

# Inclusive Trip Planner



*Project Summary for M2 Admission*

David Cuahonte Cuevas

July 2025

# Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>Project Objectives</b>	<b>3</b>
<b>3</b>	<b>Methodology</b>	<b>4</b>
<b>4</b>	<b>Project Management</b>	<b>5</b>
<b>5</b>	<b>Software Architecture</b>	<b>7</b>
<b>6</b>	<b>UML Diagrams and Visual Mockups</b>	<b>9</b>
<b>7</b>	<b>Testing and Quality Assurance</b>	<b>14</b>
<b>8</b>	<b>Security and Authentication</b>	<b>15</b>
<b>9</b>	<b>CI/CD Pipeline</b>	<b>16</b>
<b>10</b>	<b>Constraints and Limitations</b>	<b>18</b>
<b>11</b>	<b>Conclusion and Future Perspectives</b>	<b>19</b>

# 1. Introduction

Planning an accessible trip remains, to this day, a real challenge for many individuals with disabilities. While significant progress has been made in the design of digital interfaces and in adapting certain public spaces, most trip planning platforms only partially address constraints related to mobility, sensory perception, or specific cognitive needs. This shortcoming often leads to feelings of uncertainty, frustration, and sometimes even to giving up on traveling independently altogether.

It is within this context that I designed the *Inclusive Trip Planner* project. This mobile application, conceived from the outset with inclusion in mind, offers accessible, pre-designed itineraries enriched with explicit accessibility metadata. The interface, clean and intuitive, was developed to remain understandable and usable by everyone — including people unfamiliar with digital tools or those with physical limitations.

Beyond its functional contributions, this project enabled me to apply all the technical skills I acquired throughout my training. It is built on a modern and robust technological foundation: the **frontend** was developed using **Jetpack Compose** (Kotlin), while the **backend** relies on a modular architecture based on **Java Spring Boot**, **PostgreSQL**, and **Docker**.

Initially developed as part of my end-of-studies thesis, the project was then strengthened to meet the expectations of the Master 2 program. In this context, I integrated **unit tests** on the **frontend**, implemented a continuous integration pipeline (**CI/CD**), and prepared the system for production deployment. This document outlines the key stages of the project, detailing the design, technical choices, concrete implementations, and the system's prospects for evolution toward large-scale use.

## 2. Project Objectives

*Inclusive Trip Planner* serves a dual purpose: to address a concrete societal need in terms of inclusive mobility, and to demonstrate the rigorous implementation of a full-stack software project — from requirements definition to testing and deployment.

From a functional perspective, the application aims to provide an accessible and relevant tool for users with disabilities. The main goal is to allow each person to discover and plan itineraries tailored to their specific constraints. This involves an interface that is easy to use, the ability to filter routes based on accessibility criteria, and a personal space to save user preferences.

From a technical standpoint, the project set out to:

- **Analyze the limitations of existing solutions** in the area of accessible travel;
- **Define a complete and modular architecture** that integrates the frontend and backend smoothly;
- **Develop a functional MVP** with data persistence, secure authentication, and reactive navigation on mobile;
- **Validate critical user flows** through automated tests covering all essential entities and interactions;
- **Document all technical and methodological choices** with a focus on sustainability and future openness.

Rather than accumulating unfinished advanced features, the project focused on delivering a coherent and robust functional scope. The roadmap ahead follows a logic of continuity: adding user feedback, enriching metadata, and exploring new application contexts related to inclusive mobility.

### 3. Methodology

The development of *Inclusive Trip Planner* followed an iterative, pragmatic, and test-driven approach, centered around the real needs of end users. The primary goal was to deliver a fully functional MVP while laying solid foundations for maintainability and scalability.

The work was structured into three main phases:

- **Phase 1: Architecture Setup:** Initialization of the backend with **Spring Boot**, **PostgreSQL**, and **Docker**, alongside the creation of a mobile skeleton in **Jetpack Compose** (Kotlin). Key architectural decisions — including the layered structure, DTO-based mapping, and unidirectional data flow — were defined during this stage.
- **Phase 2: Implementation of Core Features:** Development of critical flows such as user signup, itinerary browsing, and profile customization. Each feature was built in a full-stack manner and manually validated before undergoing testing.
- **Phase 3: Testing, CI Integration, and Continuous Improvement:** Addition of backend integration tests and frontend unit tests. The UX and API consistency were refined progressively through short feedback loops and validation cycles.

The entire development was carried out by a single developer, which ensured full coherence between business logic, API contracts, and frontend behavior. Communication between the frontend and backend relies on **Retrofit** and **ViewModels**, using **coroutines** to handle state reactively within Jetpack Compose.

**Accessibility** guided the decision-making process from the early stages — both in data modeling and UX. For example, the onboarding flow was designed to prioritize the collection of user needs in order to enable relevant itinerary filtering from the very first screens.

The project also placed strong emphasis on modularity and testability from the outset. The backend follows a strictly layered architecture (Controller → Service → Repository), while the frontend is organized around **ViewModels**. Version control through **Git** enabled secure iteration, with strict isolation of feature branches and systematic local validation before any merge.

## 4. Project Management

The development of *Inclusive Trip Planner* was tracked using a structured **Jira board**, enabling clear visualization and organization of all project tasks within an agile workflow. This setup facilitated prioritization, traceability, and steady progress across the design, implementation, testing, and documentation phases.

Tasks were centralized as *tickets*, grouped into thematic **epics** (e.g., *Testing and Quality Assurance*, *DevOps and Deployment*, *Final Documentation*), and distributed across a main **sprint** running until the end of July 2025. Each ticket was tagged with a functional category (*frontend*, *backend*, *CI/CD*, *documentation*) and followed a *kanban-style* pipeline with the following stages:

- **To Do**: tasks planned and ready to start;
- **In Progress**: tickets actively under development;
- **In Review**: completed items pending validation or testing;
- **Done**: fully implemented, tested, and integrated features.

The use of Jira ensured full transparency of project progress, while effectively structuring deliverables aligned with Master’s requirements (C2.2.x and C2.3.x objectives). Ticket lifecycle management was coupled with a strict Git workflow (feature branches, pull requests validated after testing), ensuring project stability and fostering a development process ready for continuous integration.

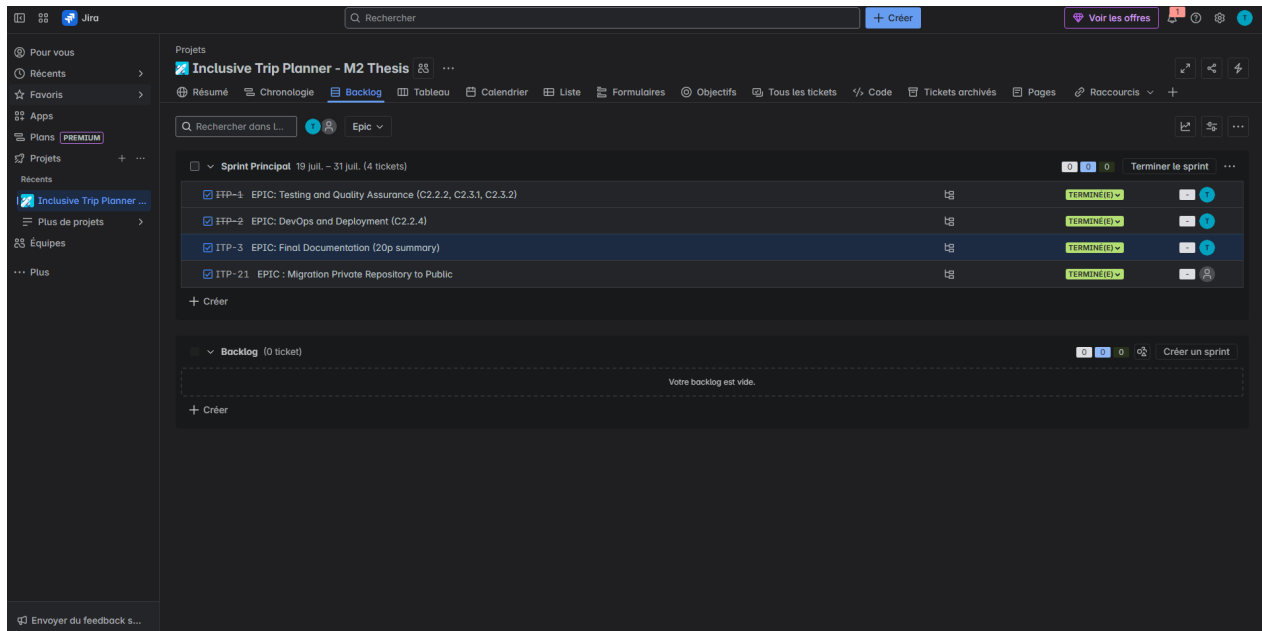


Figure 1: Main sprint tracked in Jira, epics delivered by July 30, 2025

## Budget Estimate

The project was developed by a single person during the MVP phase. The estimate below is based on a daily cost close to the minimum wage for software development in France. Each line item includes a comment about potential evolution in case of project scaling.

## Indicative Budget Estimate

Item	Details	Estimated Cost
Development time	Consolidation and documentation work, estimated at €130/day (developer minimum wage)	€1,300
Backend hosting (Render)	€7/month for 3 months – <i>Subject to change based on PCU and MAU</i>	€21
CI/CD (GitHub Actions)	Free plan – <i>Valid for current version only</i>	€0
Database storage (Docker)	Local usage – <i>To be migrated to the cloud at scale</i>	€0
Testing and documentation	Writing tests, functional validations, and technical documentation, estimated at €130/day (QA minimum wage)	€390
<b>Total Estimated Cost</b>		<b>€3,661</b>

Table 1: Indicative budget estimate for the *Inclusive Trip Planner* project

### Scaling Notes:

- This budget reflects solo development with no outsourcing.
- In case of scale-up, a **full-stack developer**, a **dedicated QA**, and a **project manager** would need to be added.
- The infrastructure (Render, database, CI/CD) will need to evolve according to usage volume (number of active users, horizontal scalability).
- A revised cost estimate should be prepared based on future business and technical needs.

## 5. Software Architecture

The Inclusive Trip Planner follows a layered client-server architecture, emphasizing modularity, testability, and separation of concerns.

### Backend Architecture

The backend is built with Java Spring Boot and consists of distinct layers: Controller, Service, Repository, and DTOs. Each layer has a clearly defined responsibility:

- **Controllers** expose RESTful endpoints and delegate logic to the services.
- **Services** contain business logic and interact with repositories.
- **Repositories** handle data persistence using Spring Data JPA.
- **DTOs** are used to decouple internal models from API contracts.

#### Example: UserController.java

```
1 @GetMapping("/{id}")
2 public UserResponse getUserById(@PathVariable UUID id) {
3     User user = userService.getUserById(id);
4     return UserResponse.from(user);
5 }
```

Listing 1: Controller layer exposing User endpoints

#### Example: UserService.java (Partial)

```
1 @Transactional
2 public User createUser(UserRequest dto) {
3     String platform = dto.getPlatform() != null ? dto.getPlatform().
4         toUpperCase() : "EMAIL";
5     if (platform.equals("EMAIL")) {
6         if (dto.getPasswordHash() == null || dto.getPasswordHash().isBlank
7             ()) {
8             throw new IllegalArgumentException("Password is required for
9                 EMAIL platform users");
10         }
11     }
12     ...
13     return userRepository.save(user);
14 }
```

Listing 2: Service layer handling user creation



## Integration Testing

Integration tests are written using '@SpringBootTest' and validate real database interaction with authentication flows.

```
1 @Test
2 void shouldCreateUserAndFetchItById() throws Exception {
3     UserRequest request = UserRequest.builder()
4         .name("Alice")
5         .email("alice@example.com")
6         .passwordHash("secret")
7         .phone("123456")
8         .build();
9
10    MvcResult result = mockMvc.perform(post("/api/users")
11        .header("Authorization", "Bearer " + token)
12        .contentType(MediaType.APPLICATION_JSON)
13        .content(objectMapper.writeValueAsString(request)))
14        .andExpect(status().isOk())
15        .andReturn();
16 }
```

Listing 3: Sample test validating user creation and fetch

## Frontend Architecture

The mobile frontend is built using Kotlin and Jetpack Compose. Architecture is based on ViewModels, Retrofit, and coroutine-based state management.

### Example: ItineraryViewModel.kt

```
1 fun fetchItineraries() {
2     viewModelScope.launch {
3         try {
4             val response = api.getAllItineraries()
5             _itineraries.value = response
6         } catch (e: Exception) {
7             e.printStackTrace()
8         }
9     }
10 }
```

Listing 4: ViewModel fetching itinerary list

### Composable Example: ItineraryDetailsScreen.kt (Excerpt)

```
1 Text(  
2     text = itinerary?.title ?: "Loading...",  
3     fontSize = 26.sp,  
4     fontWeight = FontWeight.Bold,  
5     color = Color.Black  
6 )
```

Listing 5: Composable displaying itinerary information

This structure ensures a clear separation of concerns across backend and frontend components while supporting testability and maintainability.

## 6. UML Diagrams and Visual Mockups

To support the functional design of the application, two UML diagrams were created: a use case diagram illustrating the interactions between the user and the system, and an activity diagram describing a typical onboarding scenario. In addition, three screenshots from the functional mobile application are provided as visual mockups.

## Use Case Diagram

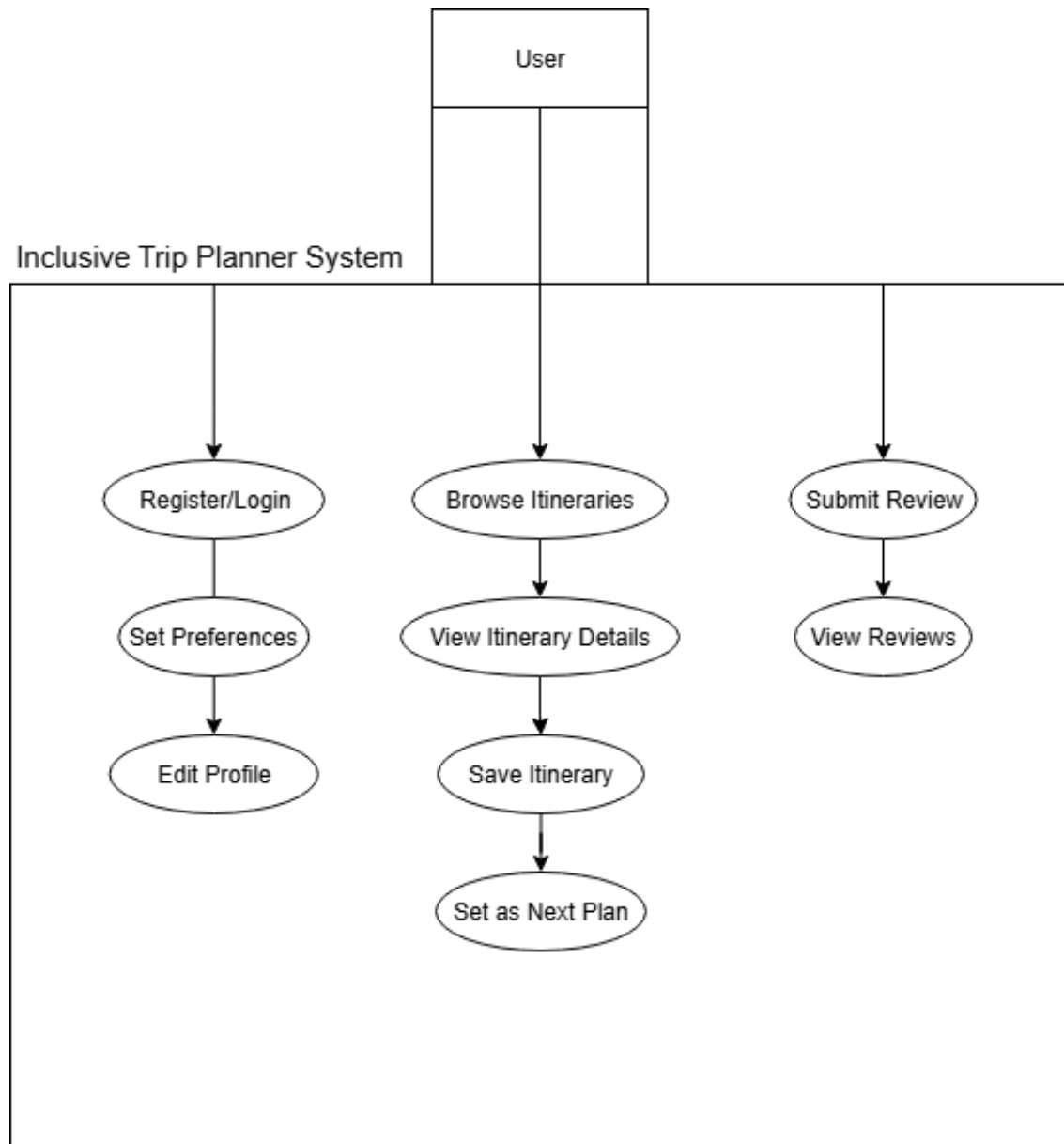


Figure 2: Use Case Diagram: Functional interactions between the user and the system

## Activity Diagram

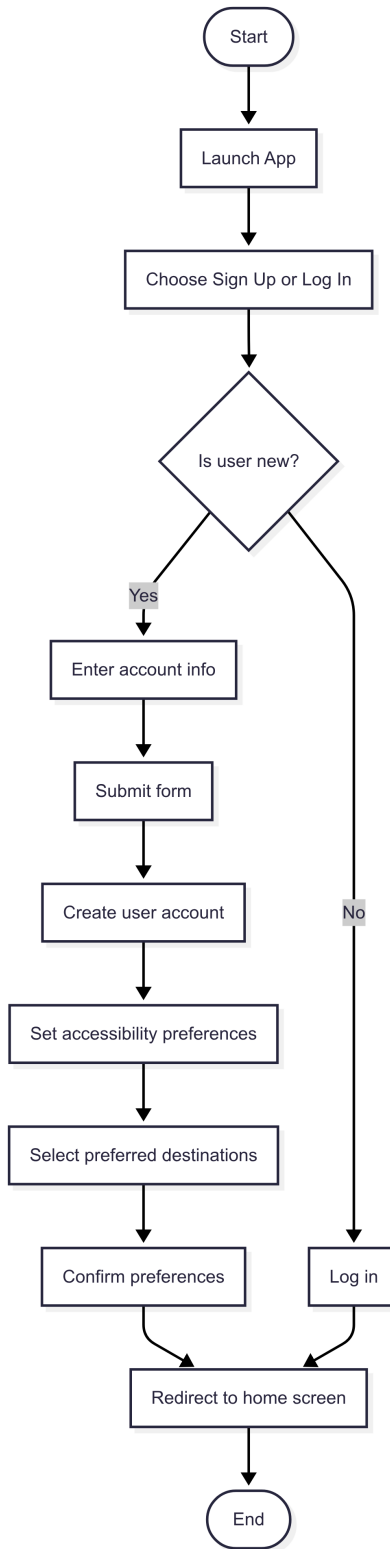
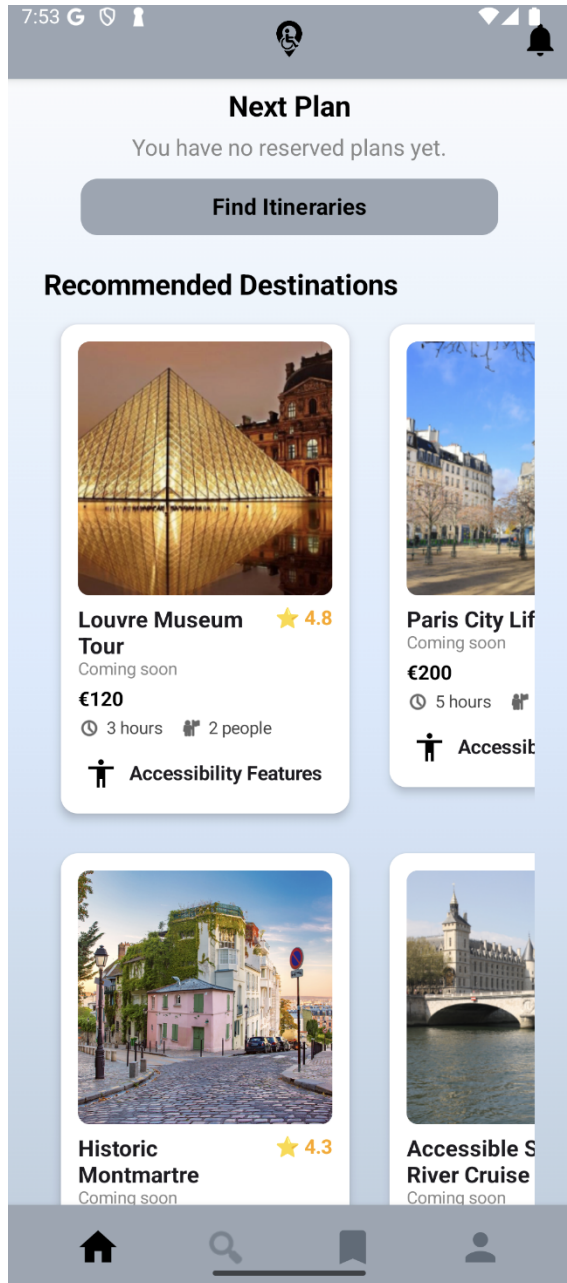


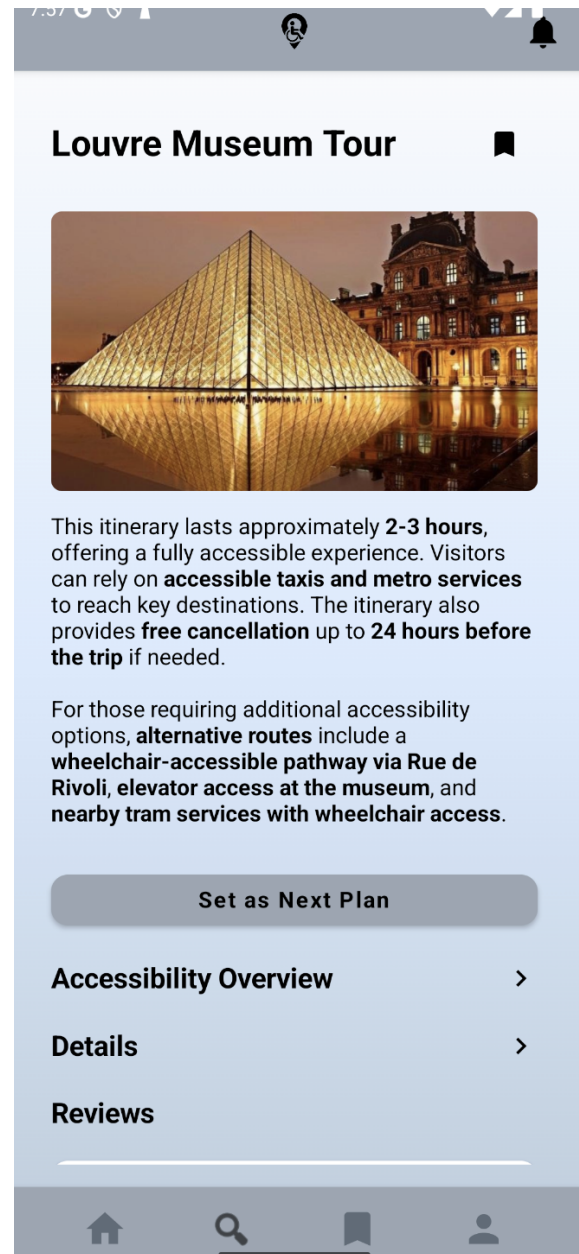
Figure 3: Activity Diagram: Onboarding flow and accessibility preferences selection

## Mobile Screens – Visual Mockups

Key screens from the mobile interface: home and itinerary details



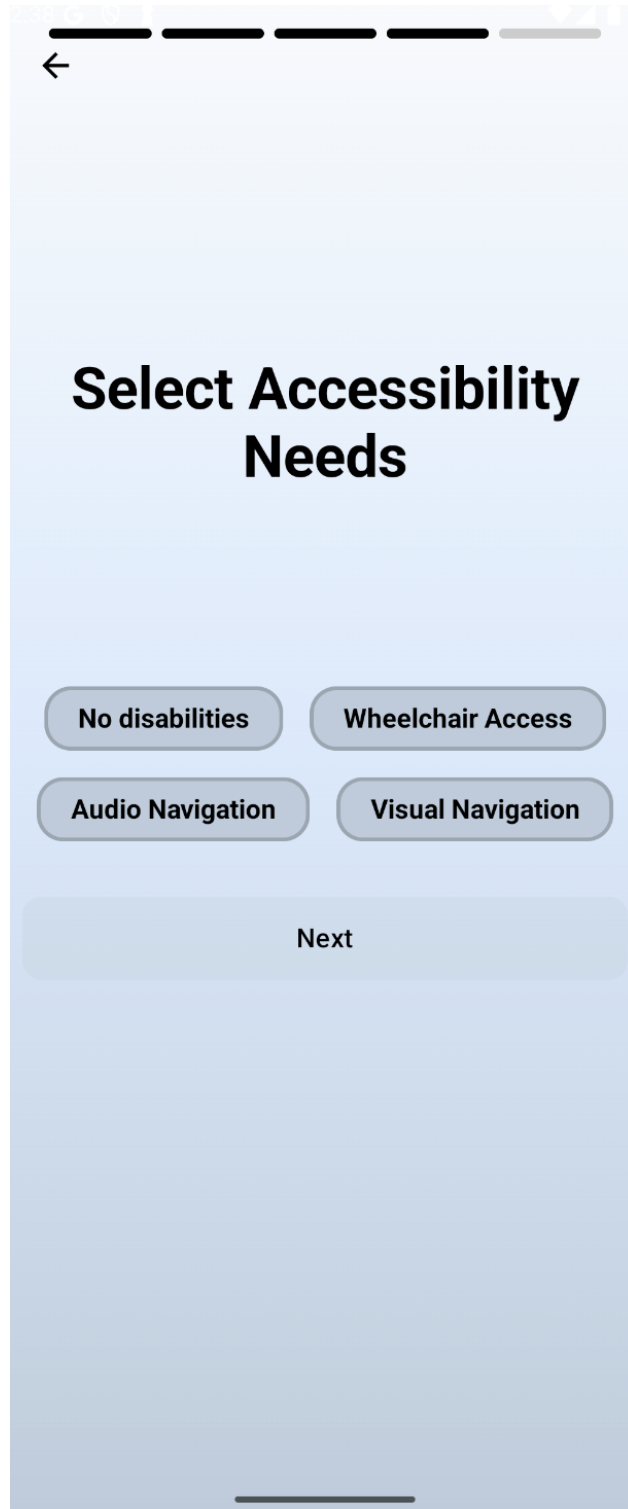
(a) Home screen displaying recommended itineraries and active plan



(b) Itinerary details screen with accessibility information

Figure 4: Mobile mockups of the application

## Accessibility Preferences Screen



A mobile app screen titled "Select Accessibility Needs". At the top left is a back arrow. The title is centered in large bold text. Below the title are four rounded rectangular buttons arranged in a 2x2 grid: "No disabilities", "Wheelchair Access", "Audio Navigation", and "Visual Navigation". At the bottom is a wide "Next" button. The screen has a light blue gradient background and a mobile status bar at the top.

←

# Select Accessibility Needs

No disabilities Wheelchair Access

Audio Navigation Visual Navigation

Next

Figure 5: Accessibility preferences selection during onboarding

## 7. Testing and Quality Assurance

The reliability of *Inclusive Trip Planner* was a central objective throughout the development process. The testing strategy covered both backend and frontend components, with particular emphasis on functional validation, regression prevention, and readiness for future updates.

The **backend** includes **integration tests** for all entities, leveraging the *Spring Boot* testing framework and an in-memory H2 database. These tests cover full request/response cycles, including CRUD operations, entity relationships, and authentication mechanisms. They are based on the `@SpringBootTest` annotation, allowing realistic usage scenarios to be simulated without relying on mocks. An example of such a test was presented in the Software Architecture section.

On the **frontend**, unit tests were written for each **ViewModel**. These rely on *mock APIs* and *coroutine test dispatchers* to simulate network call behavior while verifying that state transitions (`collectAsState()`, loading indicators, etc.) respond correctly to user interactions. The tests are written in Kotlin using native testing tools and are fully compatible with the **Jetpack Compose** environment.

Version control with **Git** further strengthened this quality-driven approach through a stable branching model. Each feature was developed and tested in isolation, and untested code was systematically blocked from being merged into the main branch.

In addition, the project integrates a **CI pipeline** (based on **GitHub Actions**) that automatically executes backend integration tests and frontend unit tests on every *push* or *pull request*. This mechanism ensures continuous validation of the project’s functional stability and actively contributes to preventing regressions throughout the development cycle.

Layer	Test Type	Coverage Scope
Backend	Integration Tests	Full cycle for User, Itinerary, AccessibilityFeature, Settings, Auth
Frontend	Unit Tests	ViewModel logic, state management, API flows
CI/CD	Pipeline Execution	Automated tests on every push/pull request

Table 2: Summary of test coverage across the different application layers

The combination of local tests and an automated pipeline proved essential in ensuring the project’s stability. In particular, CI validations helped prevent regressions during last-minute changes and significantly improved the reliability of deliverables, even in short and intensive iteration cycles.

## 8. Security and Authentication

The *Inclusive Trip Planner* application relies on a stateless authentication system based on **JSON Web Tokens (JWT)** to ensure secure and scalable access control. This architectural choice eliminates the need for server-side session storage, while facilitating integration with frontend clients such as mobile applications.

Security is enforced using **Spring Security**, configured through a custom filter chain. The backend defines granular authorization rules, protecting each endpoint based on user roles and authentication state. The key components of this architecture include:

- A **JWT authentication filter** that intercepts requests, extracts and validates the token, and populates the Spring security context.
- A **password hashing mechanism** using a secure algorithm such as BCrypt, ensuring that credentials are never stored in plain text.
- **Role-based access control**, used to restrict endpoint exposure and enforce strict authorization rules across the API.

JWTs are signed using a server-generated secret and include *claims* such as the user ID, role, and expiration date. When a user logs in or creates an account, the backend returns a signed token, which the mobile application stores locally (e.g., using *shared preferences*) and then sends in the *Authorization* header of every subsequent request.

### Authentication Flow Summary

1. The user sends their credentials to the `/api/auth/login` endpoint.
2. The backend verifies the submitted data and returns a signed JWT.
3. The mobile application securely stores this token locally.
4. For each API call, the token is added to the request headers.
5. A dedicated Spring filter validates the JWT and authorizes access to protected routes.

This system provides a secure, stateless API that can scale horizontally. It also lays the groundwork for integrating external identity providers (such as Google or Facebook via *Firebase Authentication*) or implementing multi-factor authentication mechanisms.

The current implementation remains intentionally open to extension. An OAuth2 login strategy has already been initiated using Firebase tokens, and the backend architecture is prepared to incorporate new filters or providers as needed.

In summary, this JWT-based authentication system strikes a balance between ease of use, technical robustness, and scalability—while preparing the application for more advanced identity management in the future.



## 9. CI/CD Pipeline

The *Inclusive Trip Planner* project relies on a complete CI/CD pipeline to ensure that every code modification is automatically tested, validated, and production-ready. This automation is based on **GitHub Actions**, **Docker**, and cloud hosting on **Render** for the backend.

### Backend CI: Testing and Containerized Deployment

The backend pipeline is triggered on every push or pull request to the main or develop branches. It performs the following steps:

- Compile the code using Java 21;
- Build a Docker image using `docker build`;
- Launch the full architecture (Spring Boot + PostgreSQL) using `docker compose`;
- Actively wait for API availability using a `curl` loop on `/health`;
- Execute integration tests with `mvn test`;
- Conditionally push the image to Docker Hub if the tests pass.

```
1 for i in {1..10}; do
2   if curl --fail http://localhost:8080/health; then exit 0; fi
3   sleep 3
4 done
5 exit 1
```

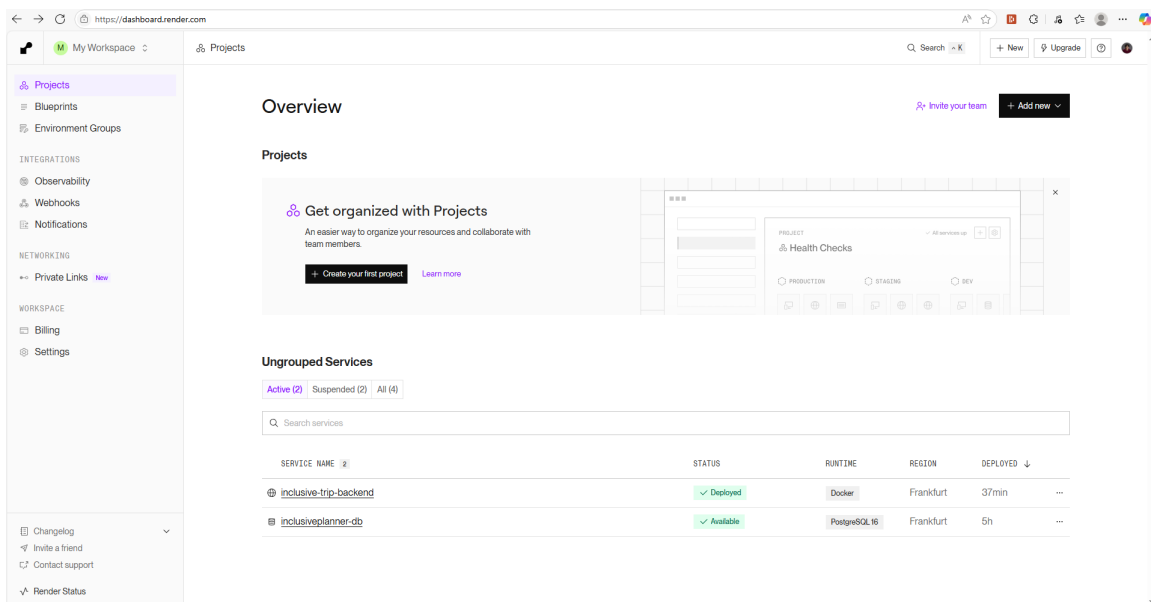


Figure 6: Automated backend deployment to Render from Docker Hub

## Frontend CI: Compilation, Unit Testing, and Code Coverage

The mobile frontend, developed with **Jetpack Compose**, has its own CI pipeline defined in a separate GitHub repository. This pipeline runs on every `push` or `pull request` to `main` or `develop`, and consists of three independent jobs executed in parallel:

- **Build APK:** Compiles the Android project using `./gradlew assembleDebug`, in an environment set up with JDK 17 and Android SDK. This ensures the application can be built at any time.
- **Unit Tests:** Runs Kotlin unit tests via `testDebugUnitTest`, using *Mock APIs* and *coroutine test dispatchers* to simulate API calls. Artifacts are archived for inspection.
- **Lint and Coverage:** Executes `lintDebug` to detect style or logic errors, followed by code coverage report generation using **JaCoCo** (`:app:jacocoTestReport`). Results are exported in HTML format.

```
1 - name: Run Lint
2   run: ./gradlew lintDebug
3
4 - name: Run Code Coverage
5   run: ./gradlew :app:jacocoTestReport
```

This separation into independent tasks makes it easier to isolate errors and ensures fast validation, even with frequent changes. The generated artifacts (test results, lint reports, coverage) are then accessible directly from GitHub for review.

The Android app is not yet deployed automatically, but this pipeline ensures continuous quality for the mobile codebase. Eventually, a publishing step to an app store or internal distribution channel is planned.

## 10. Constraints and Limitations

The development of *Inclusive Trip Planner* raised several challenges, both technical and organizational. As the sole developer responsible for the entire stack, strategic decisions had to be made regarding functional scope, architecture, and time allocation—while maintaining consistency and reliability across the codebase.

One of the major challenges was ensuring end-to-end consistency across the system’s layers: data modeling, backend business logic, frontend state management, and user interface reactivity. This required rigorous planning and strict Git discipline to prevent regressions and untested changes.

Another complexity lay in balancing accessibility with general usability. Treating accessibility as a central requirement meant expanding the data model, designing new user journeys (such as preference collection during onboarding), and handling edge cases rarely considered in a typical MVP. This increased both the functional and architectural complexity of most screens.

Testing also revealed trade-offs. Initial development proceeded without an automated pipeline, relying solely on manual validations and local Git workflows. While this enabled fast prototyping, it slowed down feedback loops and introduced friction. The gradual addition of CI/CD pipelines has significantly improved test automation and deployment readiness.

**Known limitations** of the current MVP:

- Suggested itineraries are predefined and do not dynamically adapt to the user’s history or geolocation.
- Accessibility filters rely on a fixed metadata schema, without real-time data or community contributions.
- Error and edge-case handling is functional but not yet fully optimized for large-scale production environments.

Despite these limitations, the MVP fully meets its core objectives: delivering an accessible, secure, and well-structured travel planning experience on mobile. Its modular architecture and comprehensive test coverage provide a strong foundation for future evolution toward greater personalization, scalability, and integration with third-party mobility services.

## 11. Conclusion and Future Perspectives

The *Inclusive Trip Planner* project resulted in a robust and accessible MVP specifically designed for individuals with mobility or accessibility needs. It combines a refined user experience, a curated selection of adapted itineraries, and a modern full-stack architecture to deliver a functional solution focused on user needs. The application allows users to discover, save, and track personalized routes tailored to their individual preferences and constraints.

From a technical perspective, the project demonstrates full command of the software stack: reactive frontend development using **Jetpack Compose**, a secure and scalable backend built with **Spring Boot**, integration tests, and a CI pipeline to ensure long-term maintainability. Core architectural principles—such as separation of concerns, API design via DTOs, and test-driven iterations—were consistently upheld throughout the development process.

Accessibility was not treated as an afterthought but was integrated from the very beginning, both in data modeling and interface design. This approach laid a foundation that combines strong technical performance with genuine inclusivity in the user experience.

**Planned future improvements** focus on the following areas:

- **Dynamic itinerary generation:** shift from static routes to intelligent composition based on user data in real time, assisted by rules or AI.
- **Recommendation systems:** introduce adaptive logic capable of suggesting itineraries based on user behavior, accessibility profile, or feedback history.
- **Accessibility metadata enrichment:** incorporate real-time updates, community contributions, and external API data to improve route annotations.
- **Adaptation to the healthcare sector:** repurpose the platform to assist patients with reduced mobility in organizing accessible trips to clinics or hospitals, in partnership with healthcare professionals.

The current MVP lays a strong modular foundation, capable of evolving both in terms of application domain and technical scale. Whether as a personal travel assistant or a tool serving broader inclusion, *Inclusive Trip Planner* stands as a meaningful solution at the intersection of technical excellence and human-centered design.

*I would like to thank Ynov for their attention to this project and express my full motivation to continue this work as part of the Master's program, to further deepen my skills and lead this initiative to new horizons.*

# Appendix

The appendix below provides a concise overview of the main technical tools used throughout the project, along with the rationale behind their selection.

- **Kotlin + Jetpack Compose** – Used for frontend development due to its native Android support, declarative UI composition, and perfect compatibility with a ViewModel-driven architecture. Compose’s reactive model, combined with built-in *coroutines*, makes it an ideal choice for managing asynchronous state in a mobile application designed from the ground up for this platform.
- **Spring Boot (Java)** – Chosen for backend development thanks to its rich ecosystem, ease of building REST APIs, and native support for layered architectures. Its integration with **Spring Security** and **JPA** makes it a robust solution for building secure and scalable applications.
- **PostgreSQL** – Selected as the relational database for its stability, indexing performance, and compatibility with **Liquibase**. Its native support for UUIDs and JSON structures was particularly useful in modeling certain entities and user preferences.
- **Docker** – Used to containerize backend services as well as the PostgreSQL database, ensuring consistent environments across development, testing, and production. **Docker Compose** made it easy to orchestrate integration tests and local simulations of the full architecture.
- **GitHub Actions** – Set up as the foundation for continuous integration (CI) for both frontend and backend. It provides reliable automation for testing, linting, code coverage, Docker image generation, and deployment workflows—with systematic validation on every *push* before merging.
- **Liquibase** – Database migration tool ensuring versioned and reversible schema evolution. It guarantees that every database state is traceable, reproducible, and validated before the application starts, which is essential for stability in testing and production environments.
- **Render** – Platform selected for hosting the backend as a Docker container. It offers a simple interface for deployment and secret management, along with HTTPS hosting, build logs, and health checks. It provides a lightweight alternative to more complex CI/CD platforms.
- **Retrofit** – Library used on the frontend to manage network requests in a structured and type-safe way. It automatically converts backend JSON responses into Kotlin objects and integrates natively with *coroutines* and **ViewModels**, facilitating smooth communication between the UI and the backend.