the cypress.io framework. Cypress is more than just a testing tool. It provides a graphical user interface to see what is being tested, where it fails, and other features. It is an extremely pow-

erful tool for UI testing, which has always been a challenging topic. As stated on its website, *Cypress enables you to write faster, easier, and more reliable tests*[10].

1.6 Stakeholders

In the context of the *Recurrent Travel* project, the goal is to simplify the experience of recurrent travellers through an application that notifies them regarding opportunities in their commonly travelled routes. In order to ensure the highest success in the project, it is essential to identify the parties interested in the project, as well as understanding their need.
The next sections, the interested parties will be identified for the project. It has been decided to organize the stakeholders in a power/interest model[11] in order to simplify the visualization of the information and simplify the analysis. Such model can be widely found in the definition of engineering requirements.

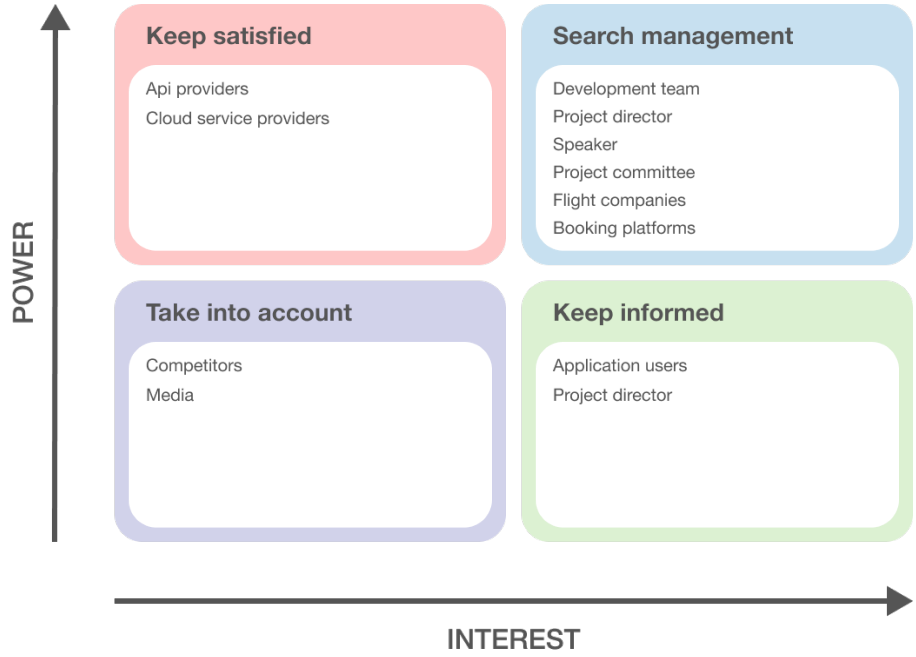

Figure 1.7: Representation of the stakeholders with the power-interest model

- **Lower power and lower interest**
  - **Competitors**: Other companies that provide equal or similar solutions in the market.
  - **Media**: Media that is specialized in the same technology and tourism.
- **Lower power and high interest**
  - **Final users**: Recurrent travellers that will habitually use the application as well as taking advantage of the notifications received.
  - **Families or recurrent travel companies**: people that could be indirectly benefited from the application as it simplifies the fact of keeping a connection with their relatives, or those who can optimize their business model.
- **High power and lower interest**
  - **API providers and external services**: Companies that will provide with the API servers and services that will be used in the application development, yet they are not directly involved with the project.

- **High power and high interest**
    - **Developers**: The team of 4 developers that will work in the design, implementation and deployment of the application.
    - **Project responsible**: The person responsible for supervising the project and providing guidance to the development team.
    - **Speaker**: The project's Master's thesis evaluator, who assesses the performance of the development team and the final outcome of the project.
    - **Project committee**: La Salle URL faculty members who supervise and evaluate the Master's thesis project.
    - **Airlines**: Air transportation companies that could benefit from the market release of the application, due to the utilization of air travel services by the app users.
    - **Booking platforms**: Companies that could integrate the application into their services or that the application itself could integrate into their search systems to enhance the experience of their recurring customers.

1.7 Alternatives analysis

In the current landscape, there are other applications and travel-tech companies that aim to address the needs of frequent travellers and optimize the habitual journeys. The following analysis overviews market alternatives, highlighting their key features and potential differences with the Recurrent travel application:

- **TravelPerk**[12]: TravelPerk is a business travel management platform that offers a solution to plan, book and manage corporate travels. The platform allows companies to centralize the management of employee trips, establishing travel policies, optimize expenses and enhance the traveller's experience. While TravelPerk focuses on the corporate realm and is not specifically designed for recurrent travellers, its cost optimization and travel management offerings could be considered comparable to the ones offered by Recurrent travel.

- **Hopper**[13]: Hopper is an application that uses machine learning algorithms to automatically predict and analize the tendencies of flights and hotels, with the goal of helping users find the best offers. Although Hopper is not focused on recurrent travellers, its focus to price prediction and travel opportunity search could attract users of the Recurrent travel. However, unlike Recurrent travel, Hopper does not allow the possibility to configure customized trips taking into account user preferences of time schedule, frequency and route.

- **Skyscanner**[14]: Skyscanner is a flight, hotel and car renting search engine, which compares prices between the different service providers. Even though Skyscanner is a popular tool to find travel offers, it is not specifically designed to attend the necessities of recurrent travellers, as it does not provide the configuration of time schedule, frequency and rote.

To sum up, although there are several market alternatives in the travel-tech sector, none of them specifically and comprehensively address the needs of recurrent travellers in terms of personalization, travel opportunity search and cost optimization. Therefore, the proposal of Recurrent travel positions itself as an innovative solution in the actual market, with the potential to significantly enhance the experience of frequent travellers.

1.8 Scope

1.8.1 Goals

The master's thesis project Recurrent travel aims to develop a web application that assists recurrent travellers in optimizing the planning and management of their regular trips, offering a personalized and efficient solution for finding the best travel deals based on their preferences. To achieve this objective, the following sub-objectives have been identified:

- Specific needs of recurrent travellers, particularly those who travel frequently for work, family, or personal reasons, are identified and analysed.
- The current market of travel applications and services is investigated to determine opportunities and gaps in relation to the needs of frequent travellers.
- An easy-to-use web application is designed and developed, accessible from various devices, allowing users to configure personalized trips by specifying the route, preferred hours, ideal price, and travel frequency.
- Efficient search algorithms and techniques are implemented to find relevant travel opportunities for users, utilizing Crawling/Scrapping techniques and public APIs.
- A notification system is developed to timely and effectively inform users about travel opportunities that match their preferences.
- The quality and maintainability of the developed software are ensured by applying SOLID principles, Domain-Driven Design, hexagonal architecture, and testing and validation techniques.
- The application is integrated with external systems and cloud environments using continuous integration and deployment techniques to increase software efficiency and ensure the rapid and constant delivery of improvements.
- The application is integrated with different flight search platforms like Skyscanner.
- The security and privacy of user data, including preferences, travel history, and personal information, are guaranteed.
- The application is developed within an agile framework, focusing on iterative and continuous value delivery, with defined delivery times to ensure a timely response to business needs and adaptability to changes in requirements.

1.8.2 Functional requirements

The functional requirements describe the specific functionalities that the Recurrent travel application must provide to meet its objectives and satisfy user needs. The main functional requirements of the project are as follows:

1. **User authentication**: The application must allow users to register and authenticate to access their personal accounts and manage their preferences and recurrent trips.
2. **Recurrent travel configuration**. Users should be able to configure recurrent trips by specifying the origin and destination, preferred dates, and travel frequency.
3. **Flight opportunities search**. The application must periodically and automatically search for travel opportunities that match user preferences, using Crawling/Scrapping techniques and/or public APIs.

4. **Filtering and result operations**. The application must be able to filter and sort search results for travel opportunities, considering the corresponding user and the corresponding alert configuration.

5. **Flight opportunity notifications**. The application must send notifications to users when travel opportunities that match their preferences are found, either by email or through messaging applications.

6. **Travel alerts management**. Users should be able to manage received alerts by marking them as read, archiving them, or deleting them, as well as modify their notification preferences.

7. **Flight booking**. Although it is not necessary for the application to allow direct flight booking, it should facilitate the process by providing links to airline websites or travel providers where users can complete the reservation.

1.8.3 Non-functional requirements

The non-functional requirements are those that describe the quality characteristics of the system, such as usability, performance, security, and scalability, among others. The main non-functional requirements of the Recurrent travel project are as follows:

1. **Usability**. The application must be easy to use and have an intuitive and appealing interface, with a clear and consistent navigation structure and presentation of information that facilitates user understanding and decision-making.

2. **Performance**. The application must be capable of performing travel opportunity searches quickly and efficiently, offering results in a reasonable time, even during high demand periods or with numerous concurrent users.

3. **Security**. The application must protect users' personal information and preferences by applying appropriate security measures, such as data encryption, authentication and authorization, and prevention of common attacks like SQL injection or cross-site scripting.

4. **Availability**. The application must be available at all times, ensuring high uptime and the ability to recover quickly from possible failures or service interruptions.

5. **Scalability**: The application must be able to scale to support the growth in the number of users and resource demands, both in terms of infrastructure and functionalities. This involves the ability to add new servers, load balancing, and improving application performance as necessary.

6. **Interoperability**. The application must be able to interact and integrate with external systems, such as airline APIs, using common standards and protocols.

7. **Maintainability**. The application's source code must be easy to understand, modify, and maintain, following established design principles and patterns, such as hexagonal architecture, Domain-Driven Design, and SOLID principles. Additionally, version control practices and adequate documentation should be implemented to facilitate collaboration and long-term project maintenance.

8. **Privacy and compliance**. The application must comply with applicable laws and regulations regarding the protection of personal data and user privacy within the European framework.

1.8.4 Risks and barriers

During the development process of the Recurrent travel project, several challenges and barriers may arise that, if not taken into account, could result in project failure or hinder its success. The following is an analysis of the main challenges and barriers that may be faced:

1. **Changes in project requirements**. Changes in project requirements may occur throughout the application development, whether they are functional or non-functional, requiring adjustments in the design and implementation. These modifications could delay project progress and increase its complexity and associated costs. To address this risk, it is essential to properly plan and prioritize requirements, maintain constant communication with stakeholders, and be flexible in adapting to changes.

2. **External applications integrations**. The Recurrent travel application will heavily depend on integration with external APIs and services from airlines and travel providers. Access to these systems may be limited, unstable, or change without prior notice, which would affect the application's functionality and create issues in obtaining up-to-date and accurate information. To mitigate this risk, it is crucial to establish recovery mechanisms in case of failures, perform frequent integration testing, and consider the possibility of using multiple data sources to ensure information availability.

3. **Scalability and performance problems**. If the application experiences a rapid increase in the number of users and resource demands, scalability and performance issues may arise, negatively impacting the user experience. To address this risk, it is critical to design and implement a scalable architecture, using appropriate techniques and technologies such as cloud computing and load balancing, that allow adaptation to the changing project needs.

4. **Privacy and compliance**. The Recurrent travel application must comply with applicable laws and regulations regarding the protection of personal data and user privacy, which may pose challenges in implementing security measures and respecting users' rights. To tackle this risk, it is essential to stay informed about applicable regulations and incorporate privacy practices from the design stage, ensuring that the application meets legal requirements and safeguards user information.

5. **Time and resource limitations**. The master's project has a limited time for completion and may face constraints in terms of available resources, such as the development team's time and allocated budget. These limitations could affect the ability to successfully complete the project within the established timeframe. To address this risk, it is crucial to conduct proper planning, set realistic goals, and prioritize tasks based on their importance and urgency.

6. **Market competition**. The travel and transportation applications market is highly competitive, with numerous players offering similar solutions. This could make it challenging to differentiate "Recurrent Travel" and position it in the market. To tackle this risk, it is essential to identify and communicate the application's competitive advantages, such as its focus on frequent travellers. Additionally, staying informed about market trends and advancements, as well as being willing to adapt and continuously improve the application based on user needs, is crucial.

1.9 Working methodology

In further sections, the working methodology is explained in more depth. Nonetheless, from the early beginning, the team decided to divide the work in responsibilities. This division has

allowed the team members, each with different areas of expertise, be able to provide the project with their knowledge in such areas.
There have been bi-weekly meetings, with the goal to share to the team and the team tutor what has been done during the last two weeks, as well as planning what should be done the following. Furthermore, each meeting was an opportunity to discuss with the tutor some of the project features and functionalities.

In terms of development, the team has organized through the GitHub Projects tool. With this tool, it has been able to plan the tasks in a style that combines elements of Scrum and Kanban. Being integrated with GitHub, it allows for seamless interactions between repositories and the project. For example, it provides automation for moving tasks from one column to another.
Since the team has worked mostly asynchronously, the usage of the board has been crucial to give a view of the project status, as well as identifying what elements were taking too much time, that were blockers of other tasks.

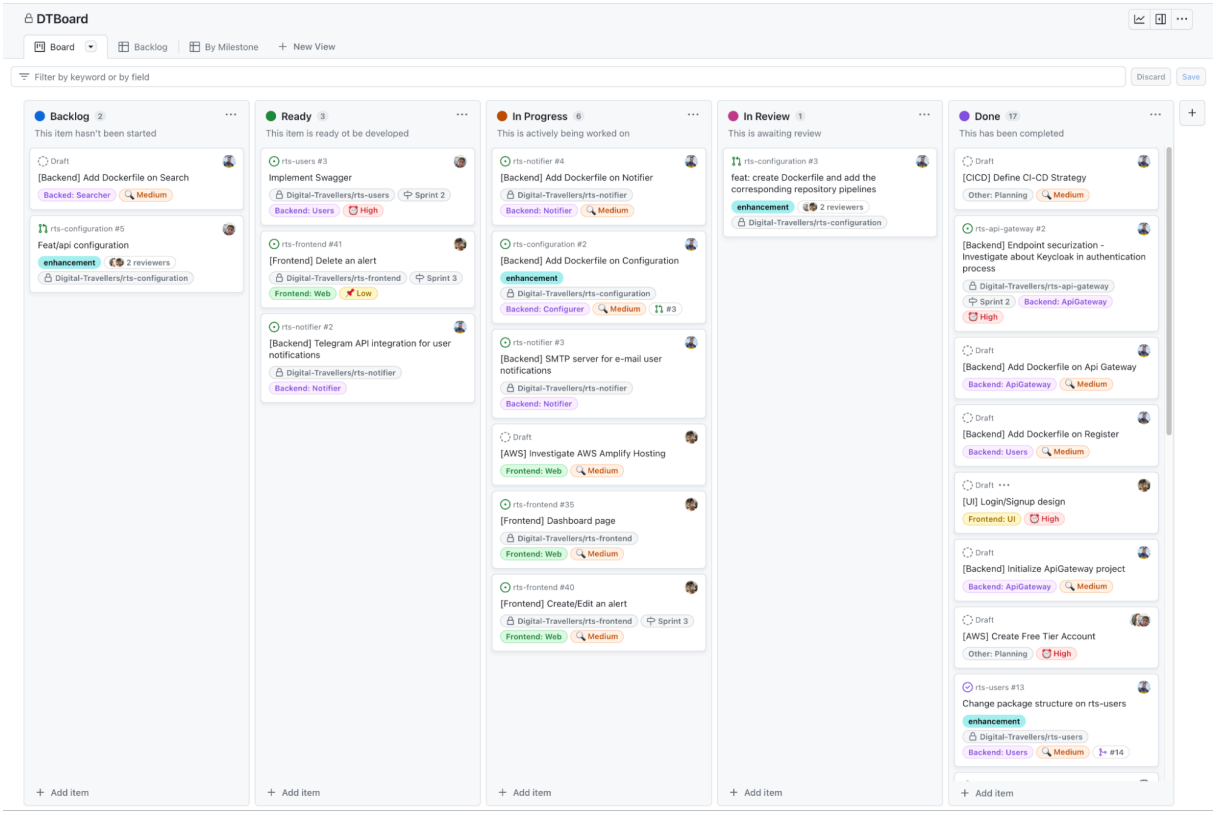

Figure 1.8: Screen capture of the GitHub Projects tool

2. Team organization

2.1 Task division

Recurrent Travel is supported by a team of four professionals responsible for the application development. One has expertise in frontend, one in full stack development, and the remaining two have expertise in backend development and dev-ops. Despite each member having specific expertise in their respective fields, the team is cross-functional, and the project development can be carried out by them without the need for external dependencies.
The project consists of the following elements:

- A user interface application to allow the user to interact with the application.
- Four services that include the backend server logic, database management and data scrapping.
- A backend communication system, such as a queue, to allow the different services to communicate between them.

Taking into account this proposal and the abilities of each team member of the team, the division of the responsibilities is distributed considering that the majority of the work focuses on backend development. Therefore, most part of the resources are assigned to the backend, with specialized members leading and executing tasks related to such area. The member with strong knowledge in interface design and frontend development will be assigned to the development of the user interface, user experience and frontend business logic. The planning and overall structure of the project are taken care by the member with knowledge in software architecture and design. Code quality, testing and similar aspects are elements that each member will have to handle by themselves. Lastly, the entire development is overseen and coordinated by a project leader.

With the previous parameters in mind, the initial planning consisted in the creation of the tickets in order to be able to accomplish the expected tasks, which were defined every two weeks. It is also worth mentioning that tasks could vary in time, as it is really hard to detect possible problems or issues in the development, when planning the tasks.

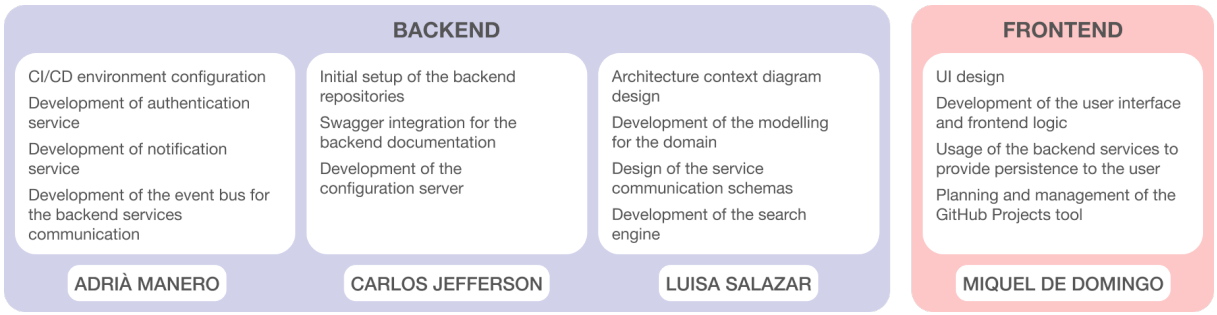

Figure 2.1: Task distribution as a schema

It is important to point out that during the development of the process each member of the team is not exclusively focused on the tasks described above, rather it contributes and collaborates to the execution of the assigned tickets to other members, when help is needed, helping diminish the handicaps of the project.
At the same time, there are tasks that are executed by all members of the team and are part of the global work required to develop the project:

- Definition of the rules in terms of code (using tools that validate a certain style of code, also known as linters) and to contribute to each repository.
- Planning of the expected work to be finished for each sprint.
- Code review of each developed task, in order to provide a second opinion in the code *to be merged*.
- Study, evaluation and proposal of technologies both for development and deployment of the systems to be developed.

Being more specific, the following are the tasks for each team member.

Miquel de Domingo - Frontend

- **Website design**. Involves all the proposed user interface designs that the application will have, including responsive designs for small screen devices, since the application will be accessed through the browser.
- **Configuration of CI and CD**. This task involves design and execution of strategies that provide frontend repository with continuous integration and deployment strategies.
- **Interactivity control**. As the frontend developer, along with the design, it should be capable to structure a coherent flow of activities and actions that the user will be taken through while interacting with the application, leveraging and experience.
- **Usage of REST APIs**. The frontend application should be capable to communicate with the backend services in order to provide the expected user experience in terms of persistence.
- **Planning and management of GitHub Projects**. As a team that uses agile methodologies to make ends meet, this task involves the member to structure the team's rules and finding tools that align with the chosen methodology.

Adrià Manero - Backend and DevOps

- **Configuration of CI and CD**. This task involves design and execution of strategies that provide backend repositories with continuous integration and deployment strategies.
- **Development of the user auth**. The service should be responsible for the direct communication with the frontend user, with the usage of an API gateway to redirect requests to other backend services.
- **Development of the notification service**. The service should be responsible for triggering the expected notifications received by the search service, for each created alert.
- **Development of the event bus**. This component will allow backend services to subscribe to message queues and trigger events for each request based on the queue.

Carlos Jefferson - Backend

- **Backend repository setup**: To maximize the shared expertise in Java and Gradle among the backend professionals, each service was placed in repositories with consistent configuration templates, providing a standardized starting point.
- **Swagger integration**. The Swagger library should be incorporated in order to simplify the asynchronous development between the frontend and the backend. Thus, the swagger docs should be the only source of truth.

- **Configuration service creation**. This service manages user registration of recurrent trips and generates alerts for the search service. It also handles persistence of user-defined recurrent trips.

Luisa Salazar - Backend

- **Context diagram**: This task involves the definition of the diagrams that will provide a general view of the architecture of the systems, including the frontend application to the backend services.
- **Domain layer modelling**. Involves defining the schema that will be used in the database, as well as modelling such entities in the domain layer, including their logic, for each service that requires it.
- **Communication schema definition**. It is important to have defined the JSON schemas that will be used to communicate between the backend services. This JSON schemas will benefit the development of the request/responses sent to the message broker.
- **Development of the search engine**. The search engine should be able to perform the search and obtain results, if any, for the travel alerts created by each user. At the same time, it is responsible to send the results obtained to the message broker in order to be able to send notifications to the user.

2.2 Project management system

As a project management approach, the development team followed an Agile methodology combined with an adapted Kanban approach. This allows the creating of a comprehensive project management approach philosophy tailored to the team and a concrete plan for delivering a high-quality project.
This decision was made taking into account the necessities of the project, the team and the approach to develop the project. Each of the member of the team already has experience working with agile methodologies and are familiarized with the principles of it. The usage of this methodology aims to:

1. Encouraging collaboration between the members, as the methodology is collaborative in its nature, promoting teamwork and shared decision-making.

2. Ensuring a fast delivery of results, as it promotes constant and time effective delivery cycles. This allows for early feedback and continuous improvement.

3. Valuing people over processes, acknowledging the diversity of team members and adapting in each iteration, in order to accommodate individual circumstances and needs.

2.2.1 Agile but not scrum

It is important to note that the agile methodology has been adapted to the development of the project, as it is not tied to Scrum, because of the team work dynamics, time constraints and asynchronous availability. Scrum methodologies imply more synchronous ceremonies which the team was not able to satisfy. Nonetheless, the team has successfully had biweekly meetings, which included work presentation, revision in order to obtain feedback for what has been developed, and planning for next meetings.

Consequently, the team has been based in a Kanban methodology, which suits perfectly to the needs of the project, as it provides a truly simple approach to find an equilibrium between work that needs to be done and the availability of each team member. The Kanban methodology it

is based in a continuous improvement methodology, in which tasks are extracted from a list of pending actions from the constant workflow.

To sum up, for the current team, the combination of both methodologies has allowed us to develop fast, asynchronously and in parallel.

2.3 Project management tool

In line with the project management methodology adopted, the team also relies on a tool to aid in the planning and to find a more efficient management of the work to be carried out. The usage of such task-tracking tool, it simplifies the creation, management and listing of task to do, in progress and completed.
By using this tool, the team can streamline the planning process and ensure effective and asynchronous time management. The team and each member are able to prioritize the tasks to be completed in each sprint, track their progress and monitor the status by looking at the board. This approach helps each member to have a better organization of the work, improving efficiency, and enhancing collaboration. The task-tracking tool simplifies task management and provides a visual representation of the *state of the work*.

There are many different tasks-tracking tools and systems, both for agile and non-agile project management. However, the team has prioritized keeping the project simple and finding the maximum automation with the minimum configuration, as key aspects for the tool to choose. This has lead the team to opt for GitHub Projects, an integrated tool within the GitHub application, over what could be considered its major competitor, Jira.

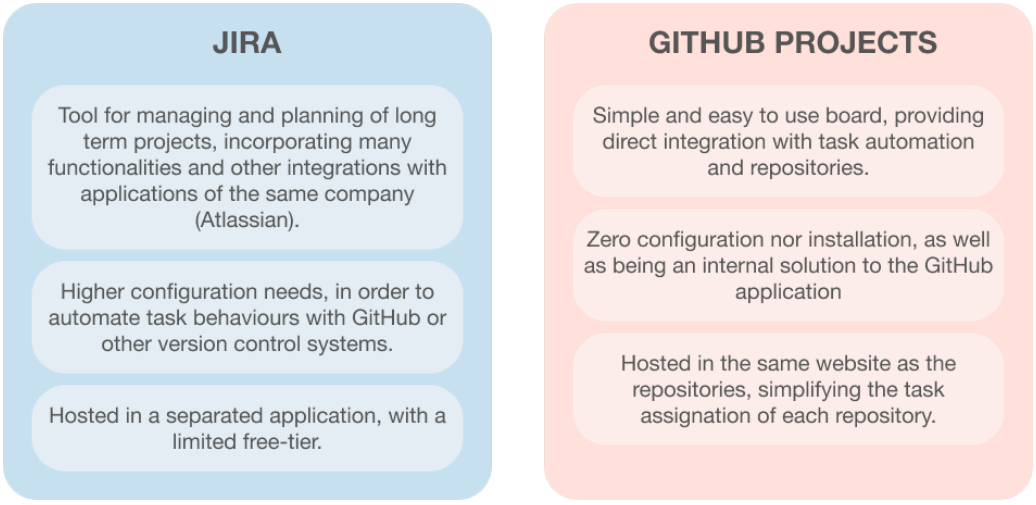

Figure 2.2: Comparison between Jira and GitHub projects

The features of Jira simplified in the above chart, present it as a task-managing tool for more complex projects. However, considering the specific development characteristics of this project, it is clear that the time constraints play an important role in preferring one tool over the other. This is a short-term project, and requires a management tool that primary allows the organization of issues and provides an easy visualization. Nonetheless, GitHub projects can be scalable to bigger projects, which would allow future teams to keep using it as their task-managing tool. Therefore, the features provided by GitHub Projects's boards, as an adaptable and flexible tool that enables tracking and planning of work within the code repository, are ideal for the execution of this project. It also facilitates the organization of tasks and provides a straightforward way to visualize the status of their resolution.

2.4 GitHub Projects

2.4.1 Team organization

In order to properly develop the project, the GitHub organization members have been divided in two teams. The *backend* team consisting of the three[1] members with backend related tasks. The *frontend* team consisting of the member[2] with frontend related tasks.

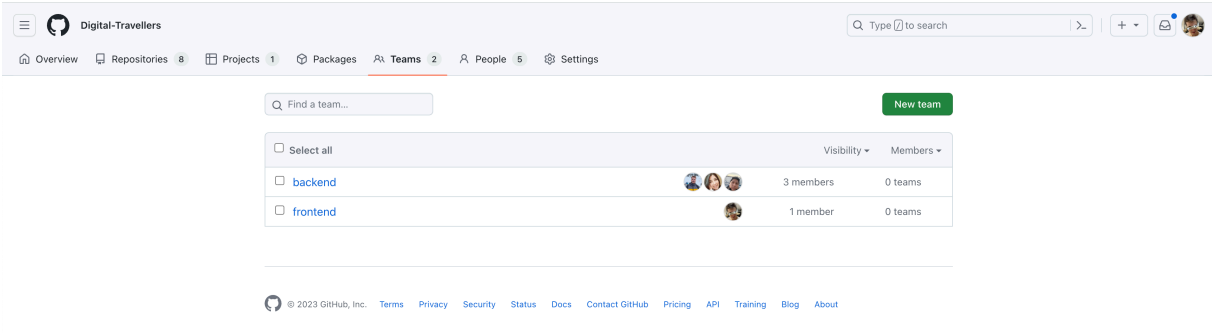

Figure 2.3: Comparison between Jira and GitHub projects

The goal of having projects within the project is to reflect the structure of the team with mentions and permissions in a cascade format. The members of each team can send notifications to an entire team or ask for reviews for a team. There, each member of the team will receive the notification. GitHub's organizations allow the creating of teams, which can have subteams within. In the current team structure, such nested level is not required.

2.4.2 Board view

The board of a project is what allows the developers to visualize the state of the project. It can accept from issues defined in the board (not linked to any repository), issues linked to a repository and even pull requests, specific for a repository as well. It also allows filtering and grouping by many different options. This filtering and grouping can also be stored as views in a separate tab.

[1] Adrià Manero, Luisa Salazar and Carlos Jefferson
[2] Miquel de Domingo

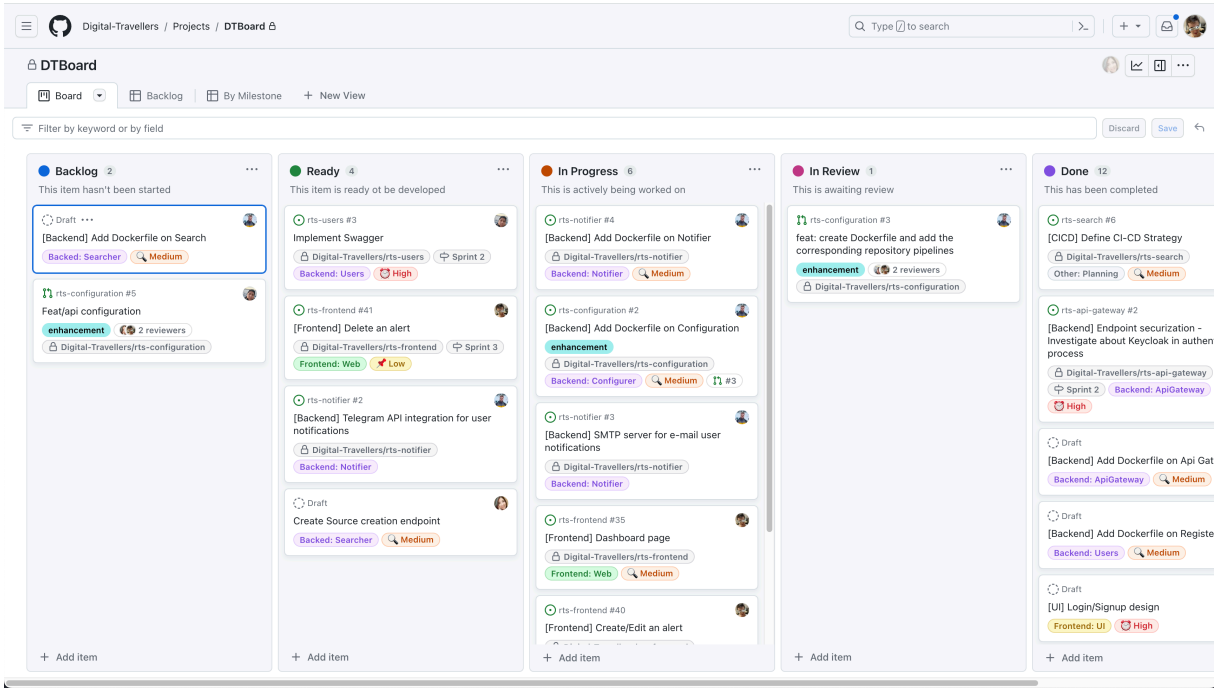

Figure 2.4: Visualization of the board

Since GitHub does not have the concept of sprints, the team has used milestones as an alternative to defining the sprints. Even though milestones are repository specific, they can be grouped for different repositories as long as they have the same name.

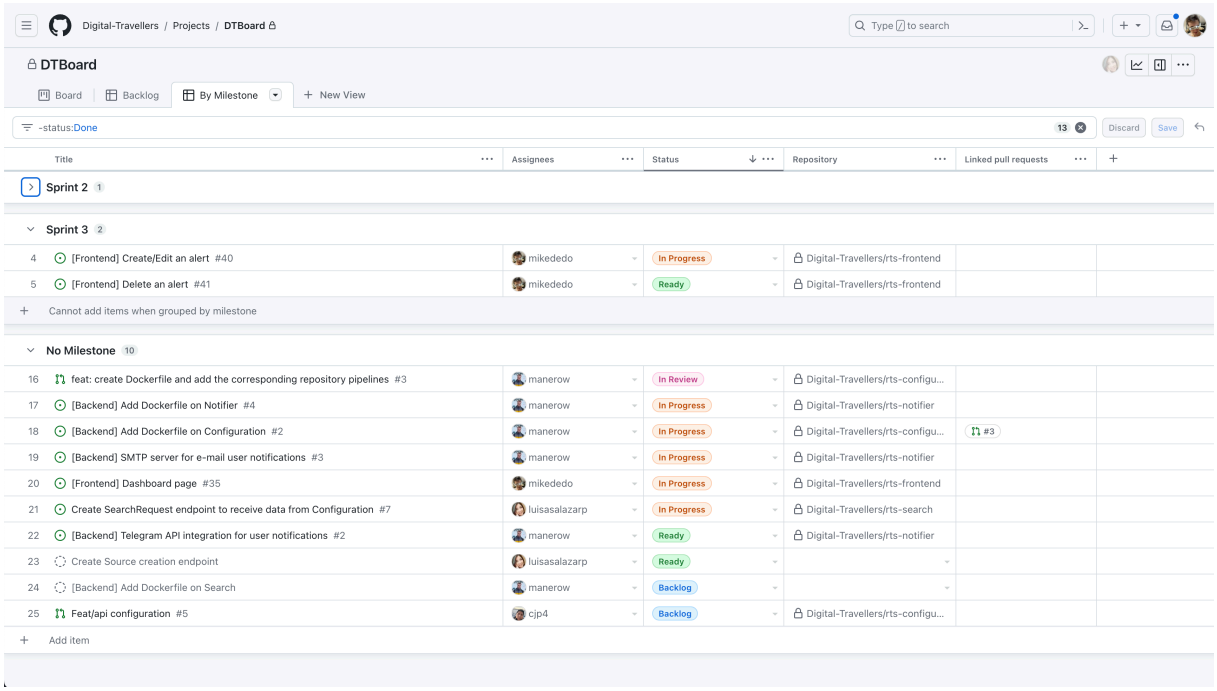

Figure 2.5: Visualization of the board, grouping by milestones

Another benefit of directly using the board from GitHub is the possibility of constantly being up-to-date with the project status, as the project will also be updated with the activity (creation of pull requests and issues) of each repository.
It also allows having personalized field for each ticket, allowing the team to add extra infor-

mation just with the visualization of the ticket. In this case, the following meta-information has been defined for each ticket:

- Status. The status allows the developers to know if the ticket is in the backlog, ready to be developed, in progress, in review or waiting to be reviewed, and done, in case the ticket has been merged successfully.
- Project. A small tag that gives information to which project the ticket is related.
- Priority. It allows knowing which tickets should be prioritized over others.

The addition of such fields is fairly simpler than its competitors.

3. Cost analysis

After conducting a thorough planning and being conscious about all the tasks that are required to achieve the expected result, this chapter contains an analysis of the costs of this project, detailing the different investments that could be made.

It has been decided to divide investments dedicated to the project into two main categories: human related resources and non-human related resources.

Finally, provisions for contingencies and any unforeseen expenses that may arise are also taken into account.

3.1 Human costs

In any project, one of the key factors and usually the most expensive is the personnel that will bring such project to live. In this case, the project is being developed by a group of students, as part of their degree's curriculum, meaning that there is no direct cost linked to the work done.

So, in order to analyse the human costs for this project, it is important to consider the different actors involved in it, and the time that will be spent in the project.

1. **Students**. Students will require the major part of the budget. As established by the university, the final project equals to 10 ECTS credits, which corresponds to a time expenditure between 250 and 300 hours, each student. In order to simplify the calculation, it has been supposed that each student will spend 275 hours.
2. **Tutor**. On the other side, during the development of the project, there were a set of scheduled meetings that involved the students and the tutor of the project. The project lasts around 22 weeks, and it has been estimated that, including the work outside the meetings plus the meetings, the tutor would spend around 3h each week.

Actor	Hours/actor	Num. of actors	Total hours
Student	275 hours	4	1100 hours
Tutor	66 hours	1	66 hours
			1166 hours

Table 3.1: Human resources costs

Given this information, the total cost of each actor can be calculated through:

$$T_c = C_{ha}(€/h) * N_h$$

In this equation, T_c stands for the total actor cost; C_{ha} stands for the cost for each hour of the actor working; N_h stands for the number of hours to work.

It should be considered that, outside the academic scope, the costs for a project inside a company are incremented due to social security taxes.

3.2 Non-human costs

The non-human costs of a project refer to all those that are not directly associated with the individuals participating in its development but are equally necessary for its successful com-

pletion. These costs encompass a wide range of material resources and services required to carry out project tasks.
In the case of the current software development project, non-human costs can be classified into three main categories: hardware costs, infrastructure costs, and indirect costs.

3.2.1 Hardware

The development of an application or a software design is required of a robust and reliable infrastructure. This infrastructure is essential for the implementation, testing and deployment of the application. The hardware costs can vary in price and include an extensive variety of items for the development and the testing of the product.
In order to calculate the hardware costs, it has been taken into account the indispensable for the project development. In terms of deployment infrastructure, free-tier subscriptions have been used, meaning there is no initial need for an investment in terms of infrastructure. For the hardware elements, their life span and their depreciation has also been taken into account.

Hardware	Life span	Aprox. unit cost	Total cost
Personal computer	10 years	2,000 €	8,000 €
Mouse	5 years	20€	80€
Keyboard	5 years	50€	200€
Headset	5 years	40€	160€
			8440€

Table 3.2: Hardware costs

Given this information, the total cost of each actor can be calculated through:

$$T_c = C_u(€/h) * N_a$$

In this equation, T_c stands for the total cost; C_u stands for the cost for each unit; N_a stands for the number of actors to work.

3.2.2 Infrastructure costs

In terms of infrastructure, the costs correspond to the cloud services offered by: Google Cloud for the backend applications, and Vercel for the frontend. These tools offer a free-tier or a free-credit tier, for educational purposes. The team has taken advantage of these tiers to deploy the application.

Google Cloud offers a $300 in credits to be used for 90 days. This allows the usage of an extensive catalogue of services. In this case, the Google Kubernetes Engine (GKE) is the only service required. This service consists of a Kubernetes cluster self-managed.

Vercel, on the other hand, offers a free plan which is targeted to personal and educational projects. It includes unlimited bandwidth for static sites, 100 GB for dynamic sites, as well as offering unlimited deployments and serverless functions.

Service	Duration	Pricing	Total cost
Google cloud	90 days	$300	0€
Vercel	Unlimited	$0	0€
			0€

Table 3.3: Infrastructure costs

3.2.3 Software expenses

The development of a product does not only imply associated costs to the infrastructure or hardware, but also to the tools and applications that allow the developers and teams to do their job. This can include operative systems, IDEs, project management software, databases, software libraries, frameworks, and so on.
It is important to note that such costs can vary a lot depending on the specific needs of the project, and the decisions of the development team. In the case of the project, given its academic focus, the development team has taken advantage of free tools in order to minimize the costs.

Software	Cost
Operative System	0€
GitHub Organization	0€
Discord	0€
IntellIJ Idea Community Edition	0€
Visual Studio Code	0€
NeoVim	0€
Docker hub	0€
	0€

Table 3.4: Software costs

3.2.4 Indirect expenses

In the execution of the project, it is not relevant to take into account costs directly linked to the production of the final project, yet also include the costs that are indirect to the development of the project: the indirect costs.

The indirect costs are crucial in the economic evaluation of a project, otherwise, their omission can be translated in an underestimation of the required resources.

3.2.4.1 Electricity

In the execution of any project, it is not only relevant to consider the costs directly linked to the production of the final product, but also those expenses that, although not directly related, are necessary for its successful completion. These are identified as indirect costs.

Indirect costs are essential in the economic evaluation of a project, as their omission can lead to an underestimation of the necessary budget. Identifying and quantifying them will allow for

more precise resource management and effective planning. Among these indirect costs are the consumption of electrical energy, which includes the use of computers and workplace lighting; internet connection, necessary for software development; rental of workspace; office supplies such as notebooks and pens; and potentially, additional training and education. All of these factors, although indirect, are crucial for the proper execution and planning of the project.

Indirect cost	Unit cost	Total cost
Electricity usage	X	0€
Internet connection	X	0€
Workspace rent	X	0€
Office material	X	0€
Additional training	0€	0€
Travel expenses	X	0€
		0€

Table 3.5: Indirect costs

4. Planning

The purpose of this chapter is to provide an overview of the planning, organization, and development process of the frontend application. Planning plays a vital role in ensuring time efficiency, making it essential to engage in thorough planning before commencing development. Additionally, creating a minimal user interface (UI) prior to the actual development phase can significantly expedite the process.
It is worth emphasizing that the planning stage should also allocate time for thesis writing, as it constitutes an integral part of the overall project.

4.1 Sprint planning

When planning the work for the frontend project, our main goal was to avoid being blocked by backend progress. It is common for frontend and native developers to fall behind schedule because they are unable to progress until the backend is complete. One solution to reduce this lag between frontends and backends is to use GraphQL. However, in our case, we opted to apply GraphQL ideas in a RESTful API, resulting in the following steps:

- Defining the expected request body of a specific endpoint, in order to let the frontend know *what the service expects to receive*.
- Defining the response, if any, that an endpoint would return for a given call.

This approach established a clear contract between the backend and the frontend, with separated implementations that both parties agreed to respect.
To facilitate development, we used a service worker to mock every API call made in the local environment of a frontend developer. The service worker respected the specified contract, allowing us to develop the frontend without being blocked by the backend. Once the app was deployed, the service worker was disabled, and the API calls were made to the *actual* RESTful API. This approach allowed for fast and non-blocking development for both frontend and backend.

The second thing to take into account before starting to structure and develop the frontend are the designs. Since there was not enough time to create a system design and then enough designs that would cover all use cases of our app, we opted for simple designs, which are shown in later sections. As a first sprint, or sprint zero, I was in charge of prototyping the frontend application, which would simplify the process of developing the frontend app.

Next steps are more involved in the development and architecture of the application. As explained in previous sections, we got together every two weeks, as well as keeping an asynchronous communication. We considered each meeting to be a deadline, and in the meeting we would discuss the next steps to take or changes if any. However, since there are always unexpected tasks, it has not been easy to strictly follow the devised roadmap. The following diagram illustrates an approximate planning of the sprints.
It is important to note that this initial planning is really subjective to changes, as in any app development, there may be incidents, bugs, or other issues that could temporarily block the development of the application. Nonetheless, the goal was to stick as much as possible to the expected plan. Also, Sprint 0 has not been added to the schema since it can be merged with the tasks in the Sprint 1.

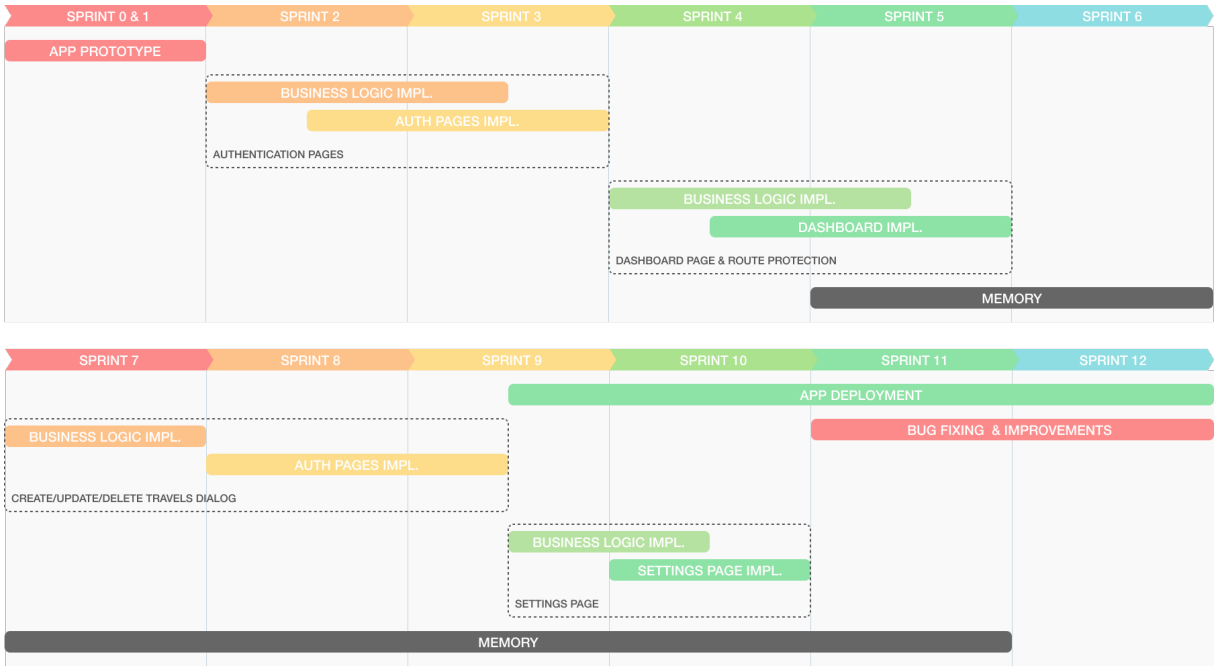

Figure 4.1: Initial sprint planning

Aside from the Sprint 0 and 1 section, other sections consist of two weeks.

4.2 Sprint development

4.2.1 Sprint 0 and 1

As explained previously, the goal of the sprint 0 was to start prototyping the application. Since there was no design system defined, it took a bit more than two weeks to finish the prototypes, even though there were some part that would change in a future.
The design not only helped us visualize the entities and important parts of our application, but also helped me architecture the monorepository. Once tool used to simplify the developer experience is Nx. Using such build system, tasks as having internal libraries and separated applications are easily handled. Therefore, the goal of the sprint 1 was to set up the repository and have it ready to roll. As I was defining the architecture of it, I was able to test possible use cases that I could come across while developing the frontend. This helped me prevent possible time-consuming issues, by handling them in an early stage.
One of the requirements for developing the application was to follow hexagonal architecture patterns and domain-driven design. However, frontend frameworks and technologies often deviate from traditional structures. For example, the React framework primarily uses composition over inheritance and follows a reactive paradigm. As a result, applying these concepts to the frontend posed a significant challenge. More details about the final design will be provided in subsequent chapters.

4.2.2 Sprint 2 and 3

Once the architecture was defined, the first goal of the sprint was to develop the frontend domain logic for user sign up and sign in. As the first code written for the core frontend, it was important to establish a solid structure that could easily accommodate changes. The initial architecture planning proved successful, with the following benefits:

- The hexagonal architecture layers (domain, application, and infrastructure) could be easily decoupled.

- The structure could be easily refactored and scaled as needed.
- The domain logic was separated from the frontend implementation, making it reusable.

Once the domain was defined and ready to be connected to a frontend application, the next step was to implement the authentication pages. This process was divided into two steps:

1. Since most components had yet to be created, an initial implementation was created in the respective package. This implementation was open to modifications and designed to be reusable within different applications, although it was already attached to the React framework.
2. With the components created, and the business logic defined, the last step was to connect both through the view layer.

Since there was not a lot of logic involved, it was possible to stick to the original plan, and start developing the frontend views (log in and register pages).

Once the logic architecture had been defined, the goal of the sprint was to start developing the domain of the frontend that will contain the logic to sign up and sign in a user. Being the first code written in the core of the frontend, it was very sensitive to change in terms of structure. However, the initial architecture planning turned out to work seamlessly well, this being:

- Easily decoupling the different layers from the hexagonal architecture (domain, application and infrastructure).
- Simplicity to escalate the context as well as to refactor it.
- Containing only business logic code, being completely unaware of any frontend implementation, which allows such logic to be reused.

Once the domain have been defined and was ready to be connected to a frontend application, the next step was to start implementing the authentication pages of the design. This process was divided in two steps:

1. Most of the components had yet to be created, therefore an initial implementation, open to modifications, was created in the respective package. Note that, even though it is only used in one application, another idea of the components internal library is that it can be reused within different applications. However, it is already attached to a framework, which, in this case, is React.
2. Having the components created, and the business logic defined, the last step was a simple as connecting both through the view layer.

Authentication pages require of a very basic layout, and most of the logic happens in the form. However, as the logic had already been implemented in the core package, such logic required only to be connected with the components. Therefore, it simplified a lot the development of the view.

Initially it may not seem as an advantage, however if ever appeared the necessity of developing another application that required the same logic, it would mean that it could be reused. Therefore, the developers could only focus on the implementation of the frontend, and then attaching the logic to it.

4.2.3 Sprint 4 and 5

For most of the tasks, to development of them involves a first implementation of the business logic, a second part which may be optional which involves the development of the required components, and a third one that involves applying the logic to the view.

Similar to the authentication pages, the first task of the dashboard page involved creating the initial travel mode. In the dashboard page, the user can see a list of travels and interact with them. The interaction or management would be later developed. The travel model is quite complex, as it is fetched from the backend in a specific model, meaning that it should be parsed and transformed to the expected object.
Another setback that appeared during the development of the logic for the dashboard was the travel model changes, which implied having to add, update or delete all fields that differed. As some parts of the dashboard already included the usage of the initial proposition of the travel model, the specific parts had to be updated.

This exposed one of the common problems between the core package and the frontend implementations that use that package. Any breaking change in the core, as it is the update of a model, can generate side effects to the frontend that depends on it. Knowing this issue, I came up with two solutions:

1. First would be to create a versioning of the packages, meaning that the frontend would depend on a concrete version of the core package. At start, it could be a good solution, yet it can be really easy to start leaving frontends outdated. The update of the frontend is not inevitable, but it would allow the team to update the changes in the main branch, and migrate each frontend step by step.

2. Second would be to work in feature branches, meaning that for every breaking change in the core package, all the affected applications would be updated accordingly. This implies that the team would have to focus on the migration in order to accept the new changes.

In this case, second option was chosen as versioning the packages was not something planned from the starting point and, even though it is possible to be added, updating all frontends was more simple. Additionally, only one application had to be updated.

4.2.4 Sprint 6

In the sixth sprint, the team agreed to focus in writing the common parts of the memory for the thesis. The goal was to finish as soon as possible such common parts and later start writing non-common parts.
Even though it was not a complex part, the team required of more coordination than usual to start writing the common parts. A part or some parts were assigned to each member, and a shared Google Docs contained all the parts.
After a group revision, a couple of changes needed to be made, meaning that the memory common parts task would expand to more that the one sprint, as initially planned.

4.2.5 Sprint 7 and 8

These sprints were planned to be focused on the development of the travel management from the frontend. However, as a team we agreed to *push* the memory and finish the common parts as soon as possible. This resulted in stopping the development of the frontend temporarily.

Once the common part had been written or more advanced, the team could focus back on the development. As explained in any previous task, first came the modelling of the alerts. Inside

the core package, there was already a lot of business logic for the alerts, which required only to be extended by adding the possibility to create, update and delete the travels.
Next, only a couple of components were created or required modification/extension. Inputs, buttons, text and so on had already been created as were required by the other tasks.
Finally, the connection between the logic and the frontend components. In further chapters it will be explained in more depth, yet this tasks also require a bit of state management. The library used to manage API calls and caching such calls is also used as a state management solution. This helps reduce the logic to be implemented in the core application, as it can be kept stateless.
All this tasks include the respective testing, both in the core and in the UI components.

Even though the development of this sprint seems really straightforward, it included more frontend logic than the other tasks. Basically, any action (creation, update or deletion) of an alert should update the dashboard table. However, as a commonly known problem in React it is the amount of rendering that must be done, for any state changes. This topic will be explained in later chapters, yet the architectural decisions that impacted performance were made during these sprints.

4.2.6 Sprint 9 and further

As the memory took up part of the previous sprints, it was at this point that the initial planning started to tumble. As explained in the beginning, the roadmap was just an idea to align with the rest of the team on what to focus. Nonetheless, during the majority of the time, it was possible to stick to the roadmap.

The management of travel tasks required more time for full development, resulting in an underestimation of the total workload. As a result, these tasks were not resolved until the 12th sprint. Another factor contributing to the extended time required was the simultaneous development of the automated application deployment process. This deployment is a critical task as it allows access to the MVP outside the local development environment. Additionally, as a team, we needed to ensure that all connections, from the frontend application to the backend application, functioned seamlessly.

With all this in mind, there was very little, to any time to develop the settings page. Also, since there was no service nor endpoint to update the user settings (also because of time constraints), such task was not developed.
Furthermore, end-to-end tests were also skipped because of time constraints.

5. System design and decisions

5.1 Backend structure

As mentioned previously, the project includes a backend component that encompasses a significant portion of the content, which is why three team members are dedicated to backend development. The objective of this section is to provide an overview of the backend architecture and structure without delving into extensive implementation details.
Although I did not actively participate in the backend implementation, the team collectively made decisions regarding its architecture and structure. Once the application requirements were defined, the team realized that the backend could be structured as either a monolithic application or a microservice architecture. Ultimately, the team opted for a service-based structure due to the following reasons:

- Simplified asynchronous development: With each service contained in a separate repository, team members could focus on their respective projects independently. This streamlined asynchronous work and reduced the likelihood of conflicts arising.
- Faster CI/CD processes: Each repository has its own set of tests and deployment processes. By separating each project, the time required to run tests and deploy applications was significantly reduced, as less code needed to be checked, compiled, and built.

A service-based architecture typically involves communication between different services or applications. This challenge is commonly addressed by implementing a message queue, which allows backends to subscribe to and publish events.

Another important aspect about each backend service design is that it has been written using a hexagonal architecture approach, combined with domain-driven design.

5.1.1 API gateway

The backend consists of multiple services, with each service having a specific functionality. However, it is important that clients using this backend are unaware of its internal structure. If clients were aware of each individual domain and had to make requests accordingly, it would create an undesirable dependency for the frontend.

To address this issue, the team implemented an API gateway as a solution. This gateway acts as an intermediary, receiving requests from the frontend and determining which service should handle each request. Additionally, the API gateway handles user authentication for the frontend and ensures that only authenticated requests are allowed, blocking unauthorized access.

The API gateway provides several advantages and is an optimal solution in terms of scalability and migration management. Firstly, it simplifies the integration process for all connected frontends by maintaining a consistent contract for each request. Any changes made to the backend services will not directly impact the frontends, as long as the contract remains intact. This allows for seamless updates and modifications to the backend without disrupting the functionality of the frontends.

Furthermore, the API gateway provides a layer of abstraction, shielding the frontends from the complexities of the backend architecture. Even if there are changes behind the API gateway, such as consolidating all microservices into a monolithic structure, the frontends remain unaffected as long as the contract between the API gateway and the frontends remains unchanged.

In summary, the implementation of the API gateway not only solves various challenges but also

offers a scalable and resilient solution that decouples the frontends from the backend services, ensuring flexibility and ease of maintenance in the long run.

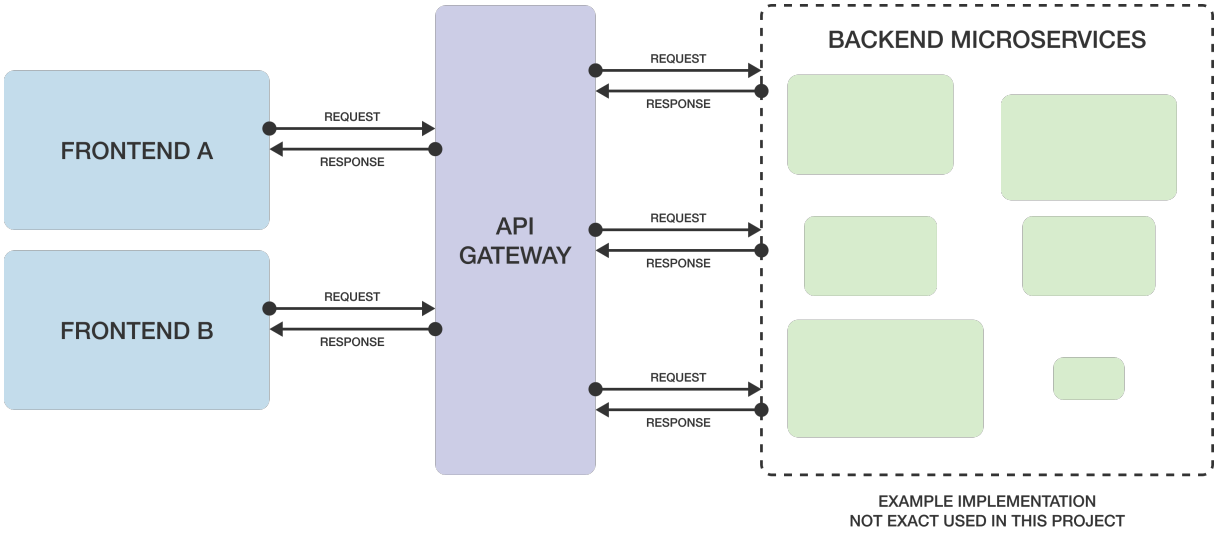

Figure 5.1: Example benefits of the API gateway

The above figure exemplifies the benefits of the API gateway. On the one hand, we can have as many frontends as want connected to the gateway. It works as a façade to what is behind. On the other hand, the microservices will also only communicate with the gateway, satisfying the contract specified.

5.1.2 Service-based architecture

Opting for a service-based or microservice based architecture in the backend was the idea that appeared more attractive to the team. As explained previously, it provides a lot of benefits in terms of development, CI/CD, while allowing us to apply other software architecture patterns such as hexagonal architecture.
Even though all backends have been written in Java, which is the predominant language known by the team members, each service could have been written in a different language. Such features are helpful in an environment and team that aspires for a scalable, easy to maintain, and robust ecosystem.

The microservices are as follows:

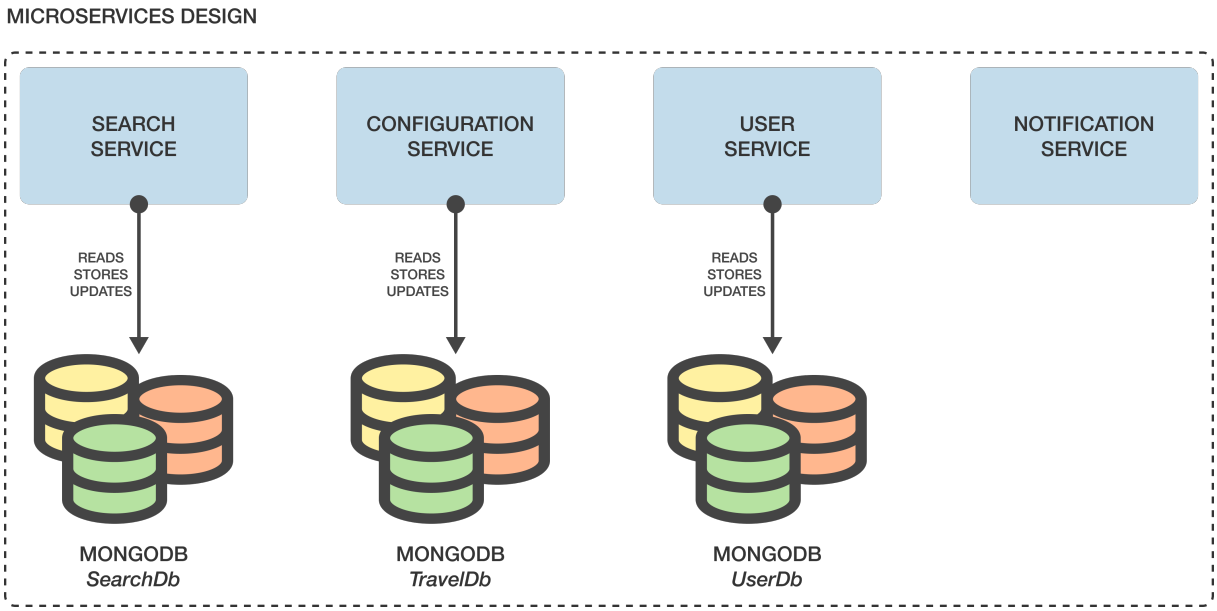

Figure 5.2: Initial backend microservice structure

Without getting into much detail about the implementation of each backend, each has its own database, all working within the same MongoDb cluster. It is common to use a different database for each service when working with microservices.
The function of each service is:

- **Search service**. Responsible for executing the asynchronous job of searching for results of each travel alert.
- **Configuration service**. Responsible for managing the travels that a user can create.
- **User service**. Responsible for managing the user data. It provides the endpoints for registering, authentication and token refreshing.
- **Notification service**. Responsible for notifying the user through email or WhatsApp with the new travel alerts found.

The goal of the API gateway is to hide this structure from the backend. Therefore, if the gateway is added to the previous diagram, it is obtained:

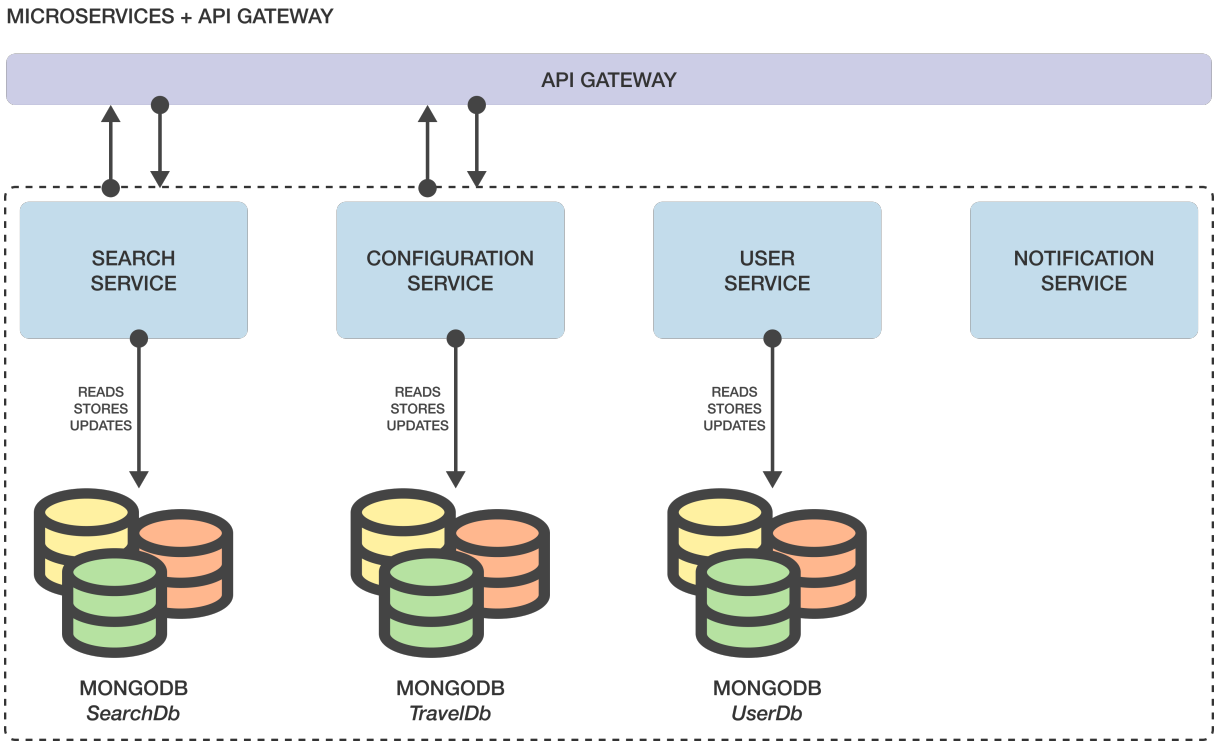

Figure 5.3: Microservices connected with the gateway

As it can be seen in the figure above, the gateway only will send requests to the user and configuration services. That is because the search service is made to work internally and share the results to the other services.
The gateway will forward authentication calls to the user service, i.e. requests regarding registering and authenticating, or a web-service to refresh the authentication token of a user. It will send the configuration of the travels to the configuration service. This service will store all the travel alerts for the user, as well as notifying the search service to start performing the required search.

5.1.3 Event driven architecture

When the configuration service requests for a list of results to the search service, it could keep a connection between the two applications. However, this behaviour is not performant at all, as the more requests there are, the more blocked the applications will be. This is translated to an extremely poor performance from our backend, as well as an extensive amount of resource usage.
Therefore, the solution is to establish asynchronous communication between the different backend applications. This approach enables a backend to send a message to another backend without actively waiting for the response. Once the response is received, the backend can take appropriate action. Meanwhile, the sending backend can continue executing other tasks concurrently.
By adopting an asynchronous communication model, backends can achieve better efficiency and utilize their resources more effectively. Instead of blocking and waiting for responses, backends can offload tasks to other components and proceed with additional operations. This asynchronous nature enables backends to maximize their throughput and responsiveness, leading to improved overall system performance.
Moreover, asynchronous communication allows for greater flexibility and fault tolerance. If a backend receives a high volume of requests or experiences delays in processing, the asynchronous approach ensures that other backends can continue functioning independently, pre-

venting bottlenecks and maintaining system stability.

In order to achieve this asynchronous communication, the team has opted to use a RabbitMq message broker for the backends. By adding the new broker, the schema is updated as following:

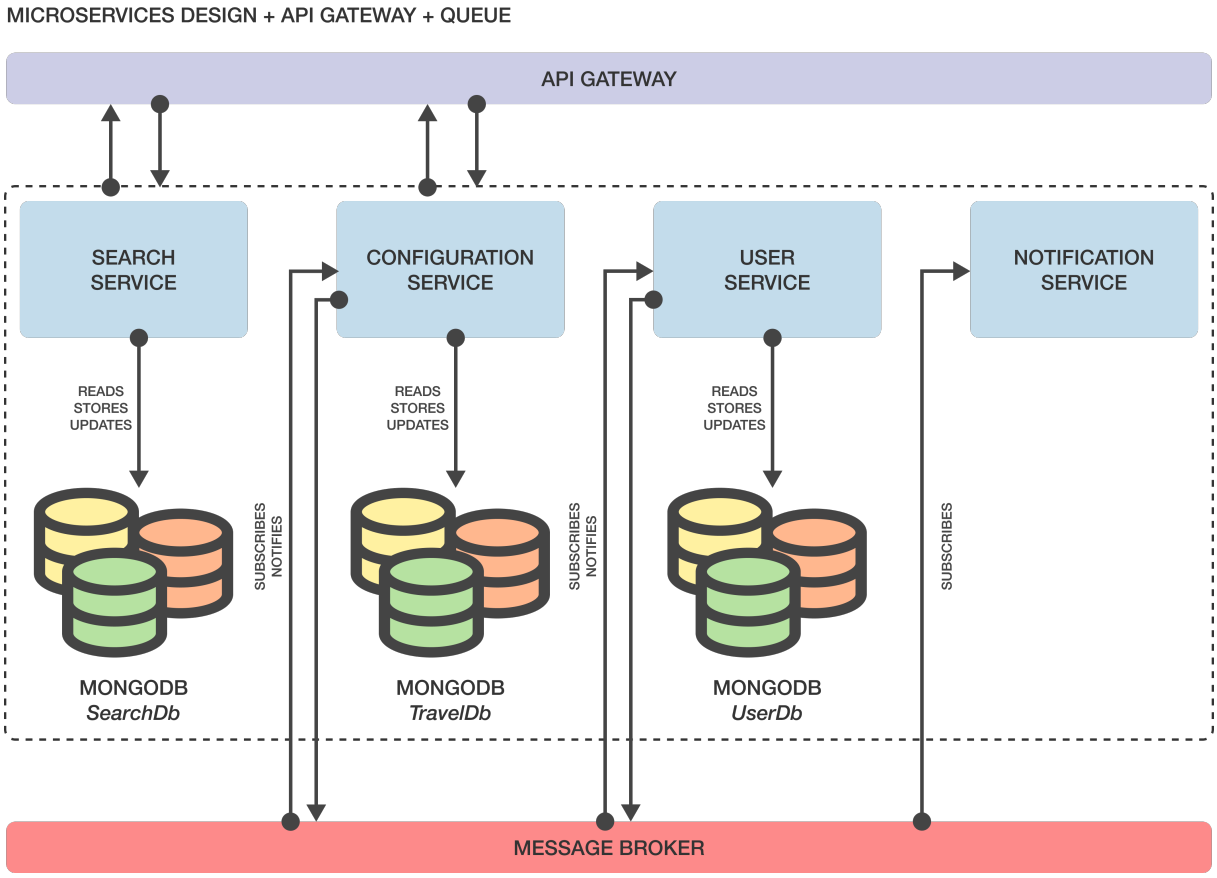

Figure 5.4: Final architecture design

5.2 Frontend structure

The frontend structure has already been introduced in previous sections, however, the goal of this section is to provide a deep dive into the decisions that have led to the chosen structure. Furthermore, as this thesis is based in the frontend application development, it is more explanatory than the backend structure section.

5.2.1 Initial challenges

Before defining how would the repository or repositories of the frontend would be, a first challenge arose regarding the usage of hexagonal architecture.
While the hexagonal architecture can be applied to various types of applications, including frontend applications, it is not commonly used in modern frameworks like React and Vue[1]. There are a few reasons for this:

- **Simplicity and ease of use**: Frameworks like React and Vue are designed to provide a simplified and intuitive approach to building user interfaces. They prioritize ease of use

[1]React and Vue have been used as the example as they are the two of the most popular frontend frameworks nowadays. It is important to note that most of the challenges explained in this section could apply to any component-based frontend framework.

and developer productivity by providing a clear and straightforward programming model. The hexagonal architecture, on the other hand, introduces additional complexity and layers of abstraction, which may not be necessary for the majority of frontend applications.

- **Focused on UI concerns**: Modern component-based frameworks are primarily focused on managing the user interface and handling the view layer of an application. They provide features and abstractions that are specifically tailored for building interactive and dynamic user interfaces. The hexagonal architecture, on the other hand, is more concerned with the separation of concerns and the decoupling of business logic from infrastructure, which may not be the primary focus of frontend frameworks.
- **Component-based architecture**: React and Vue are based on a component-based architecture, where the user interface is broken down into reusable and composable components. This approach aligns well with the principles of modularity and reusability but does not necessarily require the level of decoupling provided by the hexagonal architecture. The focus in frontend frameworks is often on managing the relationships and interactions between components rather than on separating business logic from infrastructure.
- **Limited backend integration**: Frontend frameworks typically interact with backend APIs or services to fetch data and perform actions. The hexagonal architecture is more commonly applied in scenarios where there is a need to integrate with various external systems and dependencies, such as databases, third-party APIs, or messaging systems. In frontend applications, the emphasis is usually on consuming APIs provided by the backend rather than managing complex integration scenarios.

While the hexagonal architecture may not be commonly used in component-based frontend frameworks, like React and Vue, it can still be beneficial in certain cases, especially when building large and complex frontend applications that require a high level of modularity, testability, and extensibility. However, it's important to consider the trade-offs in terms of complexity and developer experience before adopting this architecture in a frontend context.

Taking this into account, two options software design ideas were over the table:

1. Finding a code structure that would allow the developers to apply most of the hexagonal architecture patterns.
2. Not applying hexagonal architecture patterns and sticking to having business logic within the frontend.

Obviously each option has advantages and disadvantages. Therefore, in order to decide which option was more suitable or preferred for the current case, I answered the following questions, for each possibility:

1. How easy to extend the domain for the option is?
    - **HA**[2]: Straightforward, as it is one of the main ideas of such architecture.
    - **NHA**[3]: Can be on both edges. If domain is fairly small and modular, it can easily be extended. On the other hand, it can be a nightmare for developers incrementing the time for changes and increasing the possibility of side effects.
2. How easy would it be to share the domain with another frontend application?

[2]Stands for *Hexagonal Architecture*
[3]Stands for *Non-Hexagonal Architecture*

- **HA**: Straightforward, as the domain is decoupled from the framework and can be shared with other frontend applications easily, without the need of any modification. The domain of the frontend must be frontend framework-agnostic.
- **NHA**: Difficult, as the business logic is tightly coupled with the frontend framework, making it challenging to extract and share the domain with other applications. Moreover, each framework may have different methods to separate business logic from components, making these methods non-compatible between each framework.

3. How attached is the domain to the framework?

   - **HA**: Loosely coupled, as the domain is decoupled from the framework, allowing for easy substitution or changes to the frontend framework.
   - **NHA**: Tightly coupled, as the business logic is entangled with the frontend framework, making it difficult to switch or adapt to a different framework.

4. What is the quality of the *frontend* developer experience?

   - **HA**: Generally positive, as the hexagonal architecture promotes clean separation and modularity, which can enhance the developer experience in understanding and maintaining the codebase.
   - **NHA**: Variable, as the lack of clear separation between business logic and frontend concerns can lead to a steeper learning curve and potential challenges in maintaining and evolving the application.

5. How likely is it to increase the frontend bundle size, therefore, *decreasing user experience*?

   - **HA**: Less likely, as the hexagonal architecture promotes modularity and the separation of concerns, which can lead to more efficient code and better control over the bundle size. Nonetheless, it requires a strict analysis to the code and bundle size, in order to split as much code as possible, ensuring the minimal bundle size. Furthermore, utilities such as lazy loading and server-side rendering can also have a huge impact on user experience.
   - **NHA**: More likely, as the business logic being tightly coupled with the frontend framework may result in larger bundle sizes, potentially affecting the user experience. However, lazy loading and server-side rendering can reduce the amount of JS sent to the client, thus reducing the latency for the page to load. Furthermore, configuration would be less complex as in the **HA** approach, were multiple bundles would have to be configured, as well as proper configuration to serve such bundles.

6. In case of incorporating new teammates into the codebase, how likely is it for new developers to understand and quickly start working with it?

   - **HA**: More likely, as the hexagonal architecture provides a clear separation of concerns, making it easier for new developers to understand and contribute to the codebase.
   - **NHA**: Less likely, as the lack of separation between business logic and frontend concerns may require new developers to spend more time grasping the codebase and its intricacies.

7. In the case of an application that requires to be developed as soon as possible, how time-consuming is this methodology?

- **HA**: The hexagonal architecture, with its focus on separation of concerns and modular design, may require additional upfront planning and design efforts. Even more in an environment that is not thought to support or simplify the application of such patterns or methodologies. Implementing the necessary layers and abstractions can take more time compared to a more straightforward approach. However, it can provide benefits in the *long term*, such as easier maintenance and extensibility.
- **NHA**: Not applying the hexagonal architecture patterns and having business logic within the frontend can allow for faster development initially, as there are fewer layers and abstractions to set up. However, it may lead to challenges in the long run, such as increased code complexity and difficulties in maintaining and extending the application.

Taking into account the previous answers, the hexagonal architecture has been chosen despite the potential challenges and complexities that may occur when defining and designing the codebase architecture. The decision to prioritize HA is based on the following reasons:

1. **Extensibility**: The hexagonal architecture approach provides a structured and modular approach to software design, enabling easier extension of the domain and incorporation of new features or changes in the future. By separating the core business logic from external dependencies and infrastructure concerns, the application becomes more flexible and adaptable to evolving requirements.

2. **Code reusability and sharing**: The hexagonal architecture allows the domain logic to be decoupled from the frontend framework, facilitating sharing of the domain with other frontend applications or even backend systems. This promotes code reuse, improved collaboration between teams, and faster development in the long run.

3. **Maintainability and Testability**: The clear separation of concerns in hexagonal architecture enhances maintainability and testability. With a well-defined architecture, it becomes easier to write unit tests for the core business logic and make changes without affecting other parts of the application. This improves code quality, reduces bugs, and enhances overall software stability.

4. **Developer Experience**: Although implementing the hexagonal architecture may require more initial effort, it promotes clean code organization, modularity, and a better understanding of system boundaries. This contributes to a more pleasant and productive developer experience, particularly for larger and more complex projects.

5. **Future-proofing**: By adopting the hexagonal architecture, the application becomes less tightly coupled to the frontend framework, reducing the risk of vendor lock-in and providing more flexibility to adapt to future changes, such as switching to a different framework or adopting new technologies.

The decision to choose hexagonal architecture acknowledges the challenges involved but prioritizes long-term benefits such as extensibility, code reusability, maintainability, and improved developer experience. It aims to create a robust and adaptable software architecture that can evolve with the changing needs of the application and provide a solid foundation for future development.

5.2.2 Architecture solution

Being someone who has never had professional experience on working in projects that use hexagonal architecture or a variant of it, it was quite a challenge to define a software architecture that would work with the hexagonal architecture. Some aspects about the architecture were clear:

1. There should be a package or a module that will include the domain of the enterprise. This domain **must** be independent of the UI logic, therefore, frontend framework independent.

2. The frontend implementation should use the infrastructure layer of the domain module, which will abstract the business logic, therefore completely removing domain business logic from the frontend.

It is important to differentiate between the domain business logic, which should not have any relation with the frontend framework, from the user interface (or view) business logic, which may be state management and similar, necessary to ensure the application works, while being independent of the domain.

In addition, it is essential to have a design system or component library that is attached to the frontend framework but agnostic to the specific domain. This entails the existence of at least one package per framework used in any frontend application, providing a collection of base components (such as input fields, dialogues, and buttons) that can be utilized across different apps. For instance, if there is a React application, it should have a React component library that is designed to be extendable and modular, allowing for reusability throughout the entire enterprise. This component library serves as a centralized repository for commonly used UI components, ensuring consistency in design and user experience across different applications built with the same frontend framework.
Even though these libraries will not follow a hexagonal architecture from top to bottom, they will take advantage of some of its principles.

Finally, each frontend application will combine the shared domain, the shared components, and its specific framework implementation.

5.2.3 Domain architecture

The domain has to be understood as a package, even though it uses the same naming as the *domain* from the hexagonal architecture principles. The domain package it is separated in three layers: domain, application, and infrastructure. Each layer corresponds to each layer in the hexagonal architecture and is going to be developed accordingly. However, the infrastructure layer is slightly different from it commonly is. Normally, the last layer of the HA[4] is the infrastructure, yet in this specific design, it will only be the last layer of the domain package. This last layer is going to be used in the frontend, which is the last layer of the architecture, as the frontend will always be above the architecture.

[4] Hexagonal architecture, for short

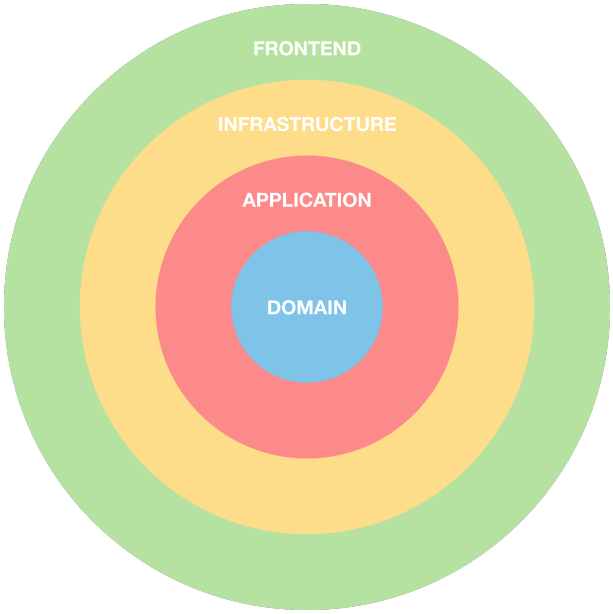

Figure 5.5: Four layer hexagonal architecture

As explained previously, the domain must be frontend framework independent, therefore it should not know anything about what is in the outer layer. This respects to the hexagonal architecture principles and allow us to have a domain package that can be shared by many frontends, independent of their technology.

5.2.3.1 Infrastructure layer

The application layer has little to add, since it does not differ an implementation that can be found in other projects that use hexagonal architecture.

Delving deeper into the structure of the domain package, the domain and application layers have been implemented following the principles of the hexagonal architecture (HA) from top to bottom. However, the infrastructure layer has been slightly modified to better cater to potential frontend requirements. For example, using the controller pattern wouldn't be suitable as the package is not intended to be a REST API. Instead, it should provide entry points for the frontends to utilize the domain logic, such as data validation or retrieval.
The purpose of these entry points is to decouple the fetching function, which can be implemented using either the native 'fetch' function or a dedicated package like `axios` from the frontend. The infrastructure layer of the domain package will simply provide a function that utilizes an HTTP client to communicate with an external server. The frontend can then make use of this entry point without needing to be aware of the underlying implementation details.
This approach is crucial to minimize the domain-related business logic within the frontend. It offers several benefits, including:

1. **Reduced Dependencies**. By providing entry points that abstract away the specifics of data fetching and communication, the frontend application becomes less dependent on external libraries or frameworks. This results in a cleaner and more lightweight frontend codebase.

2. **Simplified Testing**. With a decoupled infrastructure layer, testing becomes easier as the infrastructure can be easily mocked or replaced with test-specific implementations. This allows for more focused and efficient testing of the frontend logic without the need for complex setups or external dependencies.

3. **Abstraction and Modularity**. The use of entry points abstracts away the implementation details of data retrieval, enabling the frontend to focus on using the domain logic without being concerned about the specific technologies or protocols involved. This abstraction enhances modularity and allows for easier future changes or updates to the infrastructure layer.

By decoupling the frontend from domain-specific infrastructure concerns, the application benefits from reduced dependencies, simplified testing, and improved modularity. It ensures that the frontend remains focused on its primary responsibilities while leveraging the domain logic through well-defined entry points.

These entrypoints are exposed, as a package, named as queries. A query is a function, as explained above, that can either be a validator (i.e. a validator for an input field) or an HTTP client call (i.e. a function that fetches all the travels from a user).
To implement the contract of the domain, which is defined as an interface, the queries need to adhere to this contract. Although queries are pure functions, the interface must be implemented as a class. These classes, commonly referred to as *queriers* serve as the implementation of the domain contract.
The queriers encapsulate the logic and operations specified by the domain contract. They act as a bridge between the application layer and the queries, ensuring that the queries align with the defined domain contract. By implementing the domain contract within queriers, consistency and clarity in code implementation are promoted.
Queriers are designed as singletons, meaning that only a single instance of each querier class is created and shared across the application. This singleton approach provides memory efficiency, as there is no need to create multiple instances of the same querier. It also simplifies the usage of queriers, as the same instance can be reused in different use cases, promoting code reuse and reducing duplication. Using queriers brings several benefits to the application:

1. **Consistency**. By implementing the domain contract within queriers, all queries adhere to the same interface. This promotes consistency in how queries are structured and ensures that they fulfill the expected behaviour specified by the domain contract. This is one of the core principles of HA.

2. **Code reusability**. Queriers can be reused across different use cases, allowing for efficient and modular development. The same querier instance can be utilized in multiple contexts, reducing code duplication and enhancing maintainability.

3. **Memory efficiency**. Queriers as singletons optimize memory usage. With only one instance of each querier, unnecessary memory allocation is avoided, leading to better performance and resource management.

Since DDD (domain-driven design) is used, this implementation allows the developers to separate *queries* and *queriers* by domain, or even bounded contexts[5]. The implementation of multiple domains looks as:

[5]Since the application does not require the usage of bounded contexts, the packages have been separated by domain (i.e. authentication or travels) in a single bounded context

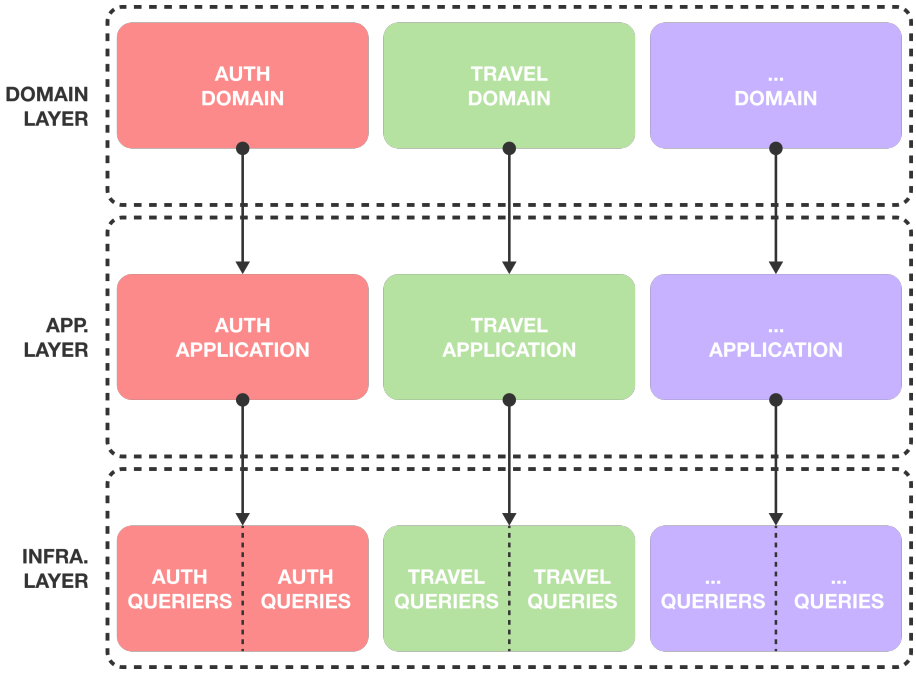

Figure 5.6: Exemplification of the implementation of multiple domains in the proposed architecture.

It is important to mention that, ideally, domains *should not interact between each other*.

5.2.4 Shared components

Once the domain architecture has been defined, the next step was to create an internal package, within the monorepository, that would contain all the components. Since the goal of this project is to also develop a scalable and extendable architecture, it was decided to create a new UI module that would have the implementation of the components from the ”design system” for each framework. For example, in this case, the UI package only contains a subpackage, named *react*, that will contain the implementation of the components in order to be used in react applications.
Moreover, this UI package can easily add another framework-dependent library, as each library will be compiled and used independently. In the long term, the team could have a unique design system, implemented in different languages or frameworks.

These components should be domain independent, therefore they should as much extendable as possible. With the appearance of React, component-based frameworks have adopted the principle of *composition over inheritance*. This principle encourages building components by composing smaller, reusable components instead of relying on inheritance. By following this principle, components become more flexible, maintainable, providing easier extension and customization.
Looking for domain-independent components, composition over inheritance is especially beneficial, as it enables the creation of components that can be easily extended and combined to build complex user interfaces. Rather than relying on a deep hierarchy of inheritance, components can be composed together, utilizing their individual functionalities to create new, specialized components.
Furthermore, by utilizing composition, developers can leverage the power of higher-order components[6] (HOCs), render props, or hooks to enhance and extend the behaviour of existing

[6]It is also important to mention that with the usage of React hooks, most frameworks are switching to a hook-based development, that reduces the usage of HOCs

components. This approach allows for the selective reuse of specific features or functionality, which also implies code duplication reduction and increased code maintainability.
Overall, by embracing the principle of composition over inheritance, domain-independent components become more versatile, extensible, and modular. This approach aligns with the component nature of frameworks like React and Vue, enabling developers to build flexible and reusable UI components that can adapt to various application requirements.

Taking into account our current architecture, by adding the component library:

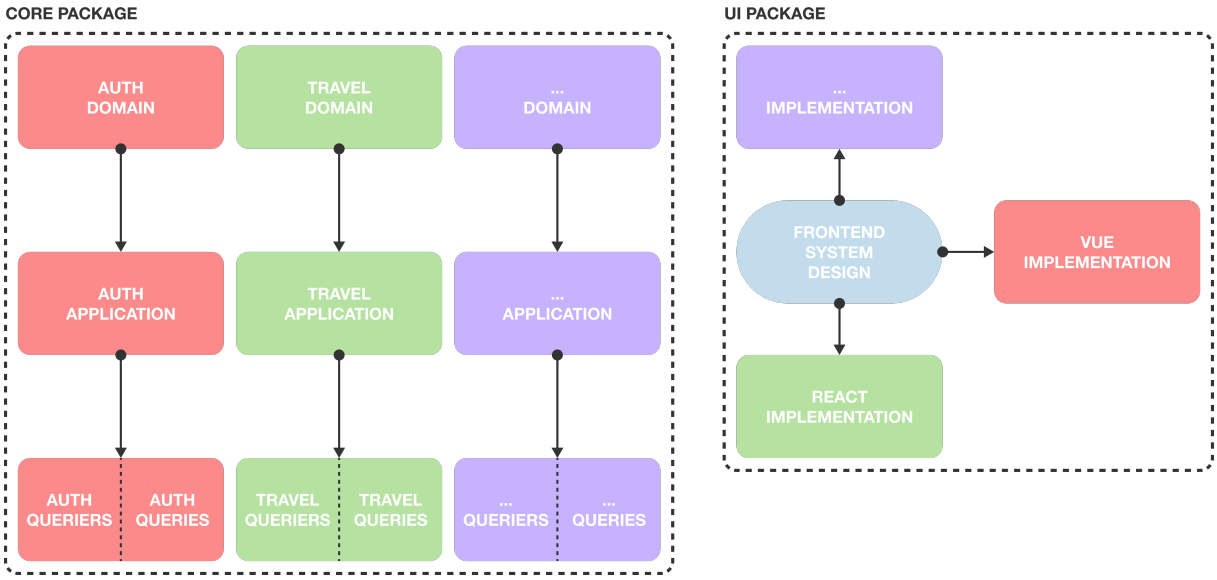

Figure 5.7: Combination of core and UI packages.

It is important to mention that the *System Design* pill should not be an implementation, rather a specification, defined and maintained by the UI/UX team. Since there was not enough time to develop a system design, this box does not exist in the project current status.

5.2.5 Site application

Last but not least is the Nx application, which serves as the front-facing layer of our architecture and contains the Next.js application that the users will interact with. This layer is tightly coupled with the framework and leverages the functionality provided by the core and UI layers to deliver a performant and intuitive user experience.

While the Nx application itself is crucial, its source code becomes remarkably streamlined and simplified. This is due to the fact that all the domain logic, including API calls and validation, resides within the core package. Similarly, the base implementation of components is contained within the UI package. With this separation of concerns, the Next.js application can focus primarily on state management and elements that enhance the user experience, such as animations, performance optimizations, resource caching, and other related aspects.
By detaching the domain logic and component implementation to the core a UI layers, the site application will contain a clearer and conciser codebase. This separation improves code maintainability, since modifications in either components or domain logic can be made in the respective layers, without having a direct impact to the frontend. Furthermore, this separation also allows the possibility of different releases per package, as explained in previous chapters.

The implementation of the site package will be explained in the following chapter, as it will focus on such implementation, explaining the decisions taken in order to use all the power of Next.js,

in order to provide a better user experience. To sum up with this chapter, the integration of the site package with the previous packages is:

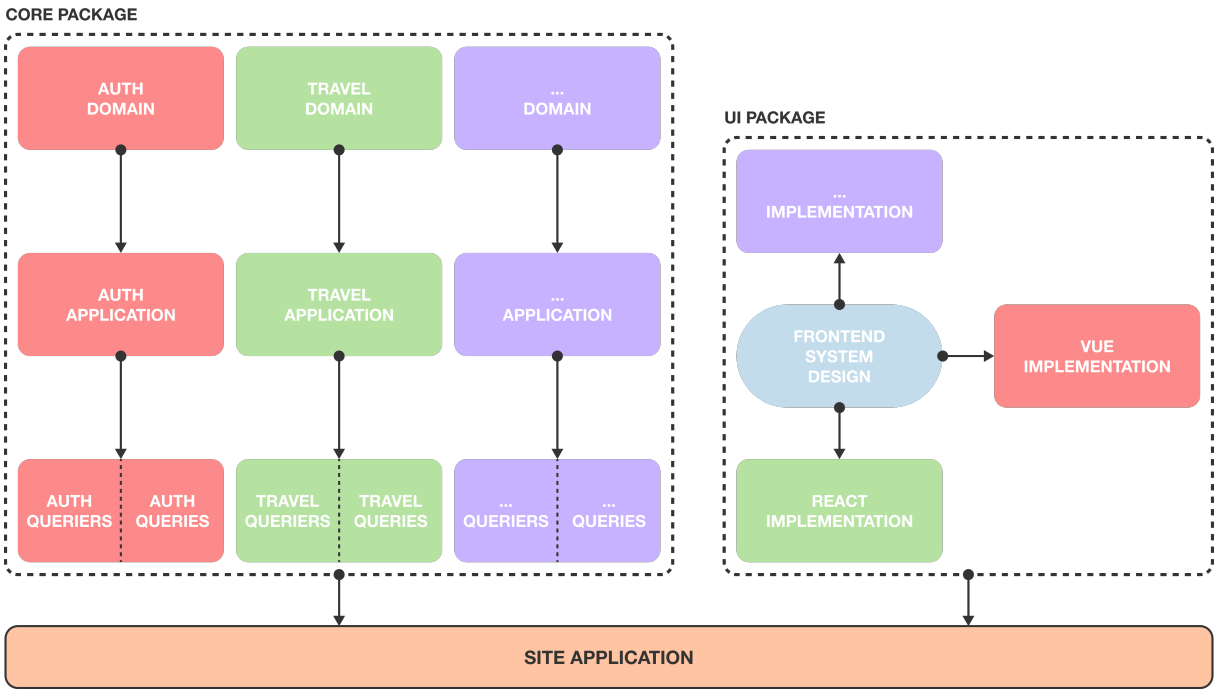

Figure 5.8: Package structure including core, UI and site

6. Implementation

This chapter delves into the implementation of the frontend application, building upon the architectural concepts discussed in the previous chapter. The UI designs, which played a crucial role in determining a portion of the application's structure, are also attached, as they helped compose the application following Next.js principles.

The architectural concepts explained in the previous chapter are brought to life through the practical implementation detailed here. By aligning the implementation with the established architecture principles, a robust, maintainable, and scalable application is ensured.

Moreover, we will explore the Next.js project structure, which provides a framework for organizing components, pages, routing, and other features. Understanding this expected structure is essential for maintaining consistency and facilitating collaboration among developers.

6.1 Designing the UI

As explained in the planning chapter, the first task involved designing the UI. The designs presented hereafter, were the initial implementation taking into account the *initial* objectives and requirements. Moreover, and explained as well in the planning chapter, during the development of the dashboard page, the team decided to change the model of the travel, improving it, and the designs do not incorporate the changes made[1]. Such changes are visualized in the results chapter.

Given the absence of specific design requirements, the design was developed with a set of self-defined requirements in mind. These requirements were: user-friendly interface, reduced complexity to expedite component implementation, an MVP (minimum viable product) approach, and focus on mobile-first design principles.
The primary objective was to create a user-friendly experience and UI that prioritized ease of use and intuitive interactions. The design decisions were influenced by my own experience in designing and working with user interfaces, aiming to deliver a seamless and enjoyable user journey. However, it is worth noting that predicting how users will interact with the application can be challenging and prone to errors. Therefore, it is recommended to incorporate internal tracking mechanisms to gain insights into user preferences and flows. This tracking would enable product owners to better understand which features and workflows users prefer, enhancing the application's usability. However, due to the time constraints of an MVP development, implementing such tracking may not be feasible.
To ensure the development process met project timelines, the design aimed for simplicity and efficiency, while ensuring an attractive user interface. By minimizing the complexity in the design, the implementation of the components became more streamlined and time-effective, enabling the team to focus on core functionality and essential features. Nonetheless, and as explained in the previous chapter, composition over inheritance will allow the developers to update components without generating breaking changes to the current user interface.
In today's mobile-driven world, mobile-first principles are essential. With this in mind, the design prioritized the mobile user experience, ensuring that the application is optimized for mobile devices. By taking into account the unique aspects of mobile devices, such as screen size and touch interactions, the design focused on providing a seamless and intuitive experience for mobile users. Needless to say, it would be more that interesting to perform a tracking analysis on the usage of mobile versus non-mobile users of the application. The results of such

[1] The reason why designs were not updated, was due to lack of time, as being a single frontend I preferred not updating them, rather following the minimum styles and creating components accordingly. Needless to say that, in a corporation, the UI/UX team should update the designs according to the design system of such.

analysis would allow the design team to improve the application for the specific platform, as well as knowing which platforms are more important than others.

Finally, by adhering to these self-defined requirements, the design aimed to strike a balance between user satisfaction, development efficiency, and the delivery of a viable product. The section will delve deeper into the design decisions made, explaining how these requirements influenced the design choices and ultimately contributed to its creation.

6.1.1 Authentication pages

One of the requirements of the application is the possibility of authenticating the user, in order to persist data accordingly. Allowing non-authenticated users is not part of the MVP, but could be considered as a future feature.

The login and sign-up pages are very basic, simply requesting for the information mandatory for each process to authenticate

- **Login**. Requirements are user email and their password. The password must: include a lower and a capital letter, a number, a special symbol and be at least 8 characters long. These requirements have now become a common standard, and ensure protection over the user's account.
- **Sign-up**. Requirements are the same as login, with the addition of the first and last name, and a phone.

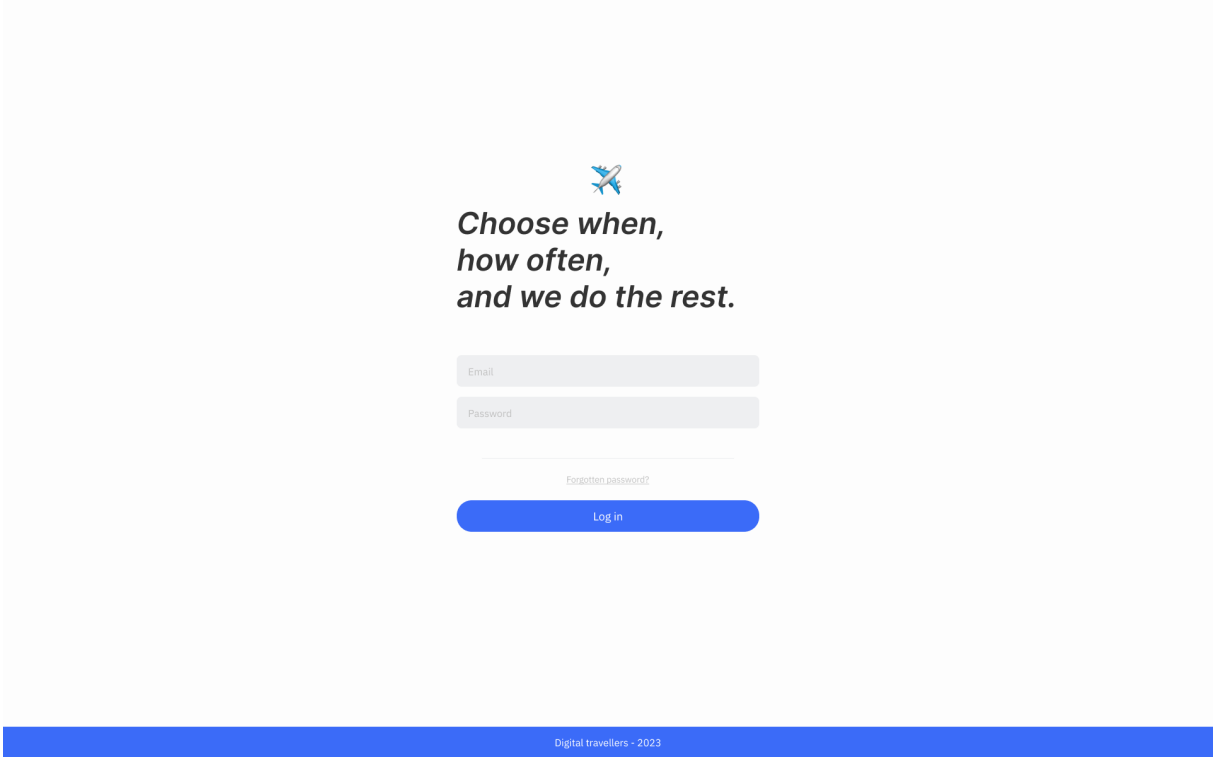

Figure 6.1: Login page design

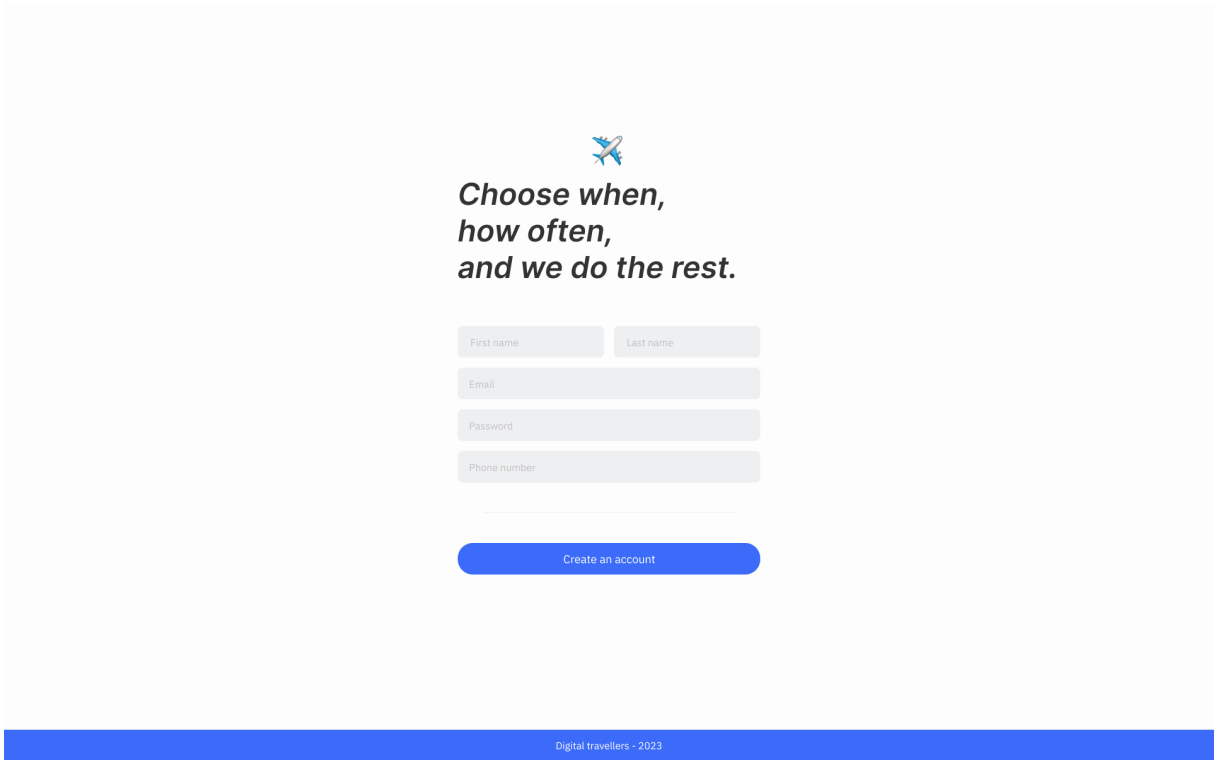

Figure 6.2: Sign-up page design

6.1.2 Authenticated pages

Once the user has been authenticated, it will be redirected to the dashboard page. The dashboard page is the page from which the user is going to be able to see the list of travels created. As explained, the designs do not reflect the correct model, as the travel's model was updated during the development of the application.
It should also allow the user to interact with the created alerts, being able to modify them or remove them, as well as being able to create new ones.

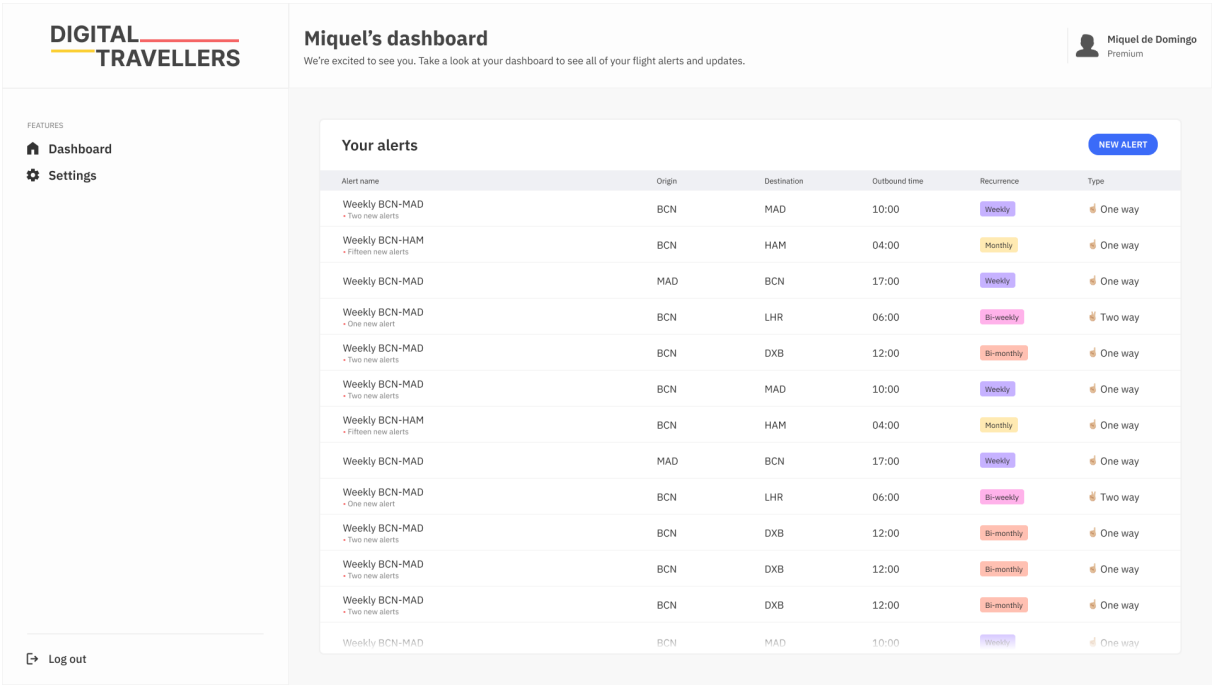

Figure 6.3: Dashboard page design

The other authenticated page, which would only be developed if there was enough time, is the settings page. This page allows the user to update their information.

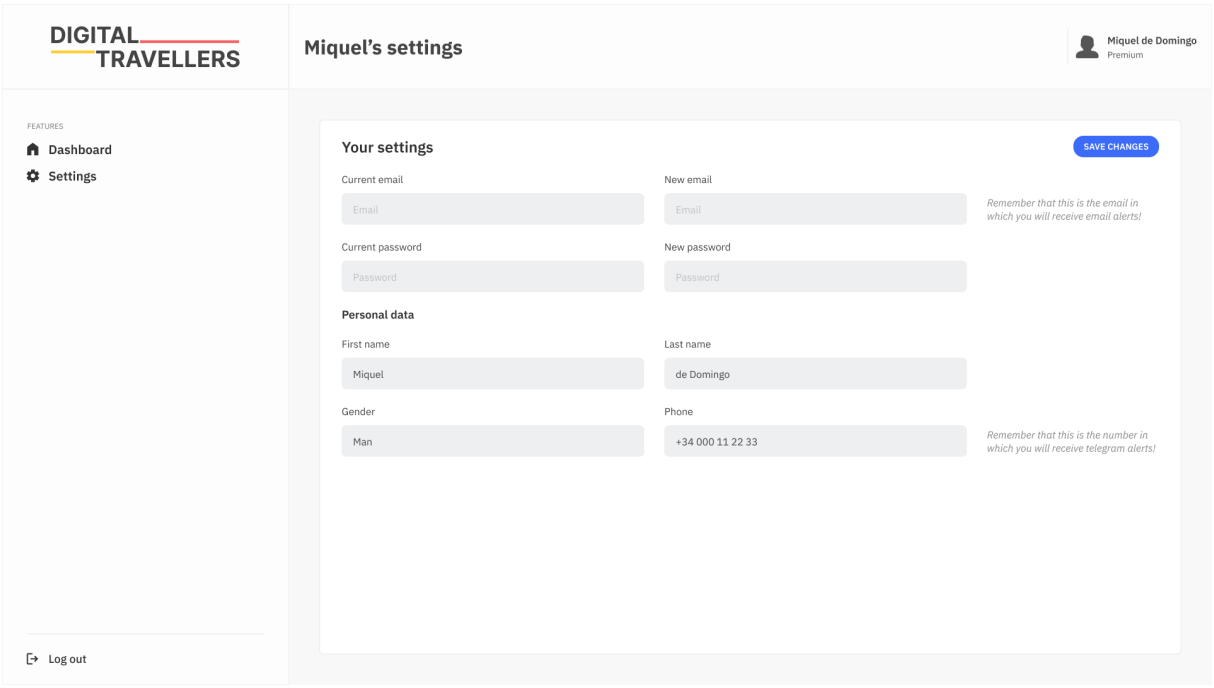

Figure 6.4: Settings page design

6.1.3 Mobile first

Last but not least, the mobile designs were a crucial aspect of the overall design strategy. The objective was to simplify the mobile design to minimize absolute differences between the desktop and mobile versions. The rationale behind this approach was to reduce complexity in the development process. As fronted developers, we often encounter intricate designs that result in a complex, challenging and hard to maintain codebase. With the MVP approach in mind, the focus was creating a design that promotes component and layout reusability in the frontend.

By aligning the mobile designs with the overall design principles, the goal was to achieve consistency and coherence across the different screen sizes and devices.

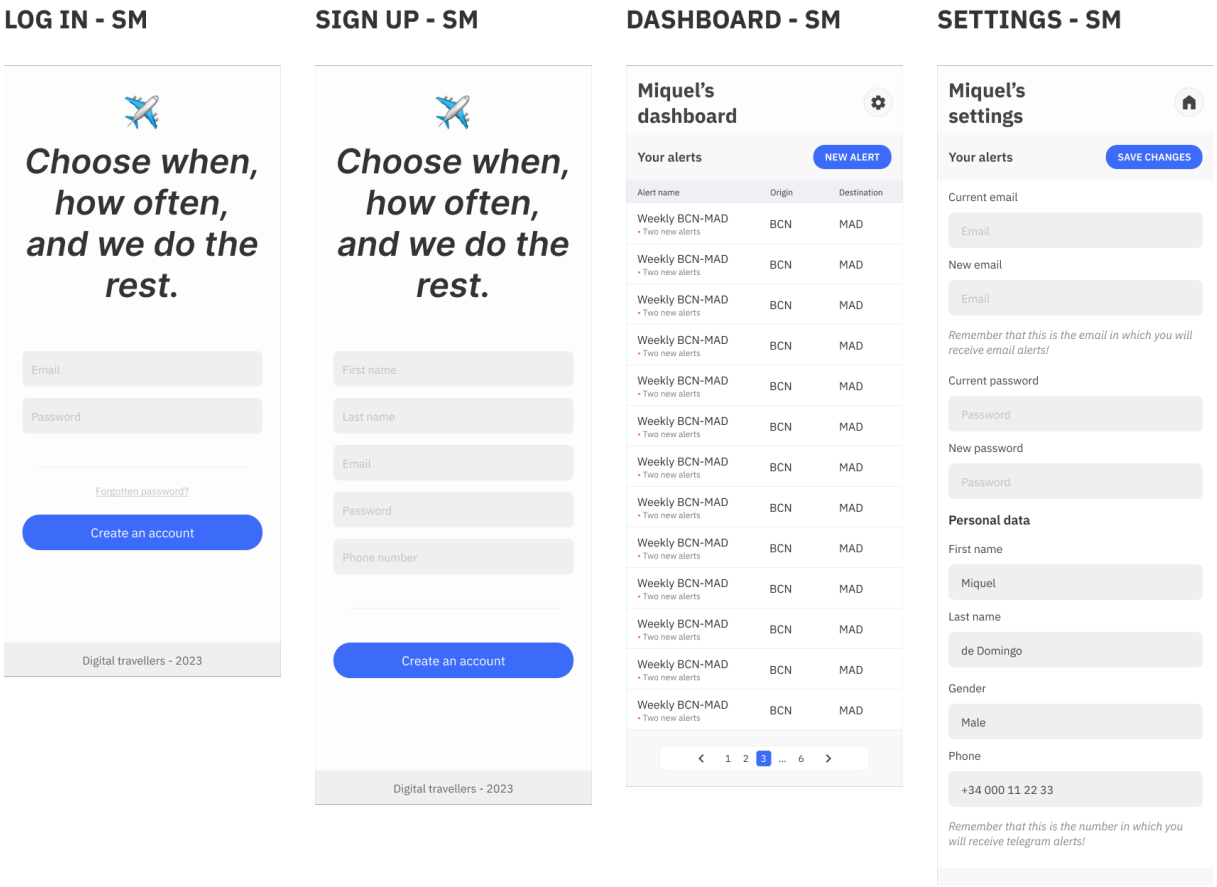

Figure 6.5: Mobile designs for the 4 initial pages of the application

6.2 Domain development

With the designs in mind, the frontend was ready to start being developed. In the development process, it has been observed that a significant portion of new tickets or tasks revolve around adding new features or making updates within the domain package, as a starting point. This package serves as the heart of the application, encapsulating the core business logic. As a result, the first phase of addressing a ticket often involves making changes within the domain package.

As it has been repeated throughout the project, the goal is to ensure a robust and maintainable codebase. Therefore, the development of the domain package strictly adhered to the principles of the hexagonal architecture. Accordingly, the development has followed a sequential

order, beginning with the domain layer, followed by the application layer, and concluding with the infrastructure layer.
Moreover, initial tasks required more time and decision-making, as the codebase had nothing written yet.

To ensure the reliability and functionality of each layer, comprehensive testing has been added. Tests have been developed per layer, allowing for thorough validation of the domain, application, and infrastructure components. Within the domain layer, particular emphasis has been placed on testing entities with complex business logic, as these entities play a central role in driving the application's functionality. Simpler entities or those closely intertwined with other layers are covered by more comprehensive and complex tests, ensuring their validation within a broader context.
It is also important to distinguish between the tests developed for each layer:

- **Unit testing**. Unit testing allows the developers to test a single unit of code, excluding all the other that depends on or are attached to it. Therefore, the domain and application layer have been tested with unit tests.

- **Integration testing**. Integration testing allows the developers to test broader units of code, in this case, integration testing has been used for each querier in the domain layer. Therefore, the tests ensure the behaviour of the code, as a whole.

End to end or acceptance test have not been developed because the domain package does not require of such testing. Since it is not attached to any framework, database nor similar, there is little benefit in having such tests. Furthermore, the configuration of such tests tends to be really tedious, complex, and time demanding.

To sum up, the development of the domain package has generally been really straightforward, and it has only required the communication between the frontend and the backends to know the expected models to received and to be sent.

6.3 Implementing the frontend

The designs also helped define the small unit of components, that, instead of being in the application package, should be in the UI package. As explained before, the UI package is responsible for having the implementation of the components for each frontend framework it is used within the frontends.

Before diving into the task at hand, the first step was to analyse the design and identify which components needed to be added to the UI React package. It was also checked if there was any existing component that could be repurposed to meet the requirements. However, there were cases where the existing components did not fully meet the needs of the task. In such situations, it was necessary to determine whether the desired functionality was something generic, applicable to future tasks, or if it was specific to the current task.
For generic features that we anticipated would be needed in future tasks, they were added to the UI package. This allowed the team to reuse the shared logic and streamline future development. On the other hand, if the feature was specific to the current task, the composition over inheritance principle was followed. This means that necessary features where added to the shared component using composition, without relying on complex inheritance structures. It is also important to not that, if the feature required any type of domain dependency, it could not be added to the UI package, as such has to be domain independent. It indirectly also adds a requirement for all the components that are developed in the UI package, which is that they should be extendable by nature.

6.3.1 Identifying components

The goal of this section is to explain how the process of identifying components has been done. This explanation will be covered for the one of the authentication pages and the dashboard page. As explained previously, this is the first step in order to define whether a component should be developed and maintained inside the application, or be part of the shared UI package. In this case, most of the components exposed in this section are part of the UI package.

When developing a component for a library in React, there are several considerations that the developer has to keep in mind. All the components should be designed to be domain-independent, as it is also one of the requirements exposed in previous chapters. This promotes the reusability of the components and ensures that the component can play different roles across different projects.
Furthermore, it is crucial to make the component extendable and composable. By designing the component with a modular or atomic[2] approach, it becomes easier to add new functionalities and customizes its behaviour as per specific requirements. This allows developers to build upon the component's foundation without the need to modify its core implementation, leading to more flexibility, adaptability, and speed of development.
Maintainability is another key aspect to consider. The component must be structured in a way that is easy to understand and maintain over time. Clear and concise code, along with proper documentation and consistent naming conventions can greatly contribute to the long-term maintainability of the component.
Additionally, the component should be designed with testing in mind. It should be easy to write unit tests for the component's functionality, ensuring that it performs as expected and minimizing the risk of introducing bugs during development or future updates. By incorporating testability into the component's design, it becomes easier to validate its behaviour and provide a more robust user experience.

In the following sections, we will discuss the implementation of integration testing versus unit testing for components. I have not been particularly inclined towards implementing unit testing for components in the past, mainly because it can be relatively complex due to the numerous dependencies that components often have (such as state and the need for other components). However, when it comes to developing a library, it becomes crucial to test each component individually since they should represent standalone units. Therefore, it makes sense to incorporate unit testing specifically for components, while integration testing may not be as necessary in this context.

6.3.1.1 Sign-up page

The sign-up design is really basic and intuitive. The idea is to have a centred form that requests all the necessary user information to create the account. The following image shows the different components identified, as well as the automatic centring of the form.

[2] Atomic design is a term commonly used in React and component based frameworks. The goal is to design the components in the smaller atoms as possible.

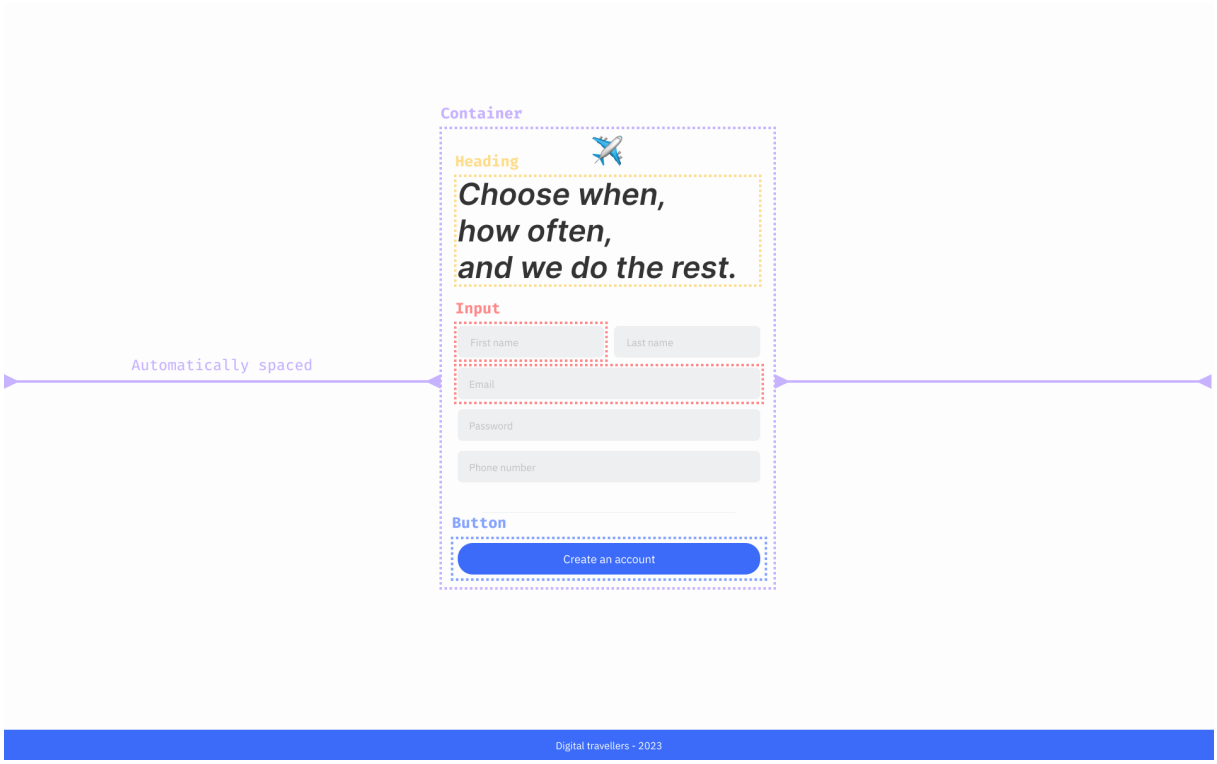

Figure 6.6: Component destructuring of the sign-up page

First aspect that can be seen is the container that wraps the form, ensuring its maximum width and allowing the form to be centred. The same container will be used for the login page. The container is a common component that it is used to wrap content and limit its width. In this case, the container has been defined in order to have different sizes, being able to be set depending on the needs. For instance, the same container can be used in a third page, yet instead of having this *condensed* size, the developers may need to expand it to fit a larger screen. Last but not least, it is the container's responsibility to be automatically centred, independent of its size and the size of its parent.

Inside the container, it is first seen the heading of the form. Modern applications tend to use a great variety of font sizes, each being a different heading, text, subtitles or even captions. HTML contains six types of headers, from h1 to h6. Headers play an important role when talking about semantic HTML. It is important to use proper headers for reasons such as improved SEO or better accessibility (helping screen readers and other accessibility tools). Planning in advance, when developing the typography, I decided to design the entire typography system. The following typography styles have been implemented:

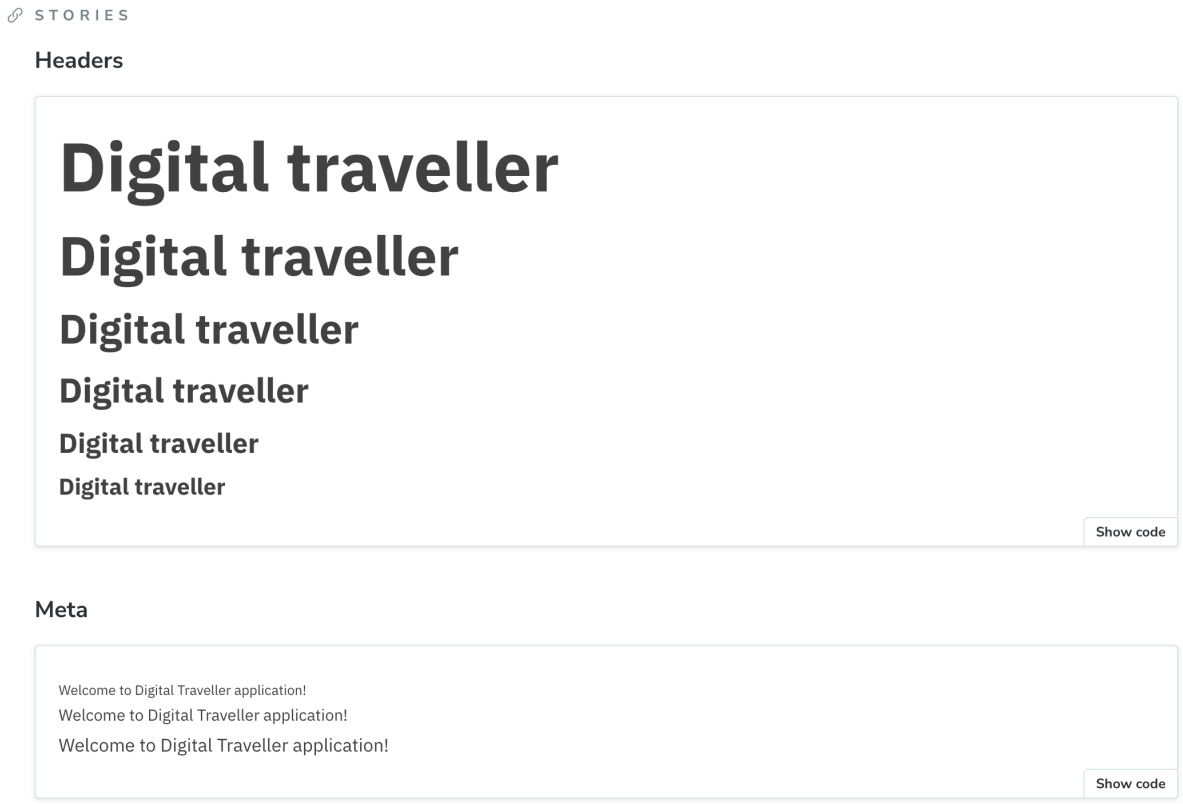

Figure 6.7: System design's typography

The implementation consists of 6 components, one for each HTML heading variant, and 1 component with 3 variants (extra-small, small and medium, which is the default text). The reason why the headings are 6 components instead of one with six variants is because each component internally renders from `h1` to `h6`. Needless to say, these components are extendable and customizable, meaning that they can be adapted to the needs of the design.

Another noteworthy aspect to mention is the library used for visualizing and testing the components, which is Storybook. Storybook provides a dedicated environment for showcasing components in isolation, allowing developers to interact with them and visualize their various states and variations. It serves as an invaluable tool for component development, as it enables rapid prototyping, visual inspection, and iterative design improvements.
With Storybook, developers can create a comprehensive library of reusable components, each with its own story or scenario. These stories represent different use cases or variations of the component, showcasing its behaviour and appearance in different contexts. This approach not only helps in the visual testing of components but also serves as documentation, providing examples and usage guidelines for other developers.
Furthermore, Storybook allows for easy interaction with components through its add-on ecosystem. Add-ons provide additional functionality and features, such as knobs for dynamically tweaking component props, actions for capturing user interactions, and accessibility testing to ensure components are usable for all users.
By leveraging the power of Storybook, developers can streamline the component development process, improve collaboration, and ensure the visual integrity and quality of their components. It aids in building components that are visually appealing, easy to understand, and thoroughly tested, ultimately contributing to the overall success of the library. Additionally, it serves as documentation for the package.

In this case, the image is displaying the story of the typography, which involves the two com-

ponents that are part of the typography.

Next component is the input component, which provides an excellent demonstration of the concepts of composition and extensibility in component development. It also plays a crucial role in user interaction and data input, making it essential to consider various aspects such as accessibility, reusability, extendability and performance[3].
When it comes to accessibility, the input component must adhere to best practices to ensure that it can be used by individuals with disabilities. Even though it may seem out of scope from the MVP, considering accessibility in advance may simplify future work. By prioritizing accessibility, the input component becomes inclusive and usable for a wider range of users, from the beginning.
Extensibility is a crucial aspect for the input, in order to accommodate potential future changes or new requirements. By following a modular design and implementing clear separation of concerns, it becomes easier to extend the functionality without disrupting its existing behaviour. An example would be the addition of features like auto-suggestions, masks, or integrations with external libraries. Such features should be added with minimal impact on the existing code base.
For example, a library will be used in order to ensure the maximum performance of forms, which is named `react-hook-form`. The input component should be able to adapt to this library by default, as it should with any other. Furthermore, it should not be attached to any library. The fact of having such simple components, ensures that its basic behaviour can easily be tested, reducing test complexity and improving readability.

Finally, the button component. As with the input component, the button should be as extendable as possible, while being accessible for all users.

6.3.1.2 Dashboard page

The dashboard page contains more complexity and logic than both of the authentication pages. Nonetheless, in this section only the main content will be focused, as the layout of the page will be explained in the next section.

[3] In React, it is very important to pay attention to the performance of inputs and forms, as bad practises or incorrect patterns may lead to an excess of renders, which will reduce the performance of the application.

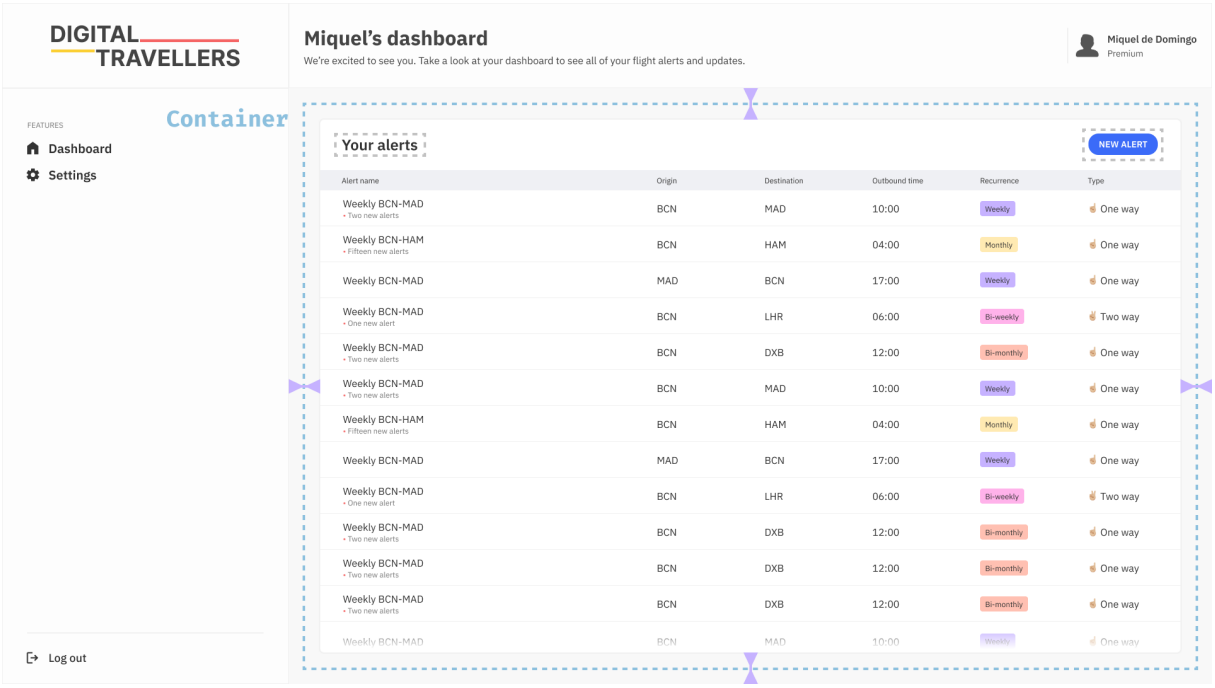

Figure 6.8: Component destructuring of the dashboard content

The content of the page consists of a card that will be centred and proportionally spaced from top, bottom, left and right. This card usage is very common in order to catch the attention of the user to the middle of the dashboard, where the alert list or table is contained.

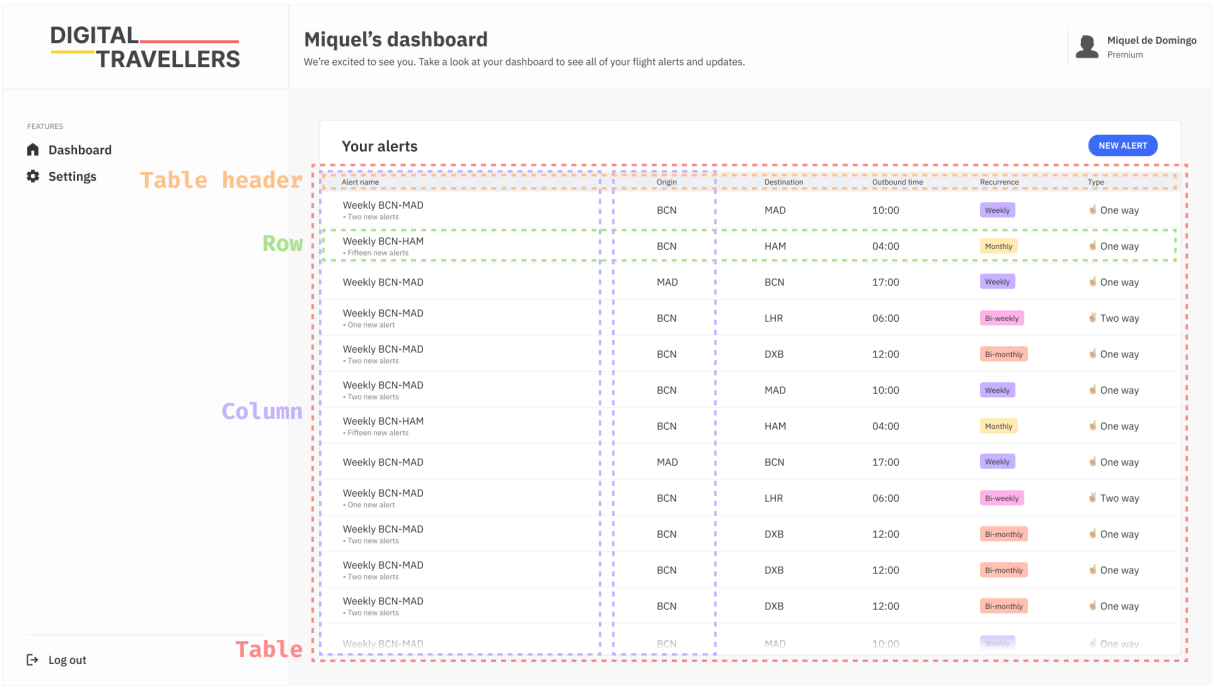

Figure 6.9: Component destructuring of the dashboard table

The table is a more complex component, as it should be responsive and ensure all accessibility requirements, and for this case, I opted to divide the table in the following components[4], some of them being marked in the design, and some not:

[4]This table has been inspired in the implementation of `mui`'s library.

- **Table container** (not marked in the image). The table container it is a container that it is used to provide a certain styling to the table in order, for example, to allow the table header to stick when scrolling down.
- **Table** (marked in the image). Represents the actual table, rendering a `table` element to the DOM.
- **Table head** (marked in the image). Represents the table header, which has a different style than the table body rows. It renders a `thead` element.
- **Table body** (not marked in the image). Represents the table body, where the content should be rendered. It renders a `tbody` element.
- **Table row** (marked in the image). Represents a row of the table and should be used both for the header and the body. It renders a `tr` element as well as providing the alternate style of one row darker than the other.
- **Table cell** (not marked in the image). Finally, the table cell that is the component of the content to display for each position of the table. It renders a `td` element.

The goal of having these many components is for the end developer to be able to construct the table based on its needs. Each component can be extended and customized, and some of them are not even required for the table to work, yet they provide certain utilities that may be necessary. For example, a design may not require of a table header component and such should not represent any inconvenience when not incorporated.
Accessibility should be a top priority. Tables should be designed in a way that allows users of assistive technologies, such as screen readers, to navigate and understand the table structure and content. This includes providing proper table headers, using semantic markup, and ensuring that data cells are associated with their corresponding headers. Additionally, tables should be designed to be responsive, allowing users to interact with them effectively on different devices and screen sizes.
Data representation is another important consideration when working with tables. The table component should provide options for sorting, filtering, and pagination to handle large datasets effectively. These features enhance the usability of the table, allowing users to find and interact with the data more efficiently. Additionally, it is important to consider accessibility features such as keyboard navigation and providing alternative text for table visuals, such as data charts or graphs. However, due to the complexity of some of these features, they have not been included as part of the MVP.

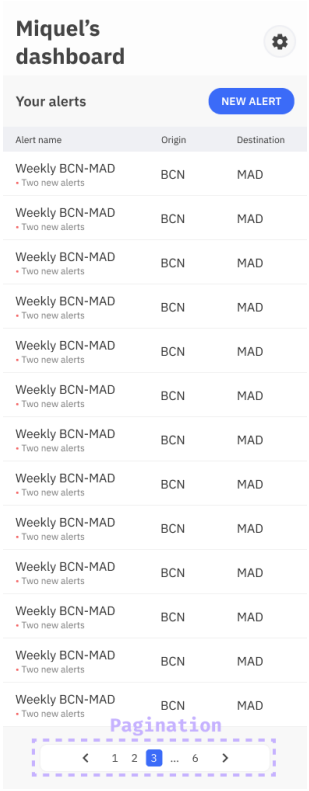

Figure 6.10: Component destructuring of the dashboard content

Finally, the mobile version was initially designed with a pagination component. This component has not been developed as I believe it does not provide the best user experience for mobile users. If the pagination existed, the user would have to scroll to the bottom and then change the page. However, if instead of pagination, infinite scroll is used, it provides a better user experience. Nonetheless, the final implementation has been done by loading all the travels. Implementing infinite scroll or pagination, could be done in a second version, after the MVP.

6.3.2 Layouts for performance

Next.js focuses mostly in server-side rendering and static site generation. One of the challenges developers often face when working with Next.js is creating persistent layouts that do not re-render the entire UI when navigating between pages. This is because, by default, Next.js re-renders the entire UI every time a link is clicked, which can lead to a less-than-optimal user experience.
This section will focus on the analysis of the layouts for the designs, so that they can be reduced throughout the application.

In Next.js, layouts [15] refer to a concept where you can define a common structure or template for your pages. A layout acts as a wrapper component that encapsulates the content of multiple pages, providing consistent styles, navigation, headers, footers, or any other shared elements. It allows you to define a higher-level structure that can be applied to multiple pages in your application.
Layouts are important because they bring several benefits to the development process and user and developer experience:

- **Consistency**: By using layouts, you can ensure a consistent UI across multiple pages. This is especially useful for elements like headers, footers, or sidebars that should remain the same throughout the application. Users will have a unified experience as they

navigate through different pages, having this shared components not disappearing and appearing on a page change.

- **Code reusability**: Layouts promote code reusability by providing a central place to define them. Instead of repeating the same code in every page, you can encapsulate it within a layout and reuse it across multiple pages. This saves development time and reduces code duplication, as well as improving maintenance and simplifying testing.
- **Easier maintenance**: When you have a common layout for multiple pages, making changes or updates becomes easier. The developer only needs to modify the layout component, and the changes will be automatically reflected in all pages that use that layout. This makes maintenance more efficient and reduces the risk of introducing inconsistencies or errors.
- **Flexibility and customization**: Layouts allow for flexibility and customization. You can have different layouts for different sections or types of pages in your application. For example, you might have a different layout for the homepage, an authenticated user dashboard, or a landing page. This gives you the freedom to tailor the layout according to the specific requirements of each page or section.
- **Improved performance**: Layouts can optimize performance by enabling code splitting and lazy loading. With Next.js, you can dynamically load the layout components based on the currently rendered page. This means that the code for layouts that are not needed on a particular page will not be loaded, resulting in faster initial page loads and improved performance.

In summary, using layouts in Next.js brings advantages such as consistency, code reusability, easier maintenance, flexibility, and improved performance. They provide a structured approach to building shared elements and help create a seamless and user-friendly experience across your application.

Taking the above knowledge into account, it was crucial to identify which parts of the designs should be a layout component in order to be shared within the frontend. In our case, and with the 4 different designs, two layouts have been identified:

- **Authentication layout**. The authentication layout is the layout that it is used for the sign in and sign-up pages. It is particularly simple and the changes between mobile and desktop can be contained in the same component. In this case, the only changes are in the footer, which changes colours. The changes in the form grid are not responsibility of the layout.
- **Authenticated layout**. The authenticated layout is the layout that is used for the pages after the authentication process. It displays user information as well as navigation. In this case, the logic between the desktop version and the mobile version is quite different, therefore a facade pattern has been used in order to know which version to display.

The following diagram illustrates the usage of the facade pattern:

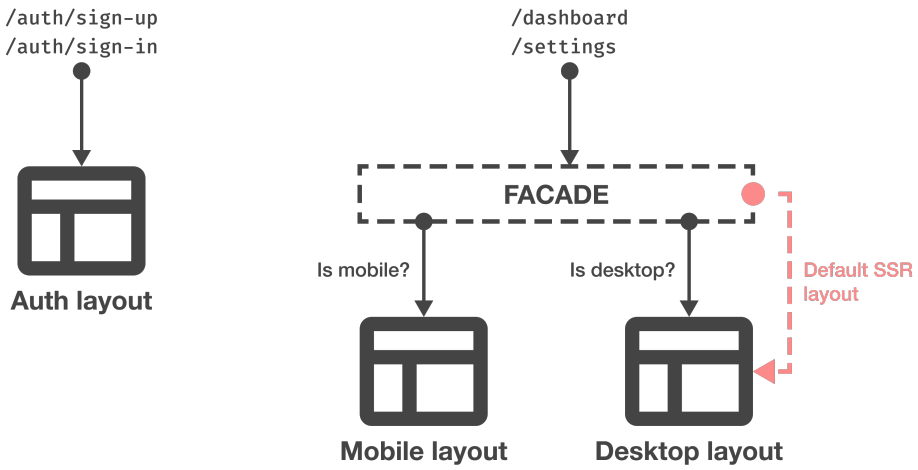

Figure 6.11: Component facade diagram

When the user enters one of the authentication URLs, which are `/auth/sign-up` or `/auth/sign-in`, the authentication layout will render the mobile or desktop version, depending on the screen viewport detected.
On the other hand, when the user enters one of the authenticated pages, since the layout has to change drastically, there are some logic aspects that need to be taken into account. Therefore, in order to avoid having to copy and paste this logic to every authenticated page, such logic it is kept in a single component, that will render one layout or the other. Moreover, new added pages will be able to use the same layout without the need of adding any additional logic.

6.3.2.1 Authentication layout

Starting with the sign-up layout page, it is divided in two different parts: the content and the footer.

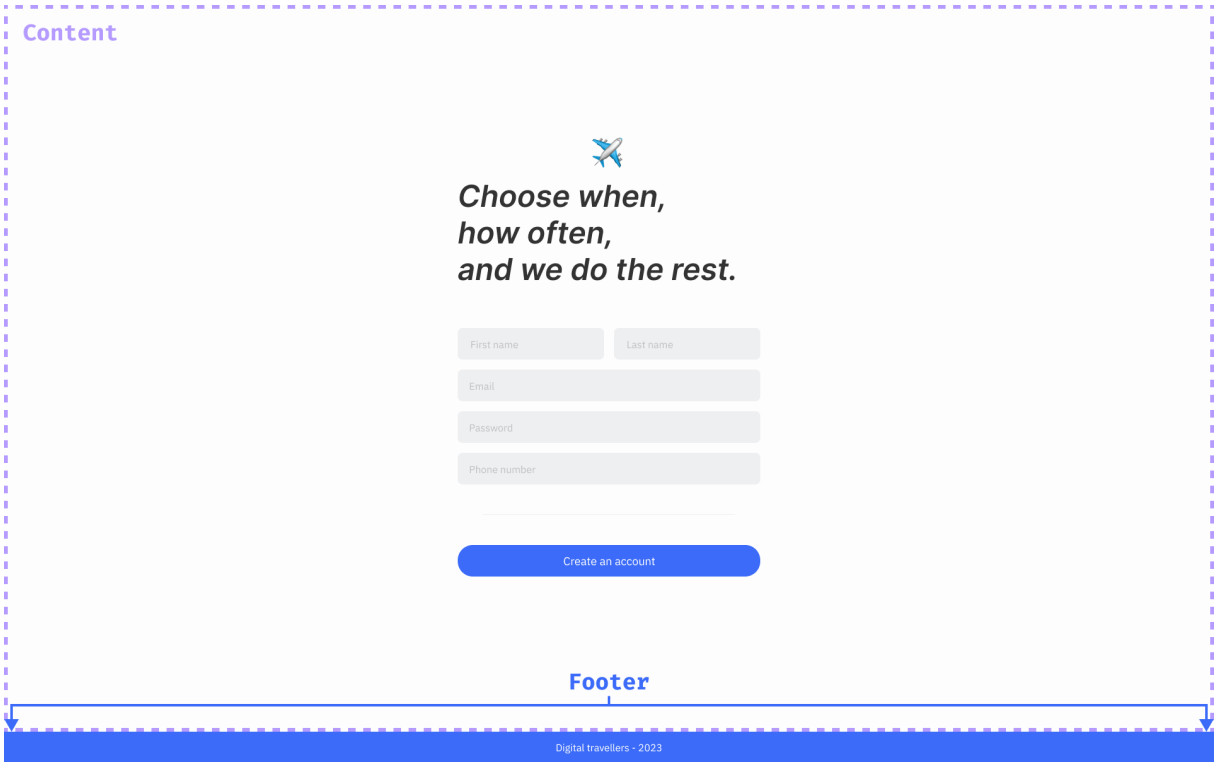

Figure 6.12: Web version sign-up layout

The only difference with the mobile version is the colour of the footer. This logic can be updated by using CSS media queries, as it will be faster than using JavaScript. Furthermore, since the styles has been added using TailwindCSS, media queries are even simpler.

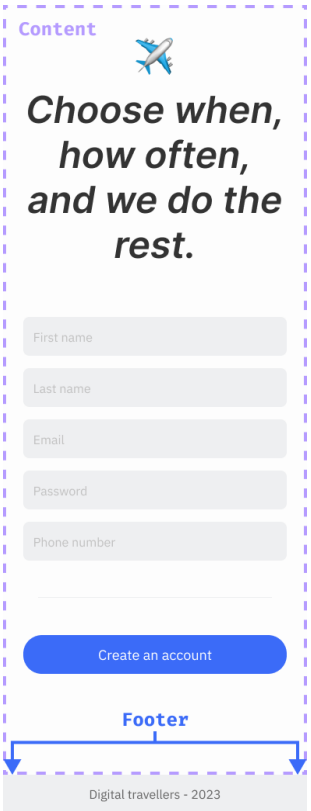

Figure 6.13: Mobile version sign-up layout

6.3.2.2 Authenticated layout

The authenticated layout is slightly more complex than the authentication layout. It is composed of the 4 following parts:

- Green part (top-left rectangle) is a section to display the logo of the application, in order to add branding material inside the application.
- The **navigation header**, which does not strictly contain navigation elements, it displays information about the current page status. In this case, it displays the dashboard title, as well as a message underneath.
- The **sidebar**, which internally contains a sidebar menu. This sidebar contains the navigation items and is going to display the current page in which the user is. This last feature has been added during the development, but was not contemplated in the initial designs.
- Finally, the **main content** which is the part that will be constantly updated, when changing pages or via user interaction.

Therefore, the developer will only need to take into account the changes inside the **main content**, which, in the dashboard's case, involves having a card and displaying the table with the alerts.

I also took the opportunity to mark with grey rectangles some of the components that have already been defined and implemented, which in this case are the title of the page and the subtitle. Moreover, the **sidebar menu** will also contain typography items, which are components that have already been defined, simplifying the implementation of the sidebar. It is worth mentioning that there are other components such as the content button or the content title, that are also already defined components.

Another point that has been taken into account has been the proper usage of markup language. The following image illustrates the usage of the HTML components:

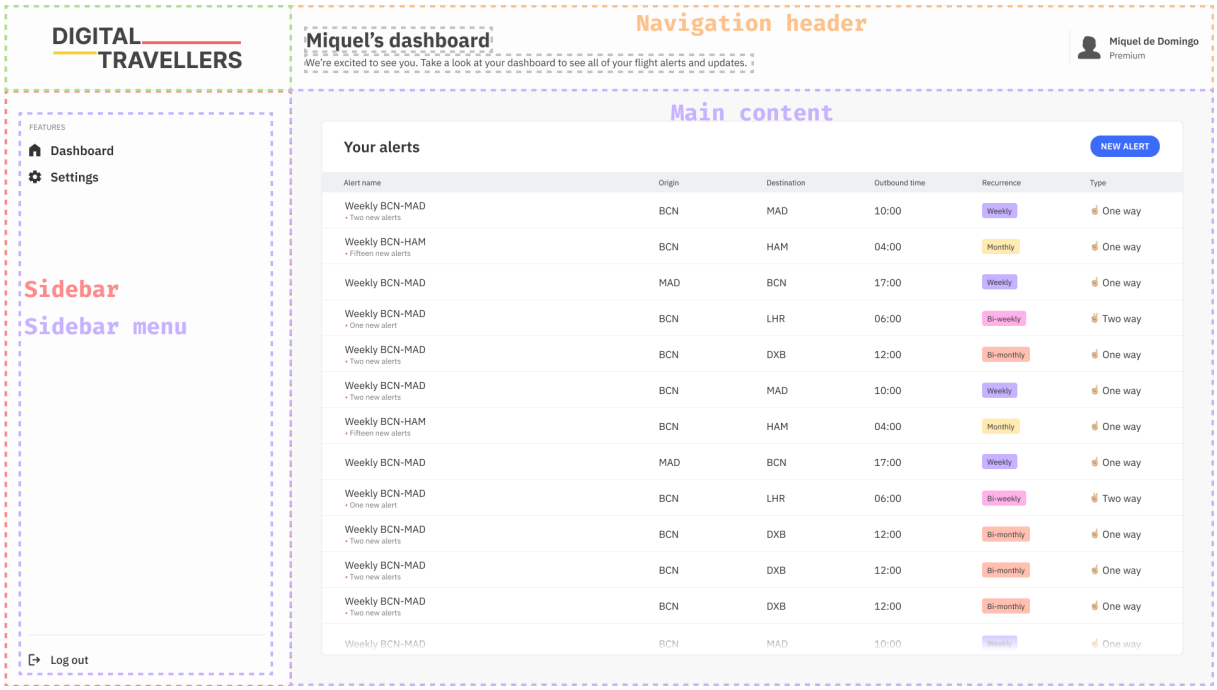

Figure 6.14: Semantic HTML markup usage

Without taking into account the sizes, the layout is composed according to expected markup. One could argue if an `article` should be used over the `section`. However, according to the MDN documentation:

- **Article**. *The* `<article>` *HTML element represents a self-contained composition in a document, page, application or site, which is intended to be independently distributable or reusable.* [16]
- **Section**. *The* `<section>` *HTML element represents a generic standalone section of a document, which does not have a more specific semantic element to represent it.* [17]

Taking both definitions into account, the `section` component semantically fits better than the `article` component.

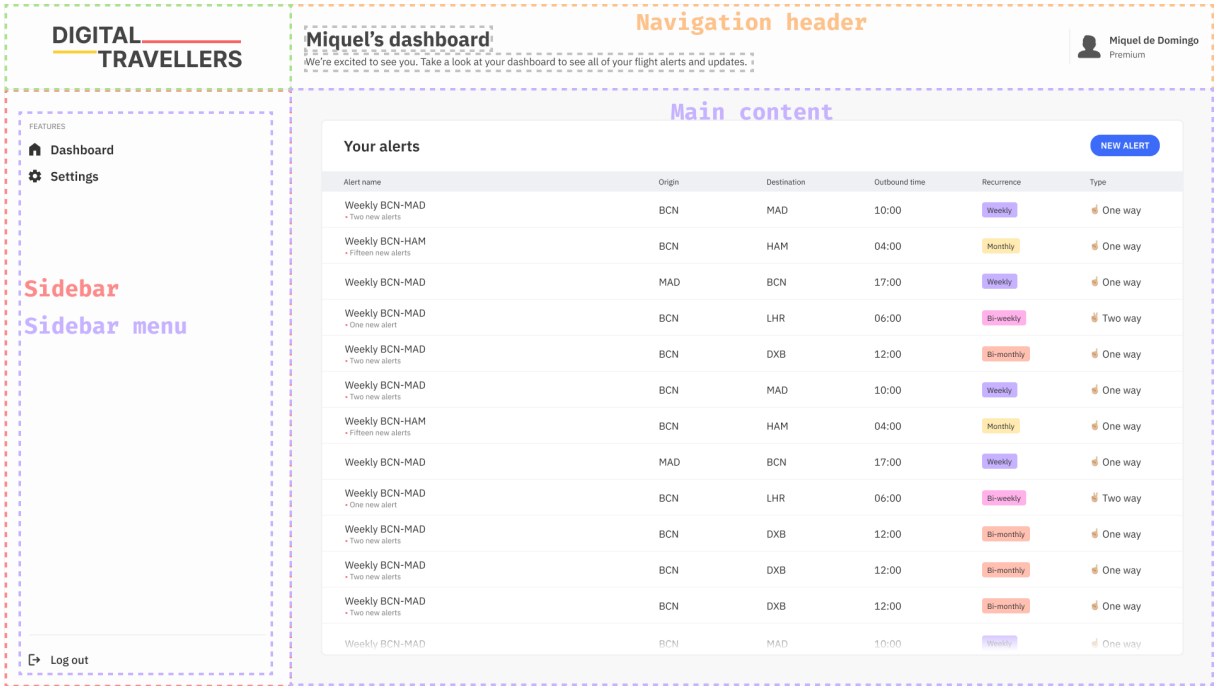

Figure 6.15: Desktop version dashboard layout

Finally, the layout is compared with the mobile version layout. As can be seen, the sidebar and branding element have been removed as they do not fit and do not provide value for mobile users. However, the navigation header and content have been retained. The HTML markup remains relevant, so the previous markup diagram still applies.

However, the user still requires some form of navigation or sidebar. This navigation has been hidden in the top-right button, which functions as a hamburger menu. Initially, the idea was that clicking the cog button would redirect the user to the settings page. However, this option did not account for the addition of new pages, and implementing a hamburger menu that opens a popover or full-screen overlay was deemed a better and more scalable solution.

Figure 6.16: Mobile version dashboard layout

6.3.3 State management

Finally, the last missing piece of the application is the state management of it. State management is the process of maintaining the information of an application's across multiple related data flows. In React, it is the process of managing and synchronizing data within an application, and apply the changes to the user interface accordingly.

React provides a built-in mechanism for state management named `useState` (a hook). This hook allows functional or modern components[5] to have their own internal state. With the state definition and its updater function, React handles the management of state changes and triggers re-renders when the state is updated.
However, as applications grow in complexity, the need for a most robust and scalable state management solution is required. Though React provides a built-in tool named `Context`, there are many other solutions that provide better performance such as jotai [18], zudstand [19], Recoil [20] or React Query [21][6].

Given that there are many possible state management solutions, the chosen solution depends on various factors such as project size, complexity, or even personal preferences. It is important to consider trade-offs between simplicity, performance, development efficiency, and current maintenance status[7].

For the current application, React Query (or TanStack Query) has been preferred since not only provides a state management solution, but also a data fetching system.

[5] Before React 16, components were class based.
[6] Currently rebranded as TanStack Query, though within the community it is still known as React Query.
[7] This point is sometimes overseen by developers. If third-party library has to play such an important role in an application, it is important that this library is currently maintained and up-to-date with current technologies.

6.3.3.1 React Query

React query provides a declarative and intuitive approach to managing asynchronous data, making it easier to interact with the server, while also providing caching utilities, and state synchronization.

At its core, React Query revolves around the concept of queries and mutations:

- Queries are used to fetch data from an API.
- Mutations are used to handle data updates, such as creating, editing or deleting resources.

When a query is executed, React Query will take care of the lifecycle of the data fetching, by keeping track of the process. Under the hood, it manages data caching and background re-fetching, ensuring that data remains up to date, without unnecessary network requests. The caching system is flexible and efficient enough to allow developers to control how data is stored and invalidated.
One of the most used features from React Query is the returned status of the query or mutation execution. For each hook run, it is possible to know its status:

- `Loading`. Property that allows you to know if the query is being fetched. In this case, RQ[8] has checked if there was data in the cache but has not found any.
- `Error`. If the query or mutation resulted in an error (i.e. backend returned non-successful response), the error will be stored in this property.
- `Success`. The query or mutation has received a response, without an error, and it is ready to display the data received if any.
- `Idle`. Queries are run by default on mounting the component. However, it is possible to skip the execution of it. If that is the case, the query is left on idle, until the skip condition is false (so it can be executed).

React Query mutations are a key aspect of the React Query library that enable easy handling of data updates, such as creating, updating, and deleting resources. Mutations provide a declarative way to interact with the server and manage the state of the application based on these updates. React Query's `useMutation` hook allows components to initiate mutations and provides functions to execute the mutation, handle success and error cases, and update the cache accordingly. With mutations, developers can perform optimistic updates, where the UI is instantly updated with the expected result before the server responds, providing a seamless user experience. React Query also supports various options for configuring mutations, such as defining custom update functions, invalidating related queries after a mutation, and providing optimistic response data.

[8] React Query

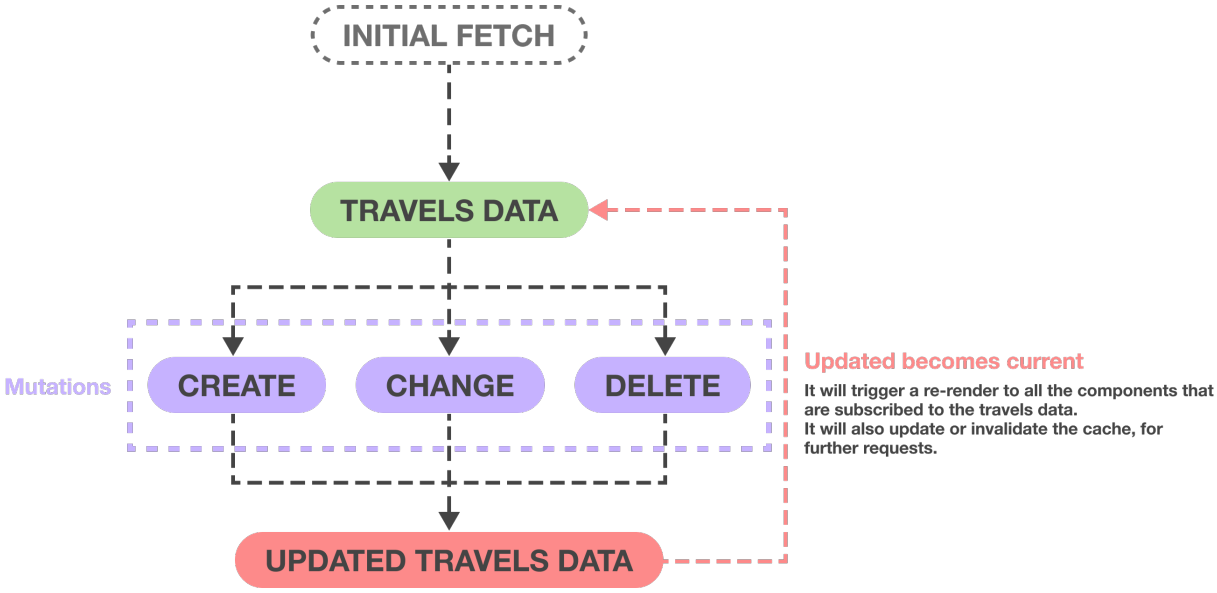

Figure 6.17: React Query travels data flow

The picture above briefly illustrates how React Query interacts with the travels' data. As it can be seen, an initial fetch is performed, which will load the list of alerts in the cache and in the local state. Then, if by any chance the user either creates, updates or deletes an alert, the previous state will become invalid and will be updated accordingly. However, not all mutations must invalidate some fetched data. For example, in order to authenticate the user, mutations are also used, yet they do not invalidate any previously fetched data.

React Query helps with state management by providing a unified and consistent approach to data management across the application. It abstracts away the complexities of managing remote data and provides a single source of truth for accessing and updating data. Components can easily subscribe to and interact with the cached data using the provided hooks, reducing boilerplate code and making state management more intuitive.

To sum up, all state management has been is being managed by React Query, simplifying the frontend code, as there is no need for any state management logic required, other than the React Query hooks.
Since the user is not updated because the settings page has not been implemented, there is no need to keep user state as part of the React Query client. Alternatively, the React Context API has been used, since it is suitable enough for our current logic.

6.3.4 Integration over unit testing

The topic of preferring integration to unit testing for the frontend application has briefly been introduced in previous sections. However, a justification is required and such is the goal of this section. There are several reasons, and my own experience developing frontend react applications, that have pushed me to prefer integration to unit testing.

First key factor, by applying hexagonal architecture, an effort was made to keep the domain logic in a separate package, which has already been extensively unit tested and integration tested. This ensures that the core functionality of the application is thoroughly validated, reducing the need for duplicative unit tests in the frontend components.

Another factor that has influenced this decision has been the usage of a tool named *Mock Service Worker* (MSW) alongside React Query, which has been explained in the previous section. MSW allows the mocking of API requests and responses, enabling comprehensive integration

testing of data fetching and management without the need for actual network calls or other complex mocking set up. Furthermore, it allows the developers to simulate a semi-real environment, as it provides utilities to simulate the delay between API calls, while maintaining the integrity of the results. React Query, on the other hand, provides a powerful data caching and synchronization mechanism, which simplifies the testing of data-driven components that relied upon asynchronous data fetching.

The presence of non-complex forms, such as the authentication ones, and complex forms, such as the creation of a travel, emphasized the need for integration testing. Forms often involve intricate interactions and validations that span over multiple components and user or data flows. By focusing on integration testing, it is possible to test the entire for workflow, including user interactions, data submission, and error handling, ensuring a robust and seamless user experience.

Additionally, the emphasis has been placed in testing user flows rather than individual component integrity. Integration testing has allowed the simulation of realistic user scenarios, such as the interaction with various components, and validation of the overall application behaviour. This approach has provided grater confidence in overall user experience and ensured that the application functions as expected in a real-world usage.

It is also important to recognize that achieving high code coverage through unit tests does not necessarily guarantee better code quality or a bug-free application. Focusing solely on unit testing can sometimes lead to an excessive focus on isolated component behaviour, overlooking potential issues that may arise from the interactions between components or external dependencies. Integration testing, on the other hand, helps uncover these potential integration issues and ensures that the applications and components behave correctly as a whole.

As a disadvantage, it is important to note that integration tests tend to require more execution time, even though it should not be extremely high (consider between 2 minutes to 10 minutes at most). In comparison, unit test will run faster.

In conclusion, by prioritizing integration testing in the frontend part of the application (excluding the domain and the shared UI library package), the development team has been able to validate the user flows and data management. The usage of tools like MSW and React Query, along with the consideration of complex forms and user experience, have further supported the decision to favour integration testing over unit testing. Ultimately, this approach has provided a more comprehensive and reliable testing strategy for the application

As a last note for this section, there are some utilities such as helper functions that have been unit tested, as in this case it has been considered important to ensure the behaviour of them.

6.4 CI/CD

Last aspect to talk about regarding the implementation process, is the continuous implementation and continuous development setup. Since the team codebase is hosted in GitHub, the easiest tool to set up is GitHub Actions. Furthermore, Nx's documentation has tutorials[22] regarding the implementation of the continuous integration in GitHub Actions.

In Nx, each project is configured with a set of commands tailored to its specific type. For example, the `context` package includes commands such as `lint`, `build` and `test`. The `lint` commands checks for static errors in the package codebase, while `build` handles bundling and generates the transpiled code. The `test` command executes all the tests associated with the `context` package.
One of the notable features of Nx is command parallelization, which allows multiple commands

to run concurrently, even within projects. This capability significantly reduces the overall execution time in CI workers, improving its efficiency. Furthermore, Nx Cloud offers the option to cache command output, which means that if the package's codebase remains unchanged, Nx can rely on the cached output, rather than re-executing the command, further optimizing performance. Nonetheless, this feature has not been used as it is a paid feature.
Another standout features of Nx is its ability to identify updated packages. It intelligently determines which packages have been modified and selectively runs the associated commands only on those affected packages and their dependent packages. This targeted approach reduces the complexity of the CI/CD environment setup, and ensures that only necessary actions are performed, saving time and resources.

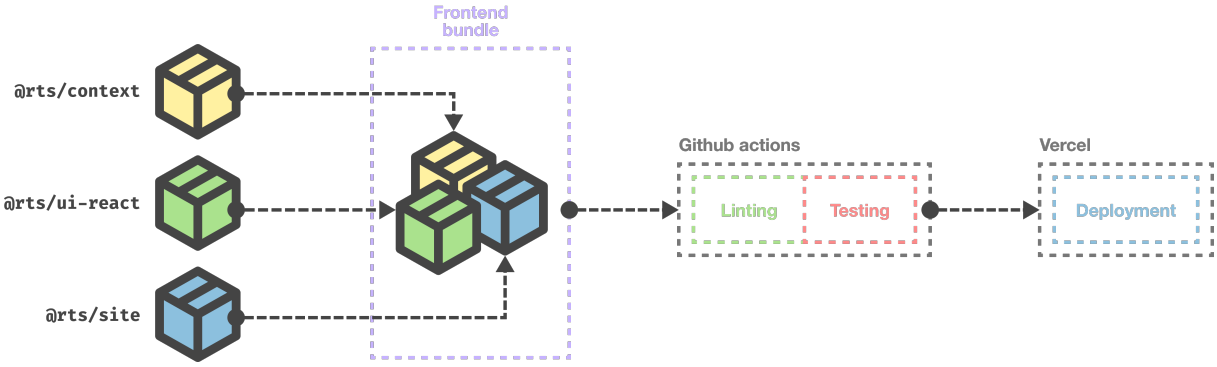

Figure 6.18: CI/CD flow

As it can be seen in the above diagram, the last step of the flow is the deployment to Vercel. The Vercel option was chosen because of its simplicity. By configuring certain aspects of the repository, the Vercel infrastructure it is capable to identify and understand the codebase, and properly deploy the application. Since Nx provides all build commands and utilities, there is no need to configure anything extra, aside from setting the command to use.
Vercel is an extremely powerful tool, however, due to the lack of time and simple implementation, aside from the automated deployment on `main` branch merge, nothing else has been used.

6.5 Code structure

As the final section of this chapter, the goal is to explain the code structure of the various packages and the reasoning behind it.

6.5.1 Context package

The context package follows a modular organization, as recommended by hexagonal architecture principles. It adheres to a layered architecture, with clear separation between application, domain, and infrastructure layers

- `src`. The source folder contains the source code of the project excluding the tests. It is divided following the hexagonal architecture principles. Some layers may include a `shared` submodule, which contains shared code within the different sub-modules.
    - `app`. The `app` folder represents the application layout of the project. Inside, it is organized by submodules for different application implementation, such as the authentication (`auth`) and the travels (`travel`).
    - `domain`. The directory represents the domain layer of the project. It is also divided in submodules, which should match the application ones.

    - `infra`. The infrastructure directory represents the custom definition of the infrastructure layer. As explained in previous chapters, the infrastructure is divided in queries and queriers.

- `test`. The test package contains the unit and integration tests for the context package. Internally, it clones the same structure as the `src` folder.

    - `app`. The `app` directory contains the tests for the application layer, mirroring the structure in `src/app` directory.
    - `domain`. The `domain` directory includes the test files for the domain layer, mirroring the structure in `src/domain`. It also includes `mothers` directories, which contain randomized data generation for the different entities.
    - `infra`. The directory includes the test files for the infrastructure layer, also mirroring the structure in `src/infra`. It also contains the setup and handlers for the MSW[9] that it is used for integration tests.
    - `util`. Contains shared utilities for the testing folder.

6.5.2 UI React library

The UI react library follows a standard *folder per component* or related components. Inside each folder, it can be found:

- The implementation of the component or various components. One component per file.
- A `spec` file with the tests for the component.
- The `stories` file type that contains the implementation of the component to be visualized in the Storybook application.
- Possibly a `d` or declaration file, for TypeScript types. If there are few types, this file is not contained.

This *folder per component* is extremely useful for libraries, because as the library grow, it can simplify the code splitting and tree shaking. This benefits will reduce the bundled code size, reducing the amount of code sent to the user's browser.

6.5.3 Site application

The site application is divided in the following folders:

- `components`. This folder contains reusable components used throughout the application. It also follows the *folder per component* in order to reduce complexity and keep the codebase organized. As a personal preference, I prefer organization through folders rather than files.
- `config`. This folder holds configurations files for the application, as well as global constants such as the route paths of the app. The goal is to have a unique source of truth for such topics.
- `domain`. The directory represents the domain-specific code for the different pages. In this case, it has been preferred to keep a domain per specific route. This allows the domain to keep escalating without issues.

[9] Mock service worker.

- `hooks`. This directory contains custom React hooks used within the application. It also contains the abstraction of the React Query hooks logic into other hooks.
- `layouts`. This directory houses the layout components used for different pages of the application.
- `msw`. This directory contains the implementation of the MSW used for local development and testing.
- `pages`. The `pages` directory holds the Next.js page components that define the routes and rendering logic for different URLs of the application.
- `public`. This directory is used for static assets that are served publicly, such as images or fonts.
- `styles`. This directory contains global CSS styles, as well as a reference for the Tailwind-CSS utilities.
- `test`. Even though each element contains the text in the same file level, the `test` directory contains shared utils for the tests.

Overall, the folder structure is Next.js standard, with some customizations, since React does not provide an official or recommended way of structuring the code.

7. Results

8. Conclusions

The flight information requirements for individuals who undertake frequent trips are increasing as technology and access to data advance. Therefore, relying on one-off search tools that require users to conduct individual searches to meet their travel needs is no longer feasible, especially when they have fixed routes that repeat in specific daily cycles, whether for work, academic, or established project purposes.

The project and application have been developed using well-known patterns and architectures widely used in companies and open-source projects. These architectures have proven to meet the application requirements, providing scalability, maintainability, and robustness. The hexagonal architecture facilitated seamless integration with external services while maintaining a layered and separated architecture. Additionally, the adoption of DDD (domain-driven design) enabled consistent communication among the frontend, backends, and product owner, ensuring a shared understanding of the system's domain.
The utilization of these patterns, ideas, and architectures has significantly streamlined development time for both frontend and backend components, while ensuring the application's maintainability and scalability.

In modern times, there are numerous solutions that simplify the deployment of projects, ranging from simple to complex ones. These solutions often rely on cloud services, such as Google Cloud or AWS, to name the most famous ones. In this case, Google Cloud and Vercel have been chosen, simplifying the deployment process and eliminating the need for local or internal servers. The utilization of cloud services offers various benefits, including automated scalability, high availability, efficient management of infrastructure resources, and built-in security measures.

Alongside the cloud environment, all projects have taken advantage of CI/CD (continuous integration/continuous development). The automation of code building, testing and deployment have enabled faster delivery of new features and bug fixes. Furthermore, it has fostered the team in terms of collaboration and efficiency by ensuring stability and code quality *throughout all stages of the development cycle*.

In the context of frontend development, the benefits of utilizing a hexagonal architecture have become evident. Although it is not common to employ such an architecture in frontend projects that use component-based frameworks, it has demonstrated significant potential in various aspects. These advantages include code sharing between frontends, simplified code splitting to reduce the amount of JavaScript served to users, and improved maintainability and scalability of domain-specific code.
One particular note is the ability to completely separate domain logic and implementation from the frontend application. This separation has streamlined the frontend development experience and simplified UI testing, which is often complex. However, it is important to acknowledge that companies aiming for rapid product delivery may choose to keep business logic tightly coupled with the frontend, as decoupling can be time-consuming.

Furthermore, the implementation of CI/CD in frontend development has reaffirmed the importance of this practice in delivering higher code quality and a more robust application.

9. Future work

Bibliography

[1] Unspecified author. Unspecified last update. **On:** *Business Travel Statista* [online]. May 15th 2023. **Available on**: https://www.statista.com/markets/420/topic/548/business-travel/#statistic2.

[2] CHEN, Guang, ENGER, Will, SAXON, Steve, and YU, Jackey. Unspecified last update. **On:** *What can other countries learn from China's travel recovery path?* [online]. May 15th 2023. **Available on**: https://www.mckinsey.com/industries/travel-logistics-and-infrastructure/our-insights/what-can-other-countries-learn-from-chinas-travel-recovery-path.

[3] TravelPerk. Unspecified last update. **On:** *Back to business: from offices to business travel, how does the US feel about returning to work?* [online]. May 15th 2023. **Available on**: https://www.travelperk.com/blog/back-to-business-travel-how-does-the-us-feel-about-returning-to-work.

[4] GHIJS, Stefaan. Unspecified last update. **On:** *The business flight travel survey statistics* [online]. May 15th 2023. **Available on**: https://flyaeolus.com/blog/2017-business-travel-statistics/.

[5] KUNST, Alexander. September 3rd 2019. **On:** *Business travellers who book their own hotels U.S. 2017* [online]. May 15th 2023. **Available on**: https://www.statista.com/statistics/719856/business-travelers-who-book-their-own-hotels-us/.

[6] SHEIVACHMAN, Andrew. October 27th 2016. **On:** *Milennials are new the most frequent business travellers* [online]. May 15th 2023. **Available on**: https://skift.com/2016/10/27/millennials-are-now-the-most-frequent-business-travelers/.

[7] Unspecified author. Unspecified last update. **On:** *IATA Annual review 2022* [online]. May 15th 2023. **Available on**: https://www.iata.org/contentassets/c81222d96c9a4e0bb4ff6ced0126f0bb/annual-review-2022.pdf.

[8] Nx. **Available in**: https://www.nx.dev.

[9] NextJs. **Available in**: https://www.nextjs.org.

[10] Cypress. **Available in**: https://www.cypress.io.

[11] Unspecified author. Unspecified last update. **On:** *Stakeholder Management using the Power Interest Matrix - Solitaire Consulting* [online]. May 15th 2023. **Available on**: https://www.solitaireconsulting.com/2020/07/stakeholder-management-using-the-power-interest-matrix/.

[12] TravelPerk. **Available in**: https://www.travelperk.com.

[13] Hopper. **Available in**: https://www.hopper.com.

[14] Skyscanner. **Available in**: https://www.skyscanner.com.

[15] Routing: Pages and layout. **Available in**: https://nextjs.org/docs/pages/building-your-application/routing/pages-and-layouts#layout-pattern.

[16] <article>: The Article Contents element. **Available in**: https://developer.mozilla.org/en-US/docs/Web/HTML/Element/article.

[17] <section>: The Generic Section element. **Available in**: https://developer.mozilla.org/en-US/docs/Web/HTML/Element/section.

[18] Jotai, primitive and flexible state management for React. **Available in**: `https://jotai.org`.

[19] Zudstand. **Available in**: `https://zustand-demo.pmnd.rs`.

[20] Recoil. **Available in**: `https://recoiljs.org`.

[21] TanStack Query. **Available in**: `https://tanstack.com/query/latest`.

[22] Setting up GitHub Actions. **Available in**: `https://nx.dev/recipes/ci/monorepo-ci-github-actions`.