POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Software Engineering II

Design Document

Project: Best Bike Paths

Authors: Ianosel Bianca Roberta, Gholami Vajihe, Errigo Simone
Version: 1.0
Date: 23.12.2025
Project Link: Errigo-Gholami-Ianosel (github.com)

Contents

1 | Introduction

1.1. Purpose

1.2. Scope

1.3. Definitions, Acronyms, Abbreviations

1.3.1. Definitions

- **Bike Path:** A route, defined by collected data, where a proper bike track exists or where road conditions are generally compatible with cycling safety and speed. Path quality is determined by its aggregated status.

1.3.2. Acronyms

- **BBP:** Best Bike Paths.
- **DD:** Design Document.
- **CRUD:** Create, Read, Update, Delete.
- **REST:** Representational State Transfer.
- **HTTP:** HyperText Transfer Protocol.
- **JSON:** JavaScript Object Notation.
- **DB:** Database.
- **DBMS:** DataBase Management System.
- **RASD:** Requirement Analysis and Specification Document.
- **GPS:** Global Positioning System.
- **API:** Application Programming Interface.

- **UI:** User Interface.

1.3.3. Abbreviations

- **[UCn]** -The n-th use case.

1.4. Revision history

- Version 1.0 (23 December 2025);

1.5. Reference Documents

The preparation of this document was supported by the following reference materials:

- IEEE Standard for Software Requirement Specifications [1];
- Assignment specification for the RASD and DD of the Software Engineering II course, held by professors Matteo Rossi, Elisabetta Di Nitto, and Matteo Camilli at the Politecnico di Milano, A.Y. 2025/2026 [2];
- Slides of the Software Engineering 2 course available on WeBeep [3];

1.6. Document Structure

Mainly the current document is divided into 7 chapters, which are:

1. **Introduction**:
2. **Architectural Design**:
3. **User Interface Design**:
4. **Requirements Traceability**:
5. **Implementation, Integration and Test Plan**:
6. **Effort Spent**:
7. **References**:

2 | Architectural Design

2.1. Overview: High-level components and their interactions

2.2. Architectural Overview

The architecture of the Best Bike Paths (BBP) system follows a classic **three-tier structure**, separating the software into a presentation layer, an app layer, and a data layer. This architectural style ensures a clear division of responsibilities and simplifies the evolution of the system over time. Since BBP is conceived as a mobile-centric platform, the presentation tier is implemented entirely through the BBP mobile app, which communicates with the backend via RESTful APIs over HTTPS.

Presentation Layer

The presentation layer consists solely of the **BBP mobile app**, which serves as the primary interface between users and the system. This layer is responsible for:

- rendering the user interface and handling user interactions;
- acquiring device-level data (GPS, accelerometer, gyroscope) during trips;
- displaying bike paths, trip summaries, statistics, and reports;
- invoking backend functionalities through HTTP requests.

The mobile app is intentionally designed as a **thin client**. All domain logic, decision processes, ranking operations, and aggregation of path information are delegated to the app layer. The app interacts with device-level subsystems such as the GPS module and external sensors (when available), but these elements are not part of the backend architecture. The mobile app also uses the secure storage facilities provided by the operating system (iOS Keychain / Android Keystore) to safely store authentication tokens and other sensitive data.

Application Layer

The app layer embodies the **core business logic** of BBP. It is implemented as a modular backend composed of independent yet cooperating **components**, each encapsulating a well–defined responsibility. These components are logically independent in terms of responsibilities and interfaces, but they are part of a single backend app. The main functional components include:

- **User Module**: contains the **AuthManager** and the **UserManager**, responsible for authentication, credential verification, and management of user profiles.
- **Trip Module**: contains the **TripManager**, which handles the lifecycle of a cycling trip and produces trip summaries enriched with contextual weather data.
- **Path Module**: contains the **PathManager**, responsible for maintaining bike-path data, computing routes, and ranking candidate paths according to their condition and effectiveness..
- **Report Module**: contains the **ReportManager**, responsible for storing and aggregating reports, managing confirmations, and updating path-condition indicators.
- **Statistics Module**: contains the **StatsManager**, which computes and stores user statistics and per-trip metrics.

The app layer also includes the **WeatherManager**, which interacts with an external weather service to retrieve meteorological data. All modules are accessed through the **API Gateway**, which exposes a set of RESTful sub-APIs and routes incoming requests to the appropriate Manager.

Data Layer

The data layer consists of a **relational DBMS** storing all persistent information relevant to the system's domain, including:

- user profiles and authentication credentials;
- trip records and associated GPS data;
- bike path segments and their aggregated conditions;
- reports, confirmations, and metadata about obstacles;
- computed statistics and weather snapshot.

All interactions with the DBMS are mediated by a single **QueryManager**, which centralises data-access operations and offers a uniform interface for executing queries. This design keeps persistence concerns separated from the app logic and reduces duplication across components.

2.3. Component View

This section describes the main software components that constitute the BBP backend and their responsibilities. As required by the three-tier architecture adopted by the system, the backend is structured into a set of independent yet cooperating modules, each exposing well-defined interfaces and encapsulating a cohesive subset of the app logic.

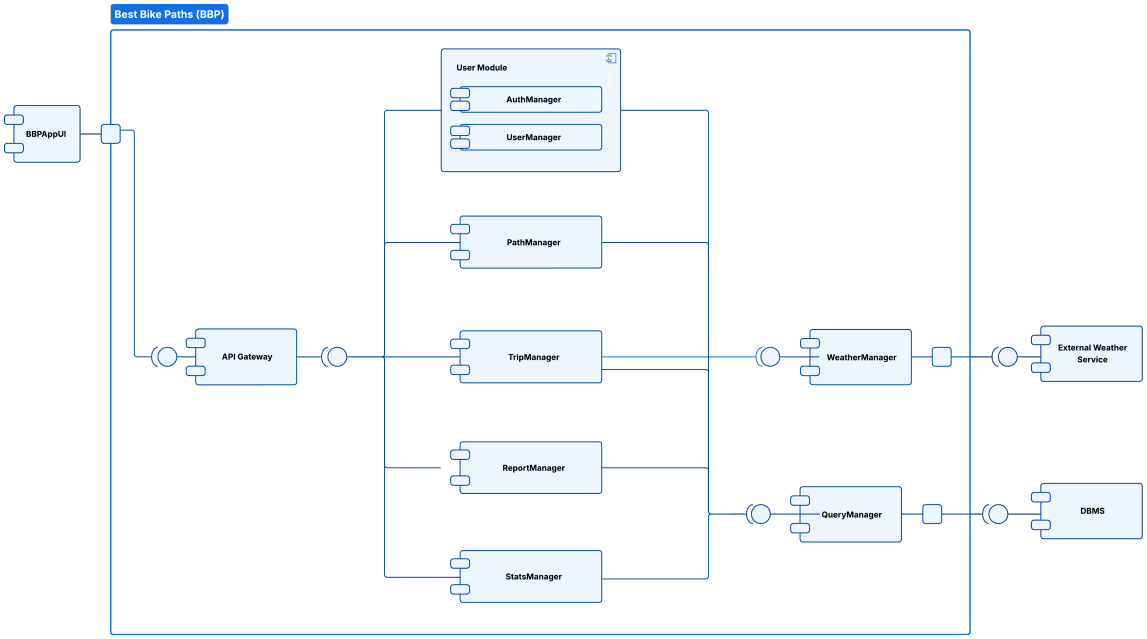

Figure 2.1: Component View Diagram

API Gateway

The **API Gateway** acts as the entry point for all interactions between the BBP mobile app and the backend. It is responsible for routing incoming requests to the appropriate internal services, enforcing authentication and authorization requirements, validating inputs, and translating domain errors into HTTP responses.

The API Gateway exposes the following logical sub-APIs:

- **AuthAPI**: endpoints for token generation and validation.
- **UserAPI**: endpoints for registration, login, token refresh, and profile retrieval.

- **TripAPI**: endpoints for starting, updating, and stopping a trip.
- **PathAPI**: endpoints for route computation and retrieval of path metadata.
- **ReportAPI**: endpoints for creating and confirming obstacle reports.
- **StatisticsAPI**: endpoints for retrieving per-trip and aggregated statistics.

User Module

The **User Module** groups two Managers:

- **AuthManager**, responsible for authentication and token issuance.
- **UserManager**, responsible for registration, profile updates, and credential-related operations.

Both Managers use the **QueryManager** for data retrieval and persistence.

Trip Manager

The **TripManager** manages the entire lifecycle of a cycling session. It receives GPS data from the mobile app, records the user's trajectory, and stores trip metadata such as timestamps, duration, distance, and average speed. When a trip is completed, the component generates its summary and associates it with relevant environmental data retrieved from the Weather Manager. The component relies on the **QueryManager** for storage and communicates with the Weather Manager to obtain contextual weather information.

Path Manager

The **PathManager** is responsible for retrieving graph data from the database, computing optimal routes between two locations, and ranking alternative routes according to their quality and reported conditions. This service exposes the routing logic to the API Gateway and interacts with the **QueryManager** to retrieve and update path and report information needed for route computation.

Reports Manager

The **Reports Manager** handles obstacle reports submitted by users or automatically detected during trips. It stores and aggregates reports, manages confirmation and rejection flows, updates path-quality indicators, and exposes relevant data to the mobile app. This component interacts with the **QueryManager** for persistence and with the Routing

& Path Manager when updated conditions affect route evaluation.

Statistics Manager

The **Statistics Manager** computes and retrieves aggregated cycling statistics. It retrieves historical data through the **QueryManager**.

Weather Manager

The **Weather Manager** interacts with the external weather API to obtain meteorological information. It provides a weather snapshot associated with trip start and end points. It stores the weather snapshot using the **QueryManager**.

QueryManager

The **QueryManager** is the data-access component of the backend. It acts as the single entry point for interacting with the relational DBMS and provides a set of methods to retrieve, insert, update, and delete domain data. Centralising data access in one component simplifies consistency checks, reduces duplicated logic, and keeps the domain layer independent of database details.

2.4. Deployment View

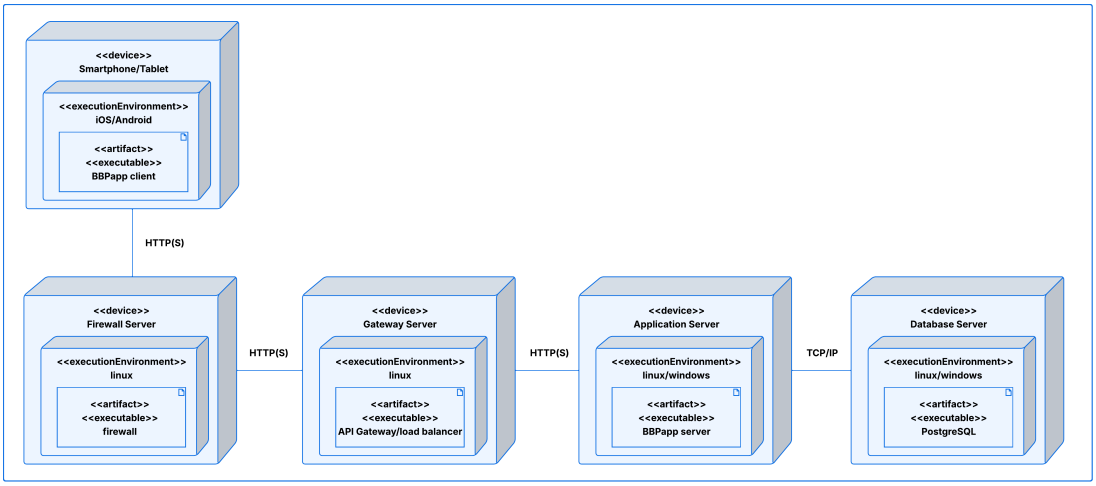

Figure 2.2: Deployment View of the BBP System

The deployment view describes the hardware and software infrastructure supporting the *BBP* system. Each tier is executed on dedicated hardware nodes and communicates with

the others using secure protocols.

- **Presentation Tier:** this tier includes all devices through which end users interact with the BBP system. The primary clients are smartphones or tablets running iOS or Android, where the BBP mobile application is installed. These devices communicate with the backend exclusively via HTTPS, ensuring confidentiality and data integrity. Although the mobile app represents the main access point, any device equipped with a modern web browser and a stable internet connection could technically interact with the system, since the backend exposes standard RESTful endpoints. No application logic is executed at this level, the devices simply capture user input, display results, and forward authenticated HTTP requests toward the Application Tier.
- **Application Tier:** this tier is responsible for handling incoming traffic, enforcing security, applying routing and load balancing policies, and executing the core business logic of the system. All external requests first pass through a Linux-based server configured as a Web Application Firewall using *ModSecurity*. ModSecurity blocks malicious traffic such as SQL injection attempts, cross-site scripting payloads, and abnormal request patterns, while supporting anomaly detection. Validated HTTPS traffic is then forwarded to the gateway node running *Traefik*, which acts as the system's reverse proxy and load balancer. Traefik terminates HTTPS connections, exposes a single public entrypoint for clients, and distributes incoming requests across multiple backend replicas using load balancing strategies. Additional middleware functionalities (request logging or rate limiting) can be applied as needed at this level. The backend application itself is executed on one or more stateless Application Servers, each running the BBP RESTful backend. Since authentication tokens are included in each request header, no server-side session state is maintained, enabling horizontal scaling and dynamic replica management. Communication between Traefik and the backend replicas is confined to a protected internal network, reducing the system's exposure to external threats.
- **Data Tier:** the Data Tier consists of a dedicated server running a PostgreSQL database instance, which stores all persistent system data such as user information, paths, trips, and analytics. Backend servers interact with the database using standard PostgreSQL drivers over TCP/IP within an isolated internal network segment. Centralizing the database simplifies backup strategies, consistency enforcement, and maintenance operations, while still allowing for potential future extensions such as replication or clustering without altering the upper tiers of the system.

2.5. Runtime View

The Runtime View describes how the components of the BBP system collaborate to realise the behaviour specified in the functional requirements. While the Component View focuses on the static organisation of the backend (**API Gateway**, **Managers**, **QueryManager**) and their responsibilities, the Runtime View illustrates how these components interact dynamically during the execution of the main use cases.

All diagrams follow the modular structure of the backend: the mobile client invokes the **API Gateway**, which routes each request to the appropriate **Manager**. Persistence operations are centralised in the **QueryManager**, whereas weather-related data requests involve the **WeatherManager** and its external API.

The following pages report the sequence diagrams for all core use cases, from user authentication to trip management, path exploration, reporting, statistics retrieval, and profile updates. Together, these diagrams provide a comprehensive understanding of how the BBP system behaves during execution and how responsibilities are distributed among its components.

[UC1] - User Registration

A new user wants to register into the BBP system. The process begins on the mobile app, where the guest user opens the registration page, and fills in the required details: email, password, and personal information. A first **local validation** step checks for malformed inputs before contacting the **backend**.

If the data is valid, the mobile app sends a registration request to the **API Gateway**, which forwards it to the **AuthManager**. This component checks that the email is not already in use by querying the database through the **QueryManager**. If the email already exists, the system returns an error and the mobile app notifies the user accordingly.

If the email does not exist, the **AuthManager** inserts the new user into the database and generates an **authentication token**. The token is returned to the mobile app, which stores it securely using the device's **secure storage** mechanism. The flow concludes with a success message being shown to the user.

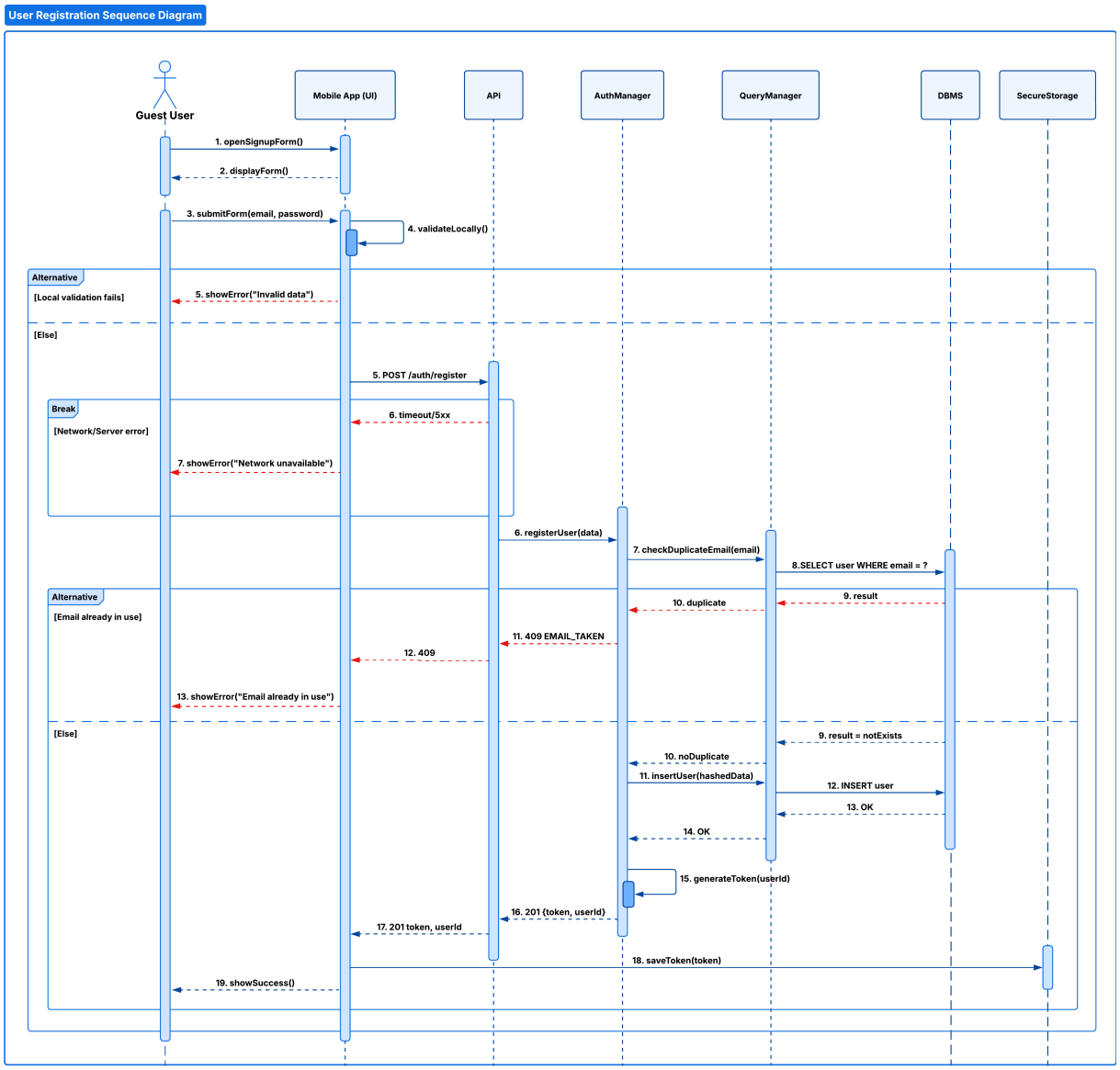

Figure 2.3: User Registration Sequence Diagram

[UC2] - User Log In

A guest user wants to authenticate and obtain access to the system. The process starts when the user opens the login form and submits credentials through the mobile app. After a **local validation**, the app sends an HTTP request to the login endpoint exposed by the **API Gateway**.

The **API Gateway** forwards the request to the **AuthManager**, which first checks whether the provided email exists by querying the **DBMS** through the **QueryManager**. If the email is not found, the backend returns a **404 Not Found** error, which the mobile app displays to the user. If the user exists, the **AuthManager** verifies the submitted password. Invalid credentials lead to a **401 Unauthorized** response and the corresponding error message on the client.

When the credentials are correct, the **AuthManager** generates an **authentication token** and returns it to the mobile app, which stores it securely using the device's **secure storage** facility. The user is then successfully logged in and the app proceeds to show the appropriate authenticated UI.

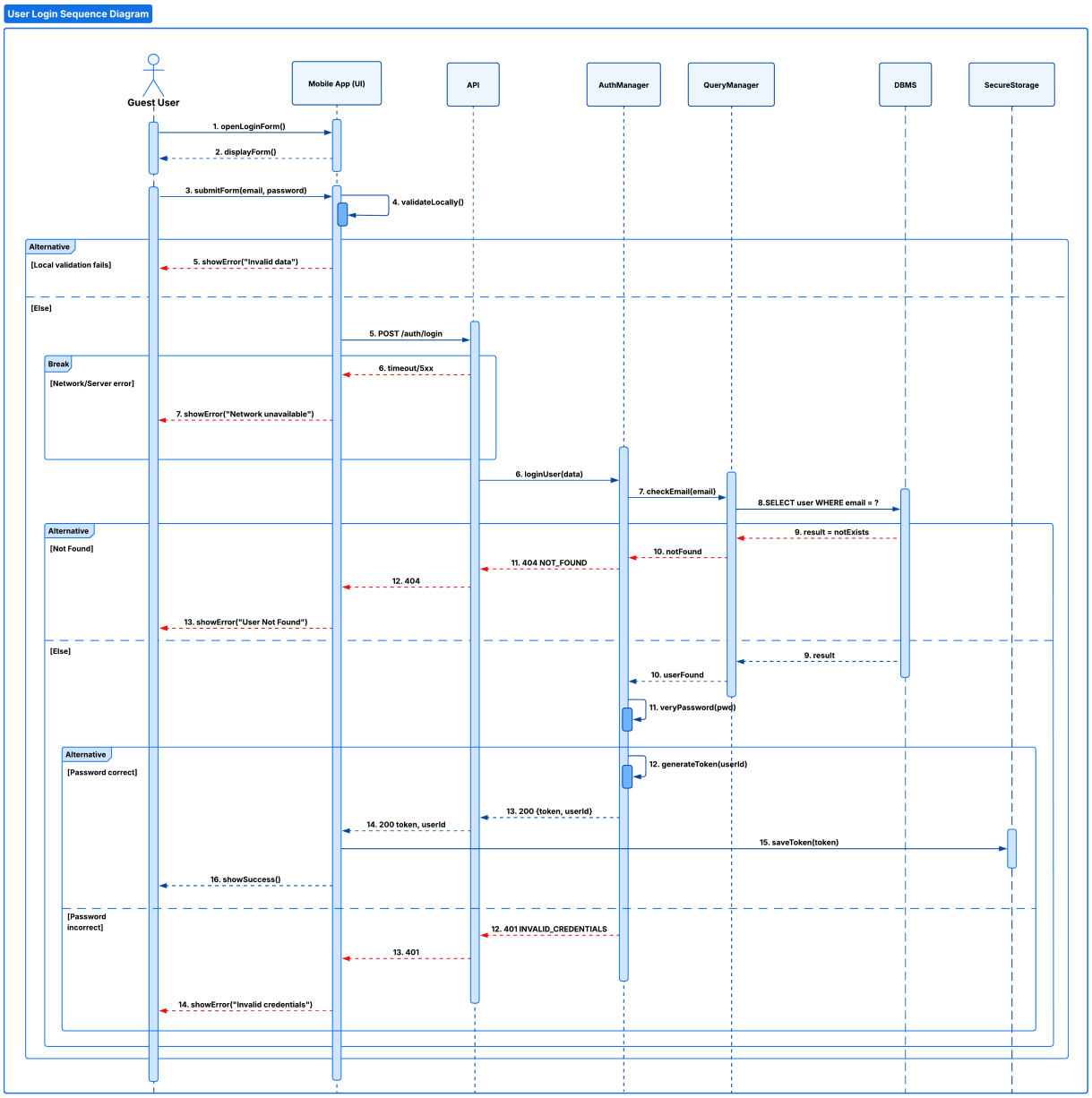

Figure 2.4: User Log In Sequence Diagram

[UC3] - User Log Out

When a logged-in user initiates the logout operation from the mobile app, the client first clears the locally stored authentication token from the **secure storage**. Afterward, the mobile app sends an HTTP request to the backend.

The **API Gateway** forwards the request to the **AuthManager**, which handles the logout process by deleting the corresponding **refresh token** through the **QueryManager**. The QueryManager executes a **DELETE** operation on the database to invalidate the stored refresh token.

If the operation succeeds, the server returns a **204 NO_CONTENT** response, and the app displays a success message to the user. If instead a network or server error occurs, the system interrupts the flow and the mobile app notifies the user with a "Network unavailable" error message.

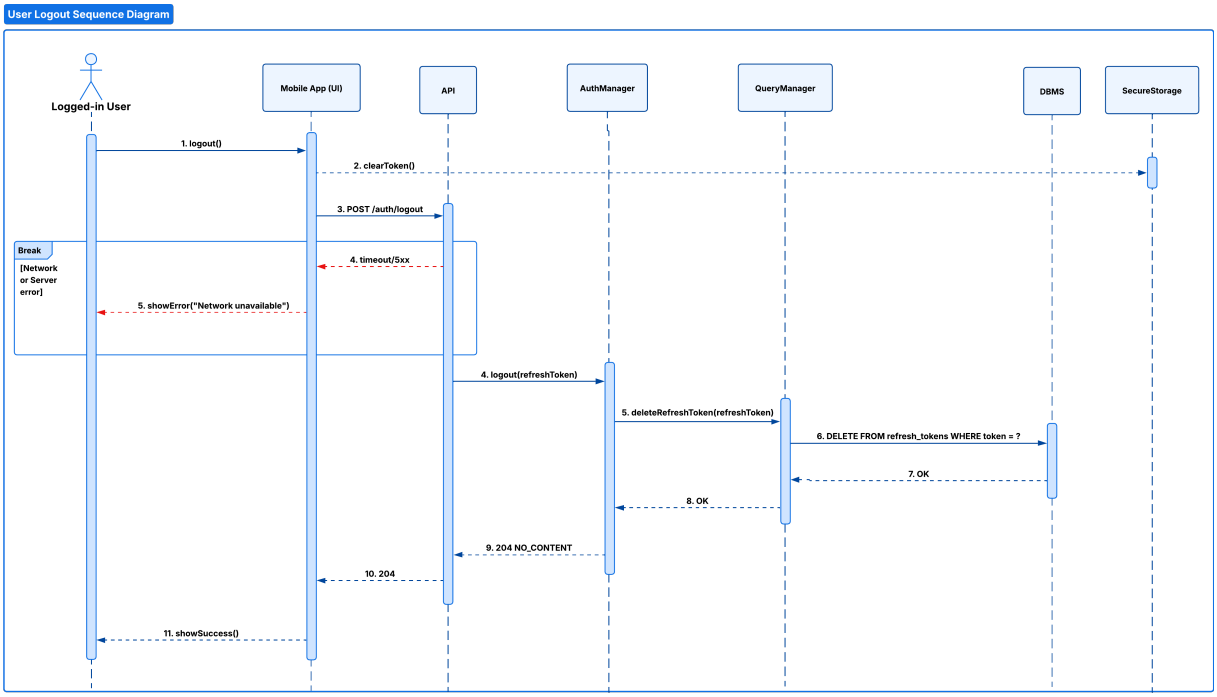

Figure 2.5: User Log Out Sequence Diagram

[UC4] - Search for a Path

A user can search for bike paths between two locations. He enters the start and end points and submits the request. The app performs a preliminary local validation and, if the data is valid, forwards the request to the backend through the **API Gateway**.

The **API Gateway** forwards the request to the **PathManager**, which loads the relevant portion of the path graph from the database using the **QueryManager**. Once the graph data is retrieved, the **PathManager** computes the optimal path(s) according to the requested constraints. If valid routes are found, the API Gateway returns them to the mobile app, which displays the corresponding suggestions.

If no route satisfies the user's constraints, the **PathManager** signals a **NO_ROUTE** condition, which results in a 404 error. In the case of network or server issues, the app notifies the user with an appropriate error message.

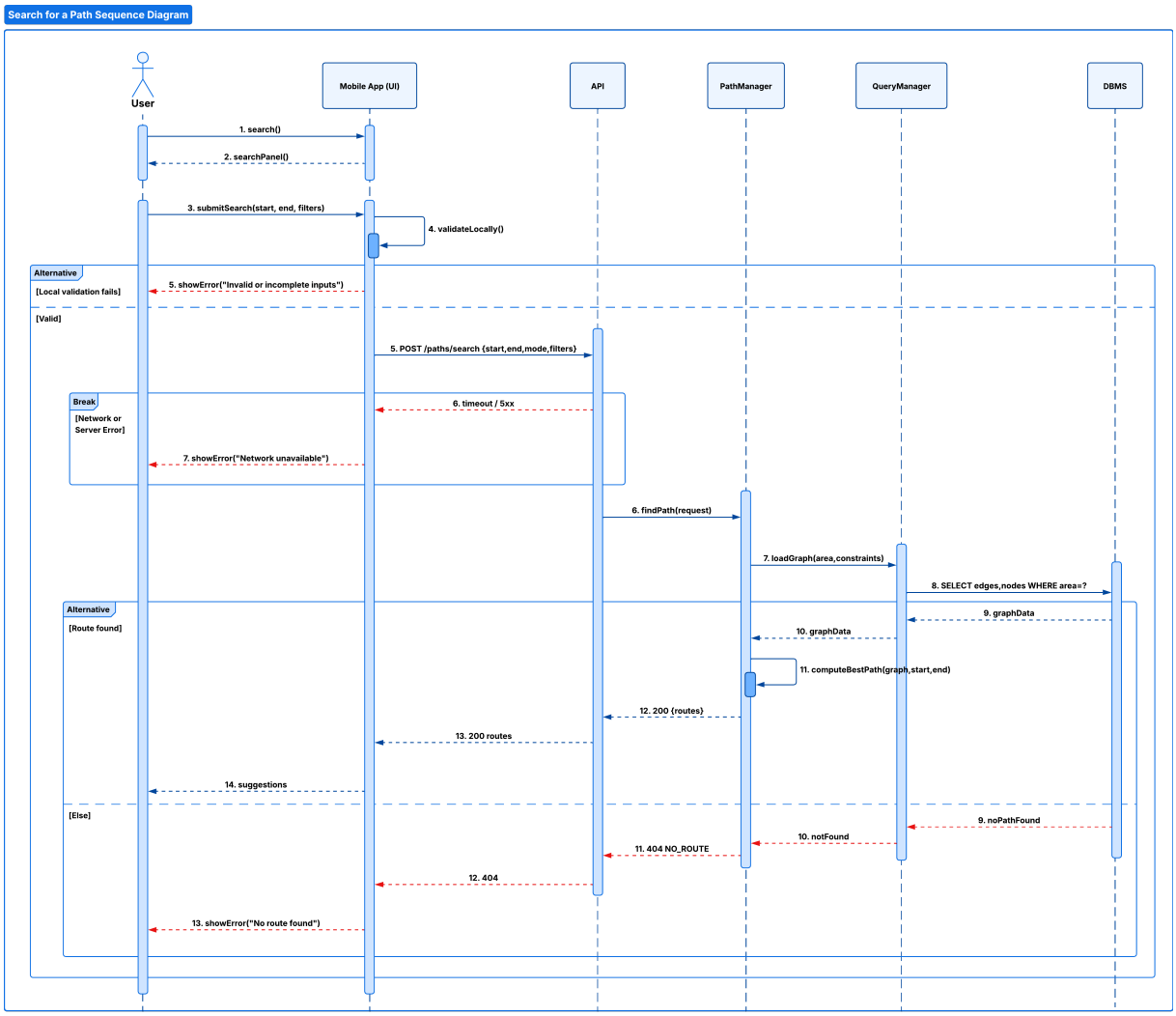

Figure 2.6: Search for a Path Sequence Diagram

[UC5] - Select a Path

The Select a Path sequence diagram illustrates how the system retrieves the details of a path selected by a user. The interaction begins when the user chooses a specific path from the list of suggested routes displayed by the mobile app. The app sends a request to the backend through the **API Gateway**, which forwards it to the **PathManager**. The **PathManager** retrieves the corresponding path information by querying the database through the **QueryManager**. If a matching record is found, the PathManager returns the path details to the **API Gateway**, which sends them back to the mobile app for presentation to the user. If the database does not contain a path with the specified identifier, the **PathManager** signals a **NOT_FOUND** condition, resulting in a **404** response. In that case, the mobile app notifies the user that the selected route is unavailable. As in other interactions, network or server errors trigger an appropriate error message on the client side.

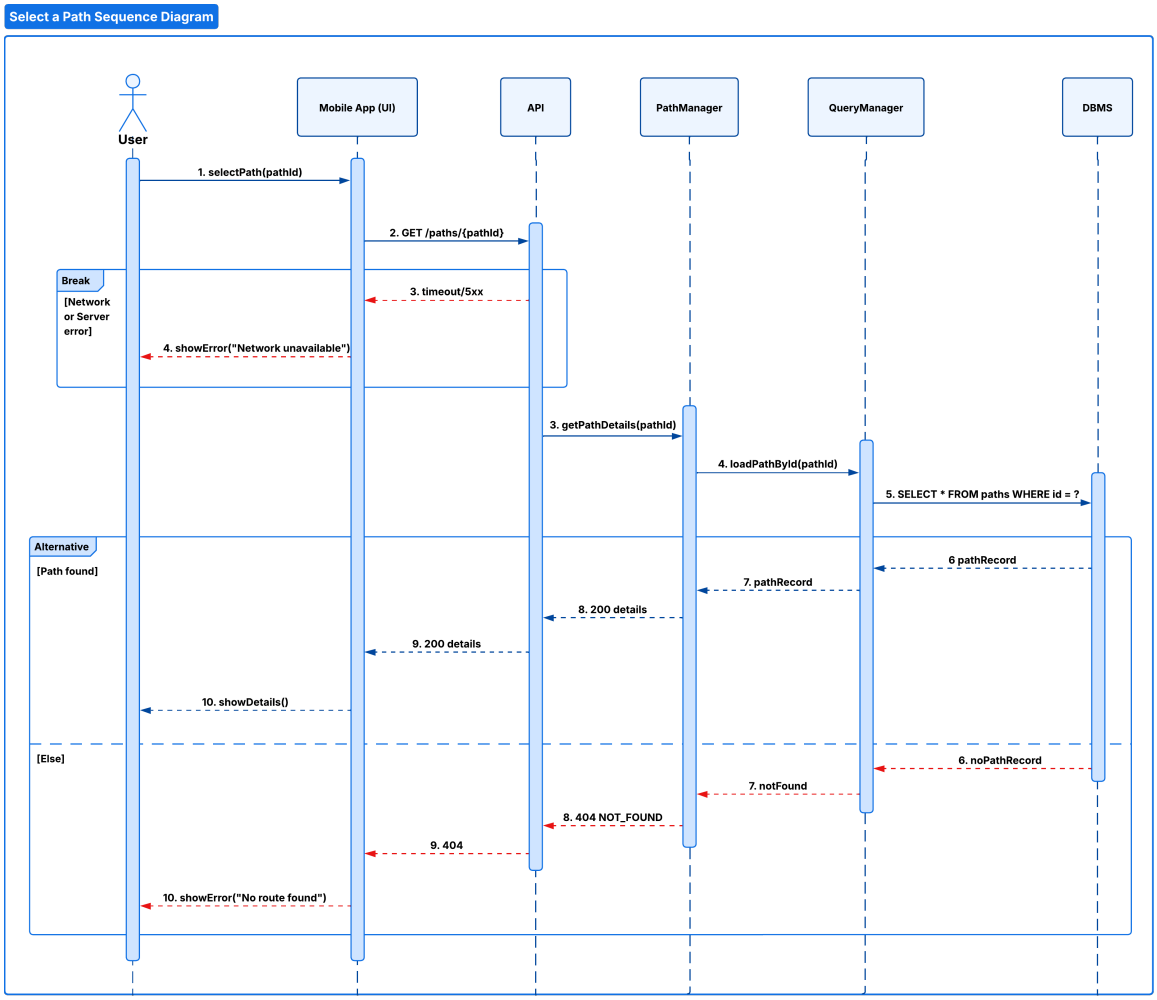

Figure 2.7: Select a Path Sequence Diagram

[UC6] - Create a Path in Manual mode

When a logged-in user wants to create a new path, he will be asked to choose between manual and automatic creation modes. If he opts for creating the path in **manual mode**, he will be presented with a form to fill in the required metadata and segment list.

Before sending the request, the app performs local validation to ensure that all mandatory fields and segments are correctly specified.

If the input is valid, the mobile app submits the creation request to the backend through the **API Gateway**. The gateway forwards the request to the **PathManager**, which is responsible for handling the creation workflow. The PathManager stores the new path by delegating the persistence task to the **QueryManager**, which executes the corresponding **INSERT** operation on the **DBMS**.

Once the database confirms the insertion, the **PathManager** sends the result back to the **API Gateway**. The gateway responds with a **201 Created** status and the new pathId. The mobile app then displays a confirmation message to the user. In the event of network failures or server-side errors, the app notifies the user accordingly.

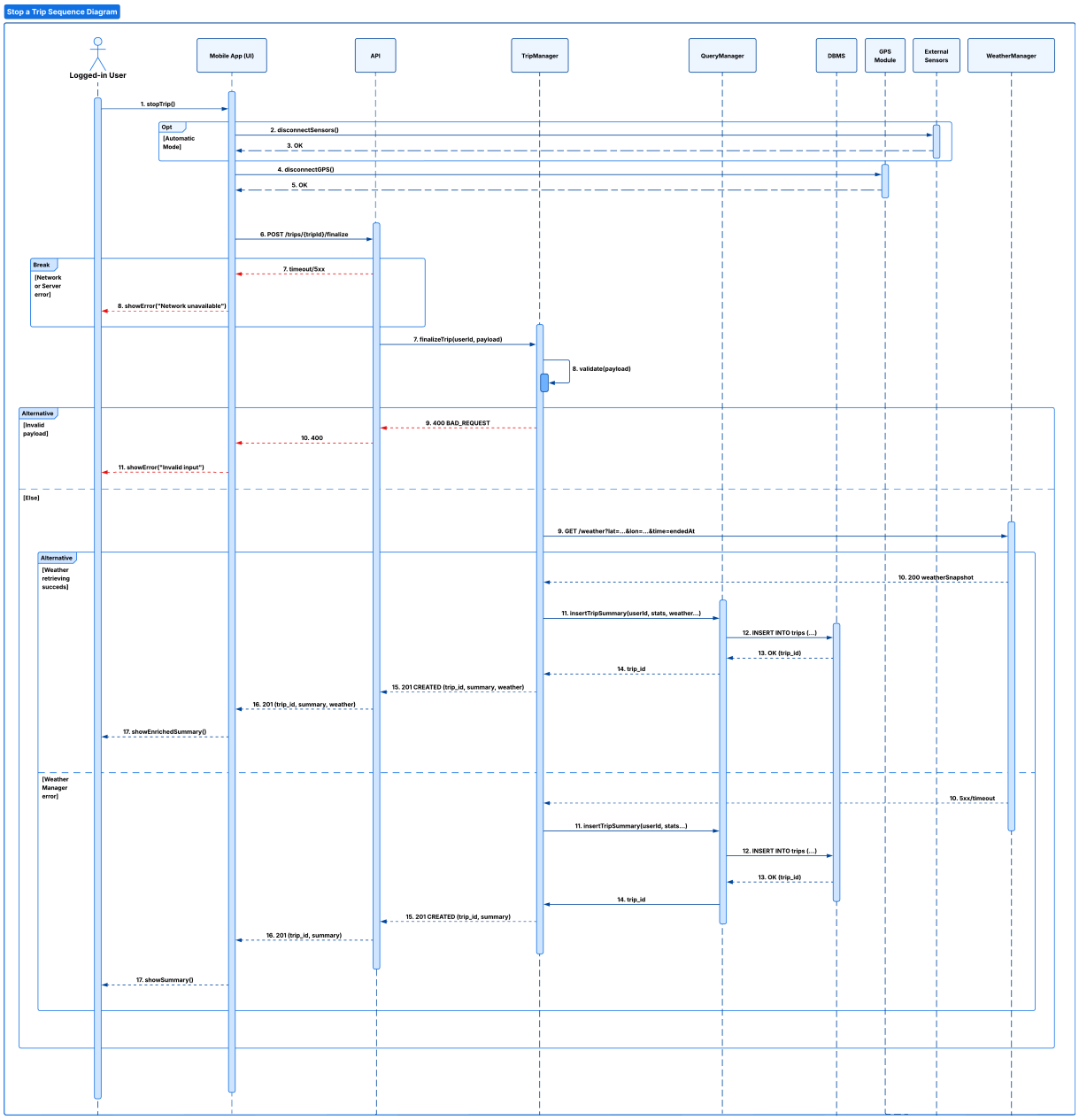

Figure 2.8: Create a Path in Manual Mode Sequence Diagram

[UC7] - Create a Path in Automatic Mode

When a logged-in user chooses to create a new bike path, he first selects the **automatic creation mode** from the mobile app. The app displays a form for entering the required metadata, such as the path name, description, and other relevant details. The app performs a first local validation. If the fields are invalid, the user is immediately notified.

Once the input is valid, the app attempts to activate **GPS tracking**. If activation fails (e.g., permissions or hardware issues), an error is shown. If tracking succeeds, the **GPS module** begins providing continuous location samples (latitude, longitude, speed, timestamp). The app collects these values during the entire movement loop.

If the GPS signal temporarily fails while moving, the app detects the error and interrupts the procedure, informing the user. When the user completes the movement session, GPS tracking is deactivated and the collected samples undergo a final **local validation**; if invalid, the user is notified. When both metadata and sensor samples are valid, the app sends a **POST request** to the backend. If a network timeout or server error occurs, an appropriate message is shown to the user.

The **API** forwards the request to the **PathManager**, which calls the **QueryManager** to store the path in the **DBMS**. The system inserts metadata and user association into the database, and upon successful insertion returns a **201 Created** response containing the new pathId. The mobile app receives the response and displays a success message, indicating that the new automatically generated path has been saved.

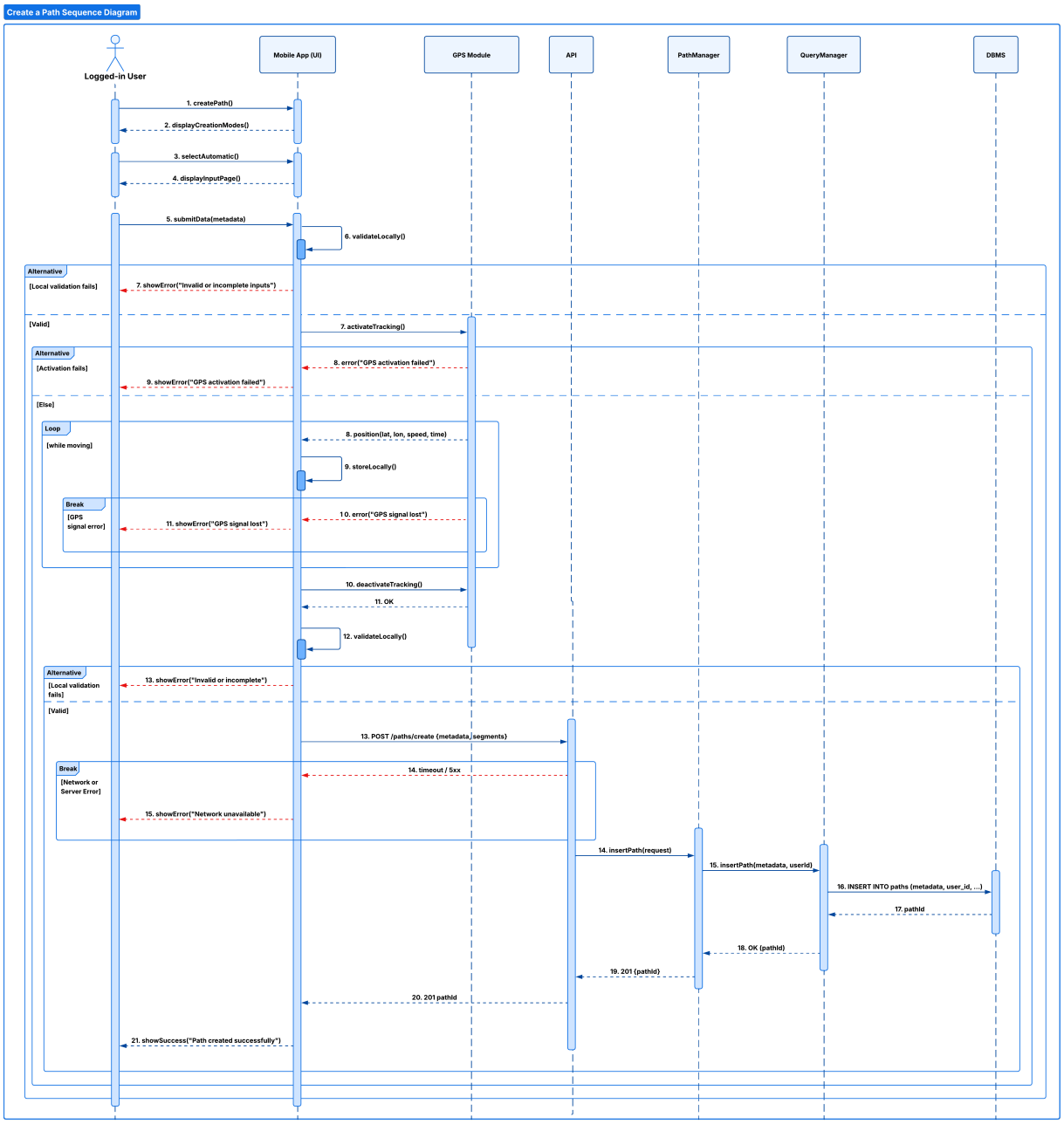

Figure 2.9: Create a Path in Automatic Mode Sequence Diagram

[UC8] - Delete a Path

When a logged-in user wants to delete one of his path, he firstly navigates to the list of his created paths in the mobile app. The app sends a request to the backend to retrieve all paths associated with the user. The **API** forwards this request to the **PathManager**, which retrieves the corresponding records through the **QueryManager** and the **DBMS**. Once the user selects a specific path to delete, the app sends a **DELETE request** to the backend.

The **PathManager** first verifies that the path exists and that the requesting user is its owner. If the ownership check fails, the backend returns a **403 FORBIDDEN** error. If the path does not exist, a **404 NOT_FOUND** response is generated.

When the user is authorised and the path exists, the **PathManager** performs the deletion through the **QueryManager**, which issues the appropriate **SQL DELETE** operation to the **DBMS**. Successful deletion results in a **204** response, upon which the mobile app confirms the removal to the user. Network or server-side failures prompt the mobile app to display a generic error message.

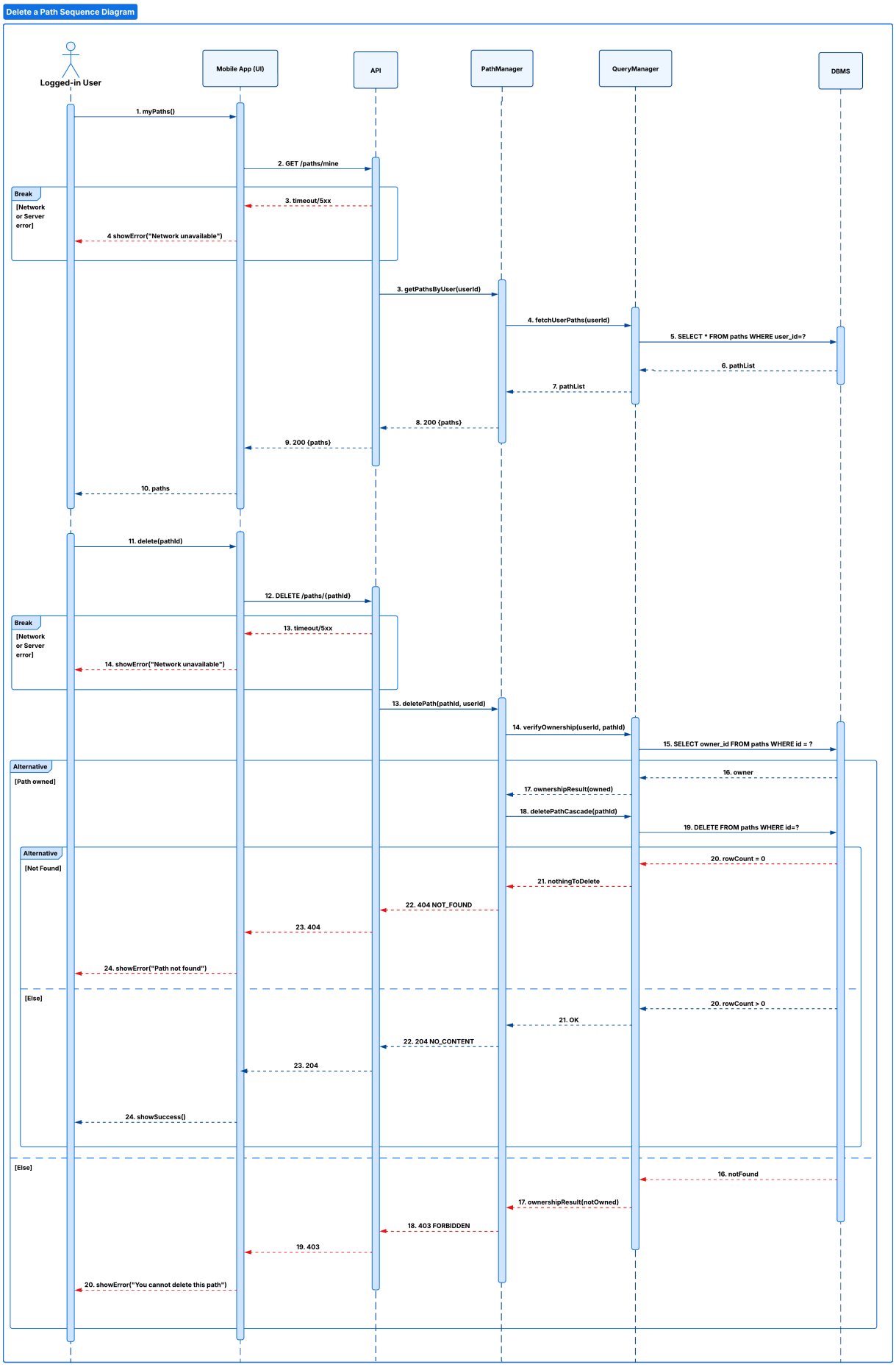

Figure 2.10: Delete a Path Sequence Diagram

[UC9] - Start a Trip as Guest User

When a guest user wants to start a trip using the BBP mobile app, he first selects a path from the available options. The app then attempts to activate **GPS tracking** to monitor the user's movement along the selected path.

If GPS activation fails, the mobile app immediately notifies the user with an error message. Otherwise, once tracking is active, the app continuously receives location updates while the user is moving and refreshes the map accordingly.

If at any point the **GPS module** reports a loss of signal, the app displays an appropriate error message to the user. No backend interaction occurs in this use case, as guest trips are not recorded or stored.

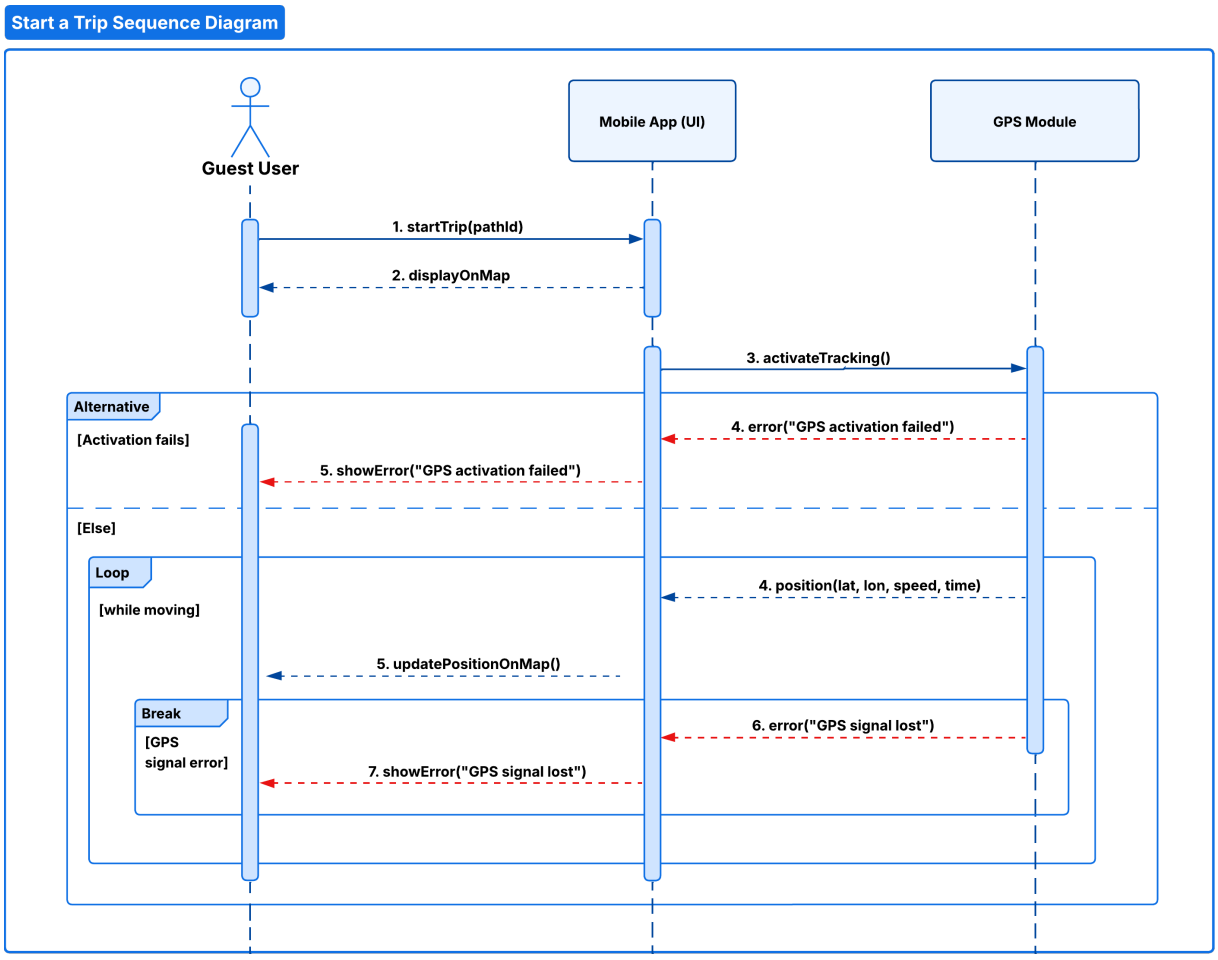

Figure 2.11: Start a Trip as Guest User Sequence Diagram

[UC10] - Start a Trip in Manual Mode as a Logged-in User

In this scenario, a logged-in user starts a trip by selecting a path and enabling **manual tracking**. The interaction begins when the user initiates the trip from the mobile app, which displays the chosen path on the map. The user then selects the **Manual mode**, and then the app activates the **GPS module**. If the GPS fails to activate, the mobile app immediately notifies the user with an error message. Otherwise, GPS tracking begins, and the device periodically emits position updates containing latitude, longitude, speed, and time. The mobile app stores these samples locally and updates the on-screen map in real time During the trip, if the GPS signal is lost at any point, the GPS module reports an error and the mobile app displays a corresponding warning to the user, interrupting the normal flow of position updates.

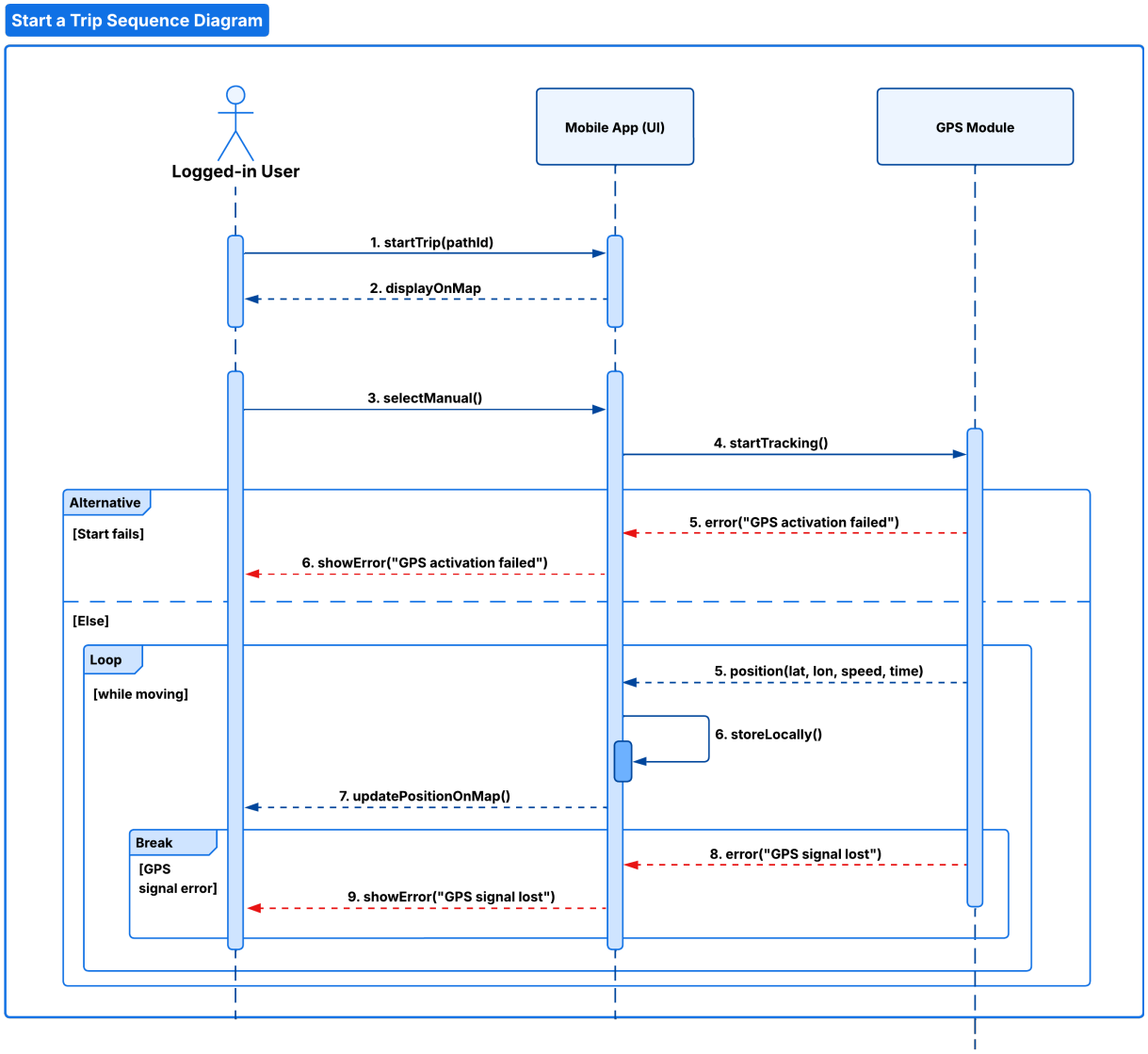

Figure 2.12: Start a Trip in Manual Mode as a Logged-in User Sequence Diagram

[UC11] - Start a Trip in Automatic Mode as a Logged-in user

When a logged-in user selects a path and chooses to start the trip in **automatic mode**, the mobile app first displays the map and then attempts to establish a connection with the **external sensors** required for automatic detection.

If sensor activation fails, the app immediately informs the user with an error message. Otherwise, sensors respond successfully and the app proceeds by enabling GPS tracking. Once tracking is active, the **GPS Module** periodically sends position updates which the app stores locally and reflects on the UI map.

During the trip, if the GPS signal is lost, the system interrupts the loop and displays an appropriate error message. If instead the connection to the external sensors drops during the session, the app stops the automatic process and notifies the user of the failure.

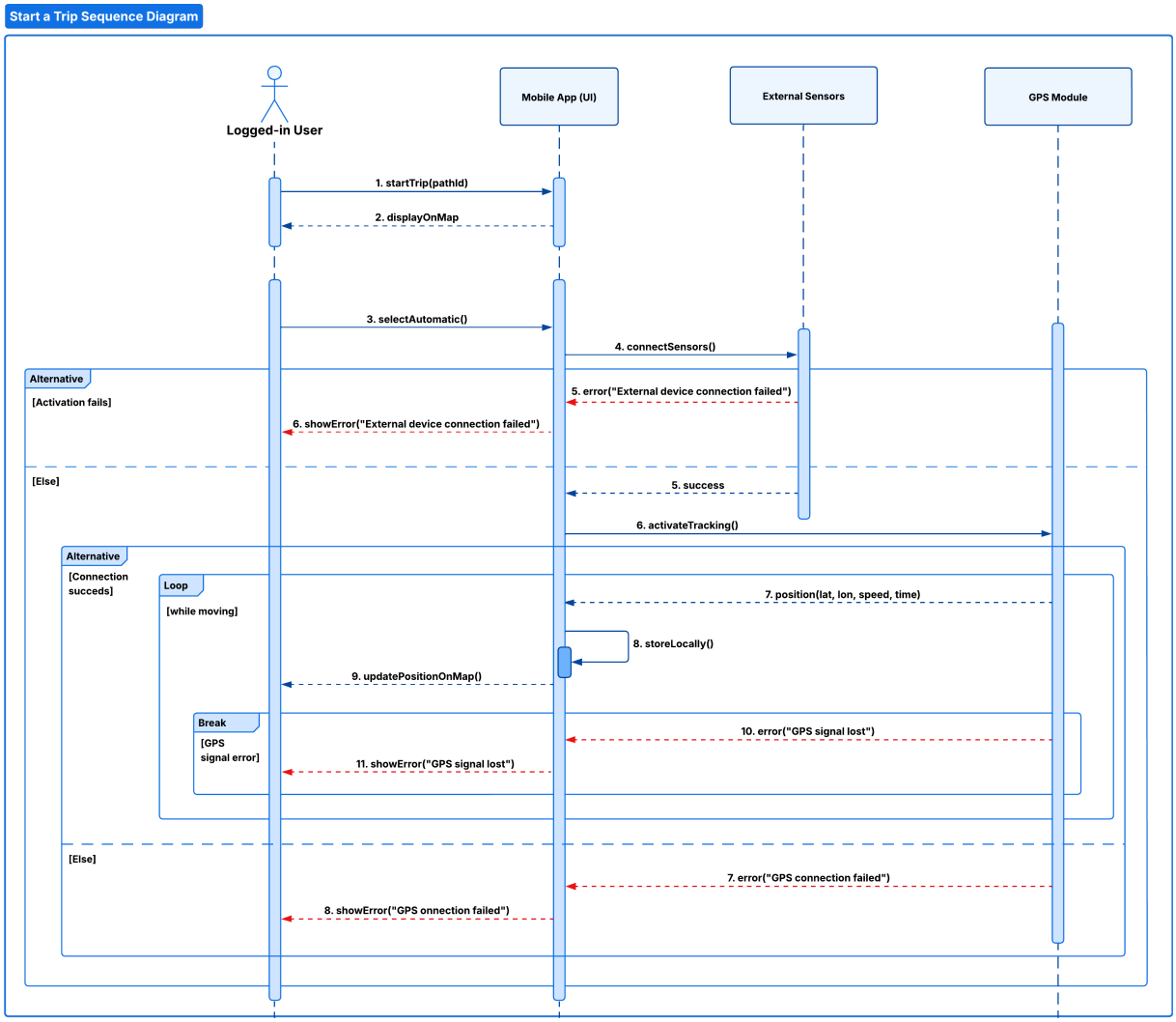

Figure 2.13: Start a Trip in Automatic Mode as a Logged-in User Sequence Diagram

[UC12] - Stop a Trip as Guest User

This sequence diagram describes how a guest user stops an ongoing trip. The interaction is entirely local, as guest users do not store trip data on the backend.

The process begins when the user selects the **Stop Trip** action. The mobile app then requests the **GPS module** to deactivate tracking. Once the GPS confirms that tracking has been successfully stopped, the app terminates the trip visualisation and returns the user to the map or home screen.

No network communication is involved, and no data is persisted, making this use case lightweight and fully handled on the device.

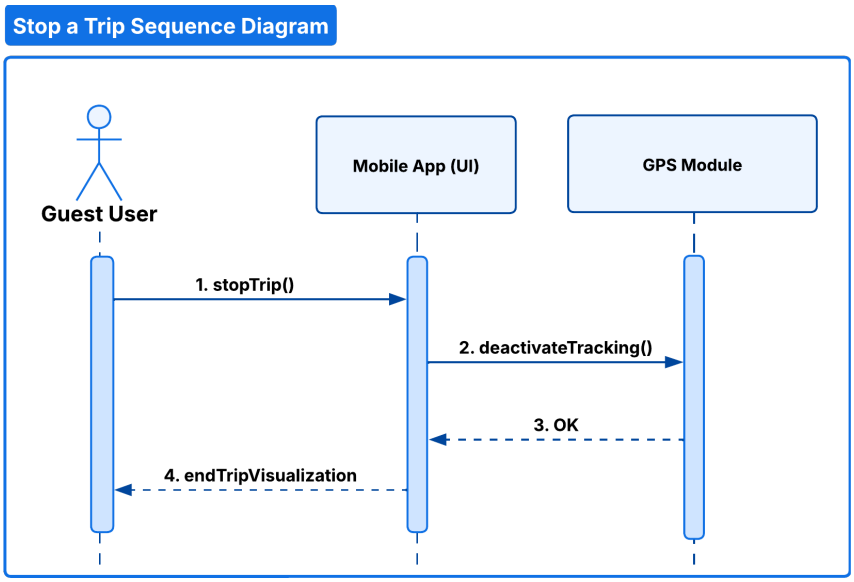

Figure 2.14: Stop a Trip as a Guest User Sequence Diagram

[UC13] - Stop a Trip as a Logged-in User

When the user wants to stop an ongoing trip, he selects the Stop Trip action from the mobile app. Upon this action, the mobile app terminates any active data acquisition (GPS tracking and, if the selected mode is Automatic, external sensor streams) and sends a stop-trip request to the backend. The **TripManager** validates the request and retrieves the corresponding trip record through the **QueryManager**. If the request is invalid, for example, if the trip does not exist or is already closed, the system returns an appropriate error.

If validation succeeds, the **TripManager** contacts the **WeatherManager** to obtain contextual weather information. This step is best-effort: if the external weather service is reachable and returns valid data, the weather snapshot is included in the summary; otherwise, the summary is generated without weather information. The **TripManager** then computes the final trip summary, including distance, duration, speed metrics, sensor-derived data (when available), and the associated weather snapshot. The summary is then saved through the **QueryManager**.

The backend responds with a **201 Created** status and the complete summary. The mobile app then displays the result to the user. Network failures, invalid identifiers, or missing trip records trigger the corresponding alternative flows.

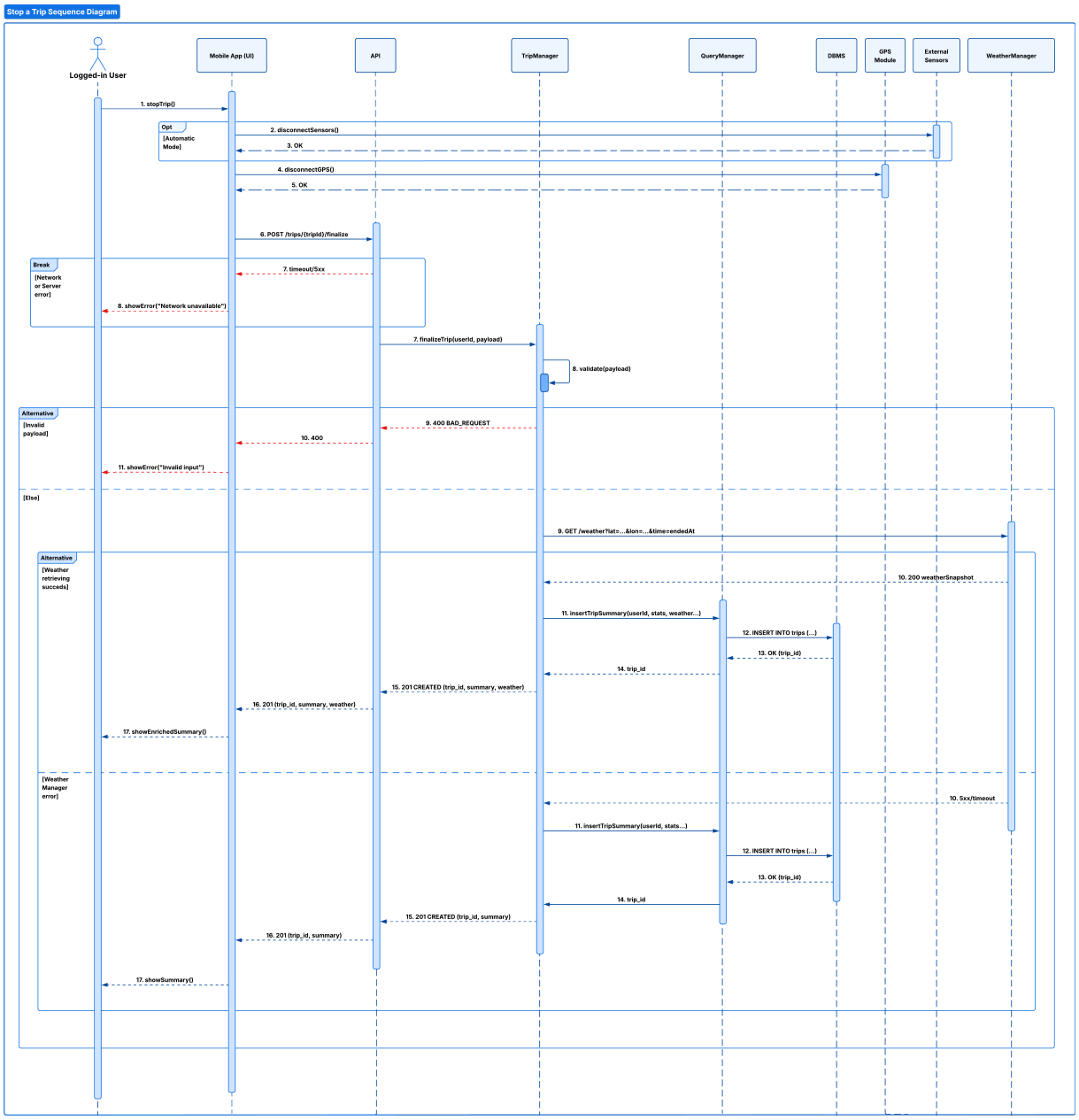

Figure 2.15: Stop a Trip as a Logged-in User Sequence Diagram

[UC14] - Make a Report in Manual Mode

The logged-in user selects a path segment on the map and opens the report-creation form. The mobile app retrieves the user's current GPS position. If the position cannot be retrieved, the app immediately shows an error.

After the user submits the form containing the report description and selected options, the mobile app performs local validation. Invalid inputs cause the app to show an error without contacting the backend.

If validation succeeds, the app sends the report payload to the backend through the **API Gateway**. Network or server-side failures lead to a timeout and an error message shown to the user.

Once the request is received, the **API Gateway** forwards the data to the **ReportManager**, which creates a report record by storing the user ID, position, and payload in the database through the **QueryManager**. If the insertion succeeds, the **DBMS** returns the generated report identifier.

Finally, the backend responds with **201 Created**, and the mobile app displays a confirmation message to the user.

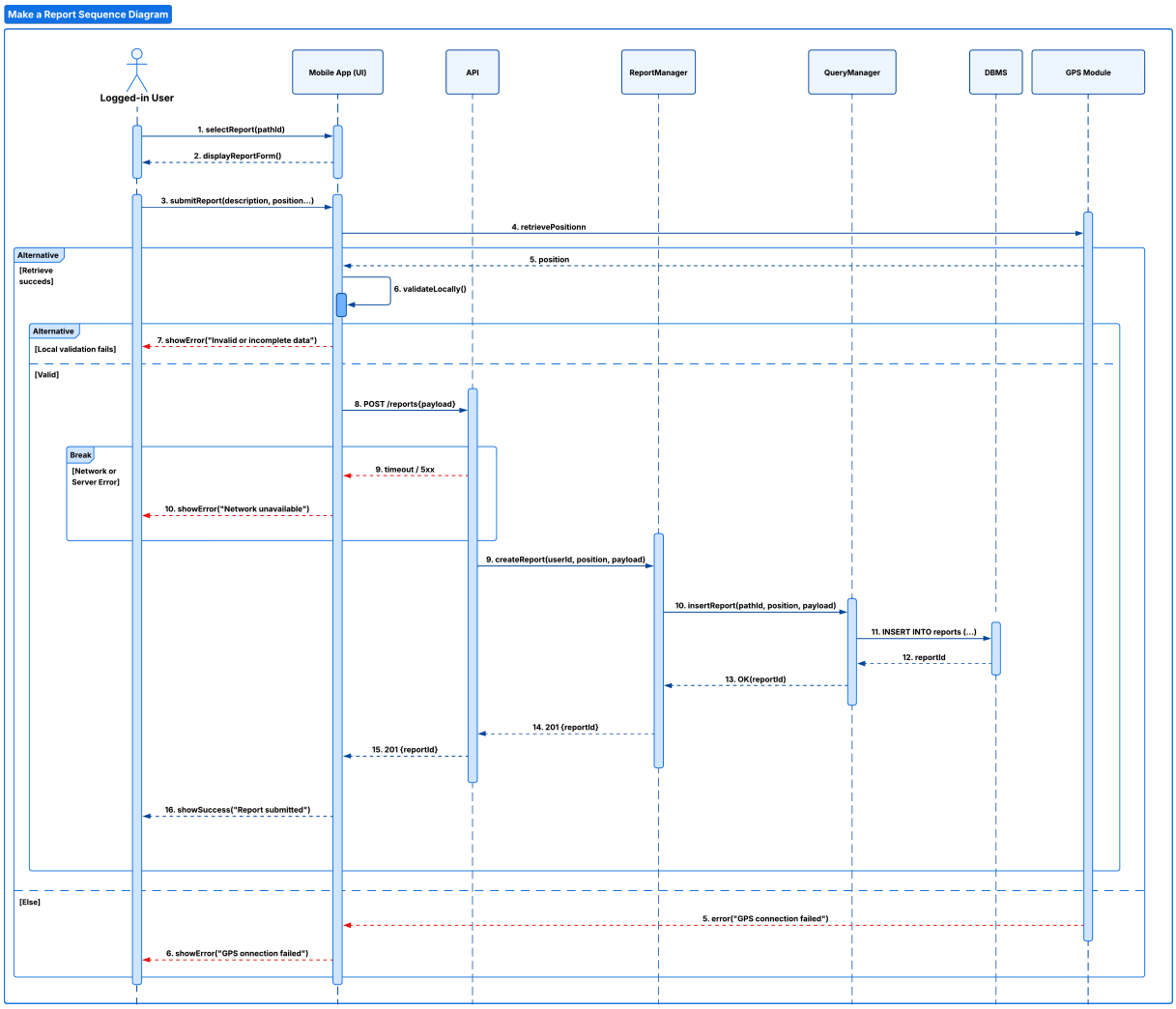

Figure 2.16: Make a Report in Manual Mode Sequence Diagram

[UC15] - Make a Report in Automatic Mode

When an obstacle is detected by the external sensors during a trip, the mobile app retrieves the current GPS position from the device. If the retrieval fails, the app displays an appropriate error message and the flow terminates.

Once the position is available, the app displays a pre-filled report form containing the detected issue. The user can review and modify the report details before submission. After the user confirms the report, the app performs local validation on the generated data. Invalid or incomplete payloads trigger a local error message and no request is sent to the backend.

If validation succeeds, the mobile app sends the report payload to the backend via the **API Gateway**. Network or server errors result in a timeout, causing the app to show a network-unavailable message.

Upon receiving a valid request, the **ReportManager** creates a new report record, storing it through the **QueryManager**, which inserts the new record into the database.

After successful insertion, the backend returns a **201 Created** response. The mobile app then informs the user that the report has been submitted successfully.

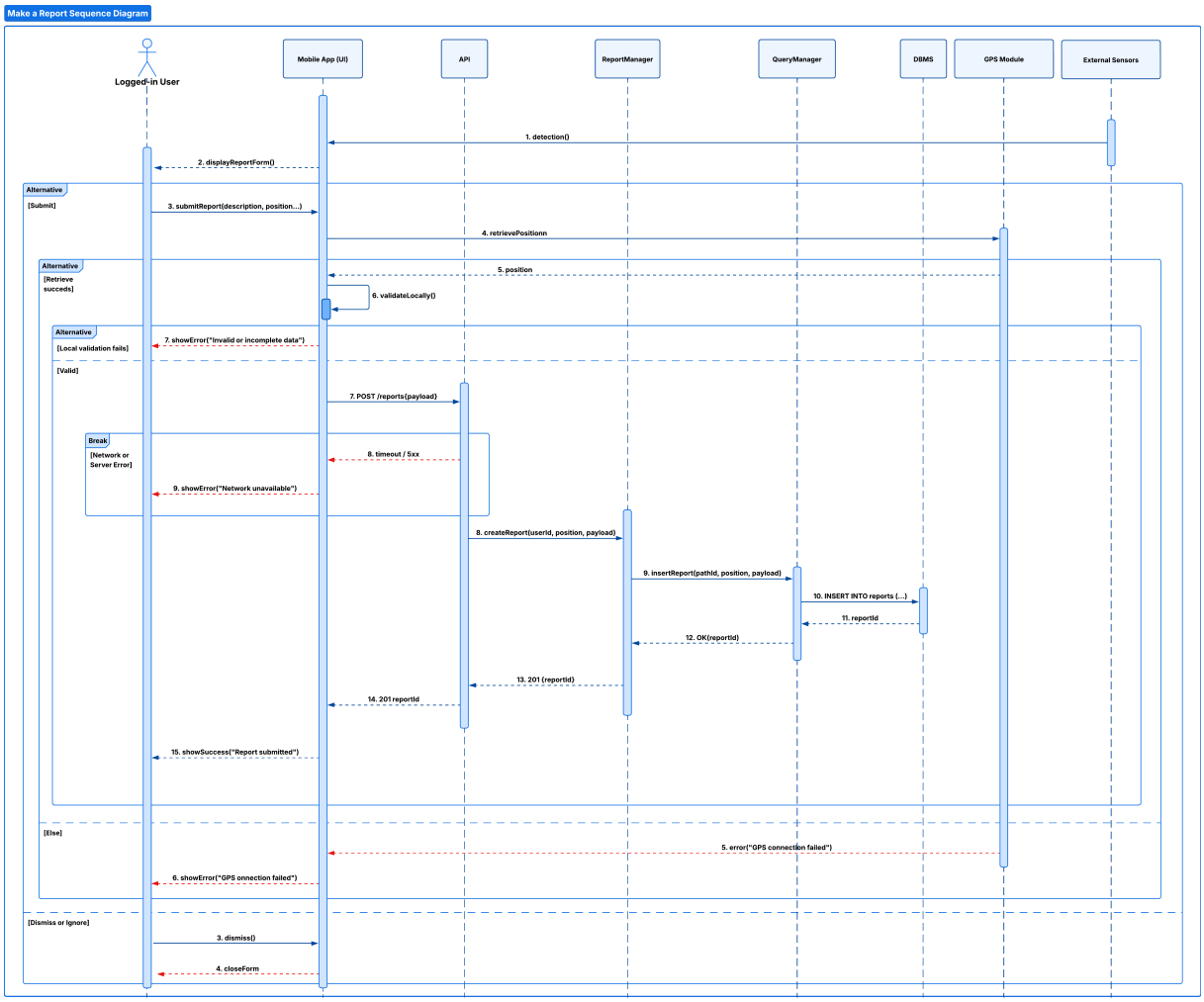

Figure 2.17: Make a Report in Automatic Mode Sequence Diagram

[UC16] - Confirm a Report

A logged-in user is on a trip, and a pop-up notification informs him of the presence of an existing report nearby. The user can choose to confirm or reject it. After the user submits the decision, the mobile app validates the input locally and, if valid, sends a request to the **backend API**.

The API forwards the request to the **ReportManager**, which creates a new confirmation entry associated with the selected report and the current user. The **ReportManager** delegates the persistence of this confirmation to the **QueryManager**, which inserts the corresponding record into the database.

If the operation succeeds, the API responds with a **201 Created** status and returns the confirmation identifier. The mobile app notifies the user that the confirmation has been submitted successfully. The user may also dismiss the form without submitting any confirmation.

In case of client-side validation errors, network failures, or server-side issues, the mobile app displays the appropriate error message.

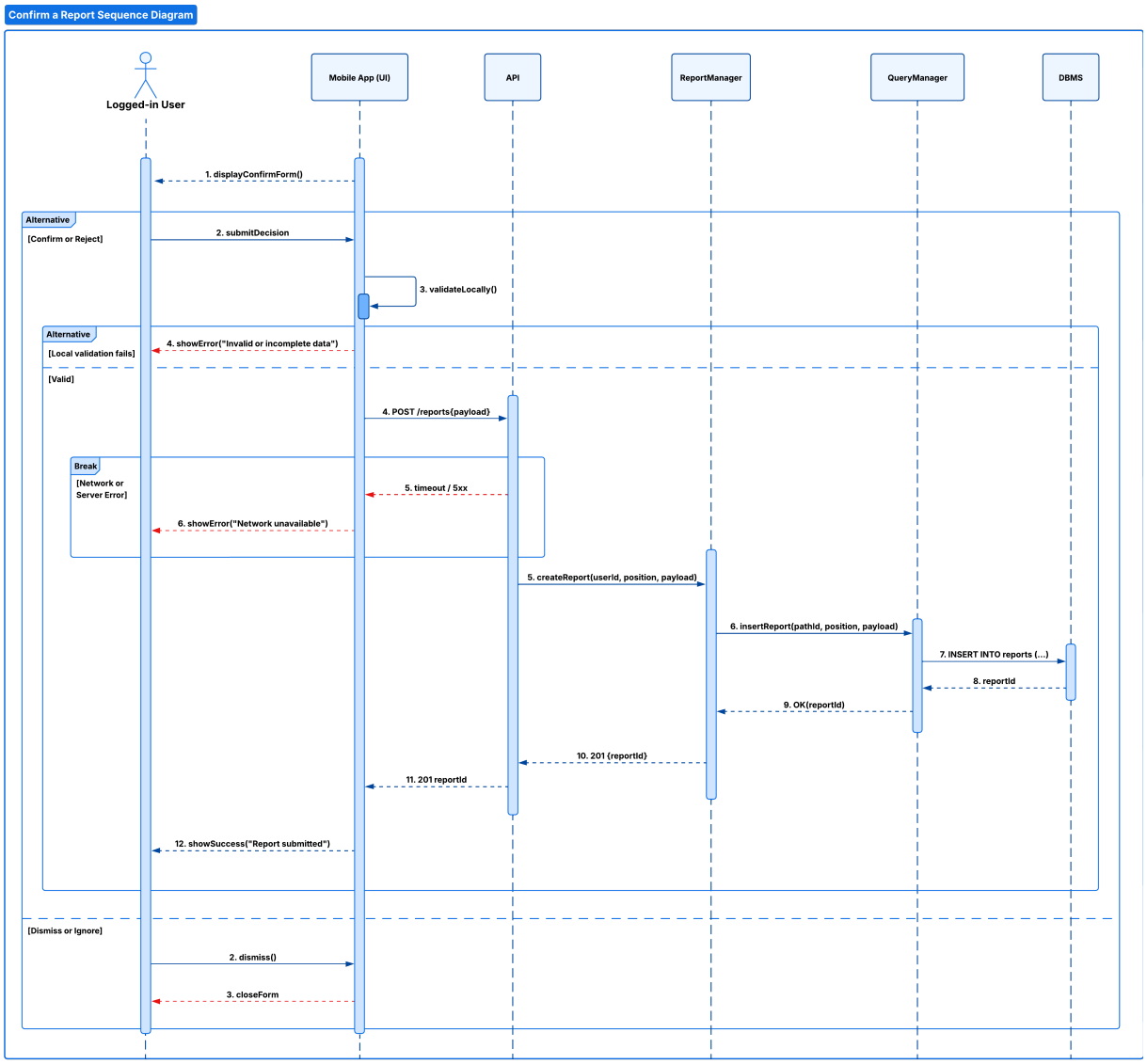

Figure 2.18: Confirm a Report Sequence Diagram

[UC17] - Manage Path Visibility

The user selects the desired path from the list of previously created paths. Then, the mobile application sends a request to the backend to retrieve the current visibility settings of the selected path. The **PathManager**, through the **QueryManager**, loads the corresponding path record from the database.

If the path is successfully found, the app displays the existing visibility configuration and waits for the user to submit the desired changes. Once the user confirms the update, the mobile application sends a request containing the updated visibility value. Upon receiving it, the **PathManager** verifies that the requesting user is indeed the owner of the path. If the ownership constraint is satisfied, the visibility attribute is updated in the database. A successful update triggers a confirmation response, and the mobile application communicates the result to the user.

If the initial fetch request fails due to network or server issues, an error message is displayed. If the selected path does not exist or no record is returned from the database, the application notifies the user accordingly. If the user attempts to modify a path they do not own, the backend returns a **403 FORBIDDEN** response, and the app signals that the operation is not permitted.

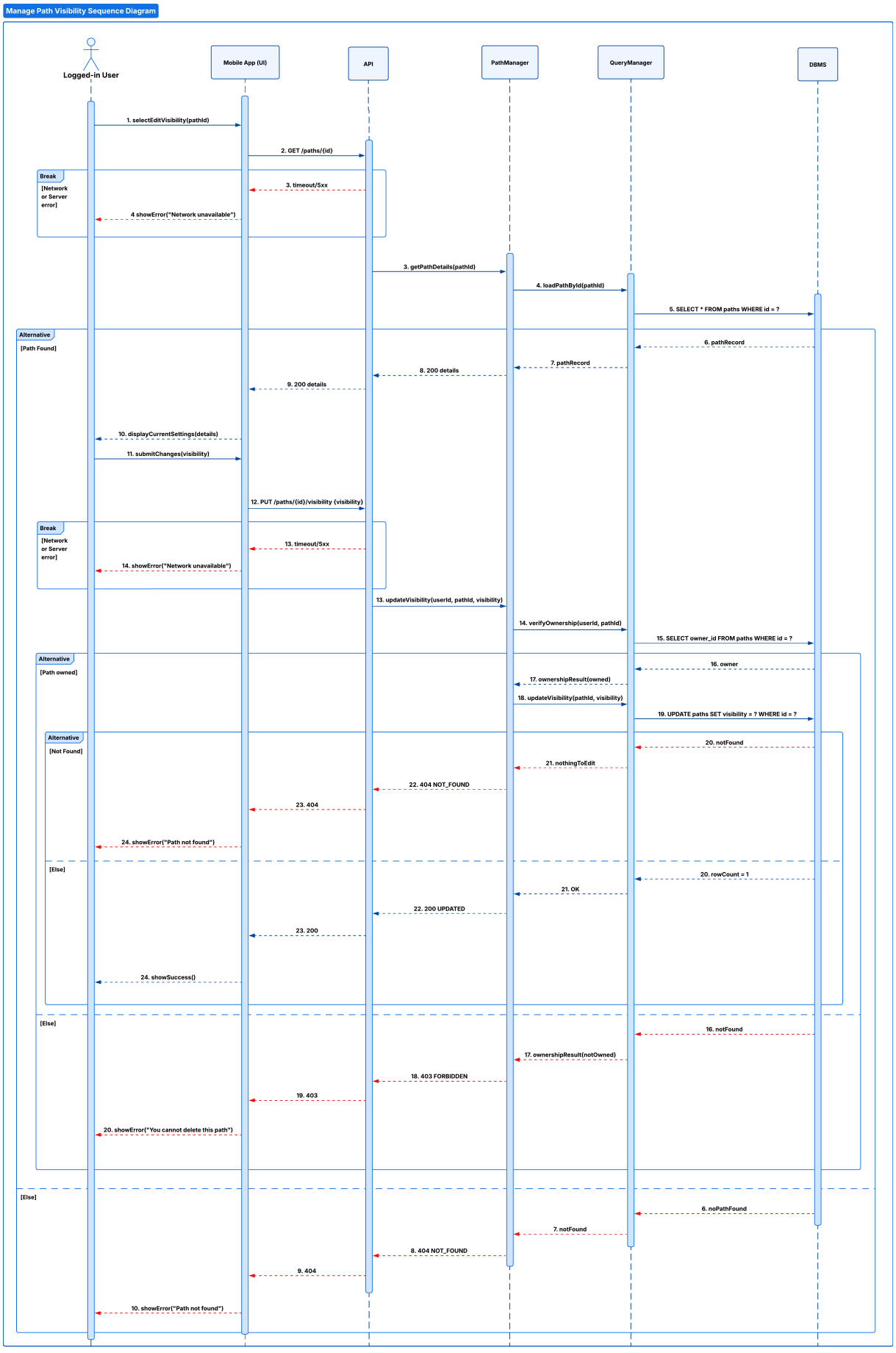

Figure 2.19: Manage Path Visibility Sequence Diagram

[UC18] - View Trip History and Trip Details

A logged-in user requests to view their trip history from the mobile app. The mobile app sends a request to the **API Gateway**. If the request succeeds, the **API** delegates the operation to the **TripManager**, which retrieves the list of the user's trips through the **QueryManager**. The **DBMS** returns the corresponding records, and the App displays the resulting history.

When the user selects a specific trip, the app issues a request. If the trip is missing or does not belong to the user, the **TripManager** returns a **404 Not Found**, which the client displays accordingly.

If the trip exists, the **TripManager** loads full details from the **DBMS**, including timestamps, distance, speed metrics, the associated path, and any storedweather snapshots. The details are returned to the App, which presents a complete summary of the selected trip.

In case of any network or server failures, the app notifies the user with an appropriate error message.

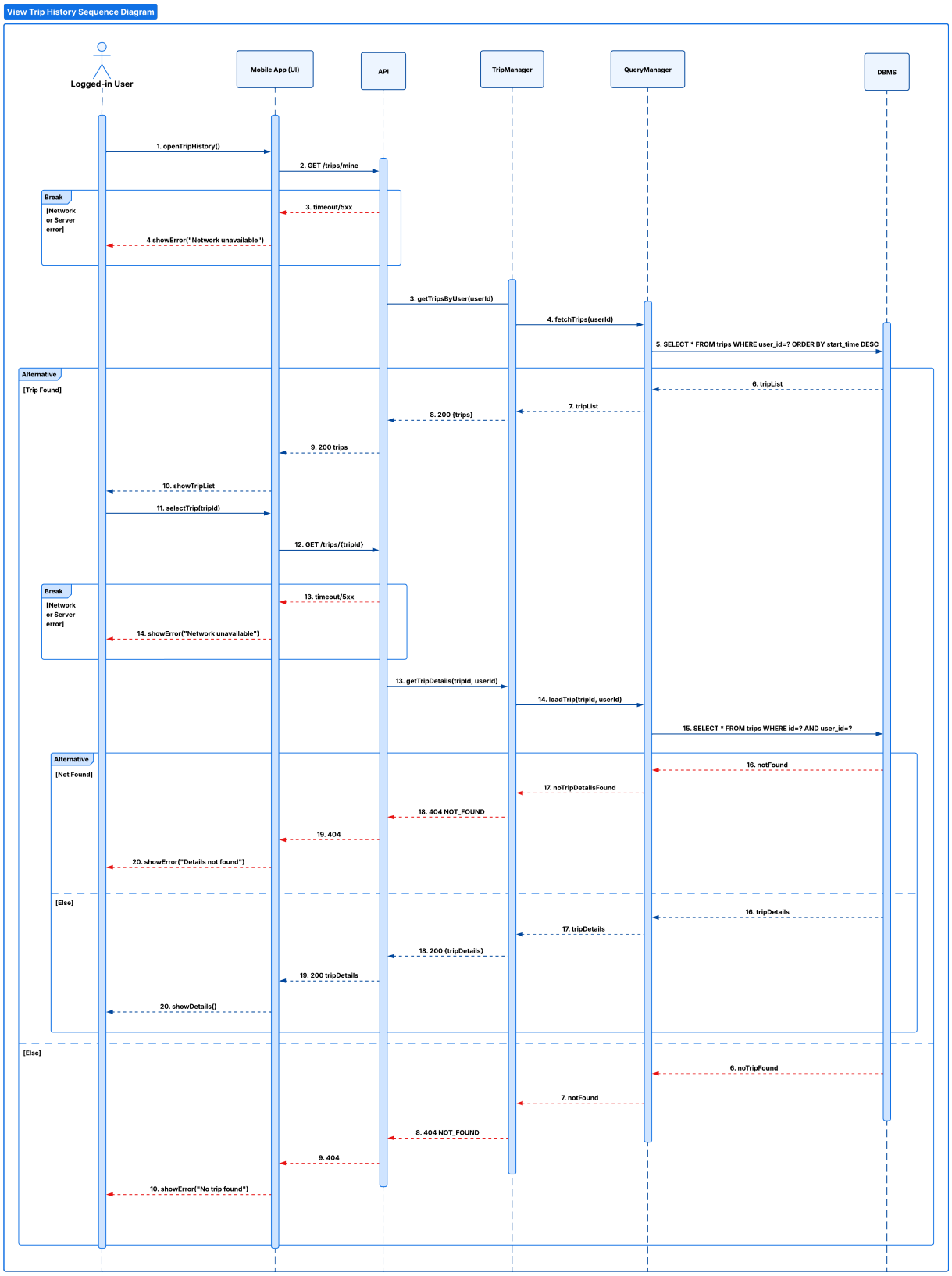

Figure 2.20: View Trip History and Trip Details Sequence Diagram

[UC19] - View Overall Statistics

Upon opening the statistics section, the mobile app requests the user's overall metrics from the backend. If the request cannot be completed due to network or server issues, an error message is shown. Otherwise, the **API** forwards the request to the **StatsManager**, which loads the user's trip history via the **QueryManager**. The latter retrieves all relevant rows from the **DBMS**.

If trip data exists, the **StatsManager** computes all required aggregates (e.g., total distance, duration, average speed) and returns the final statistics to the mobile app, which then displays them.

If no trip records are found, the system returns a **404 NOT_FOUND** response and the app informs the user accordingly.

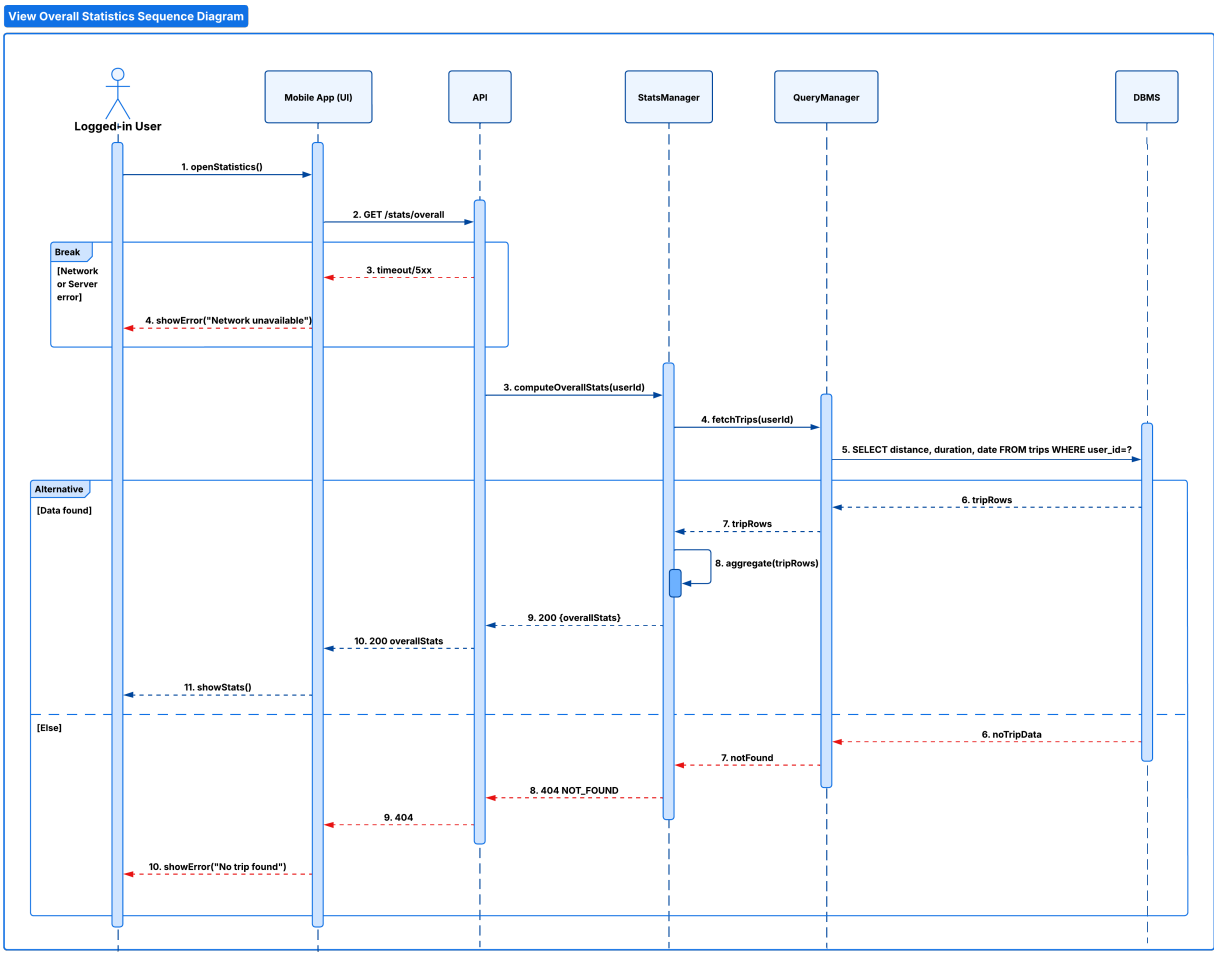

Figure 2.21: View Overall Statistics Sequence Diagram

[UC20] - View Trip Statistics

When the Logged-in user selects a trip, the mobile app sends a request for detailed metrics. Network or server failures are handled locally by showing an appropriate error.

If the request reaches the backend, the **API** delegates it to the **StatsManager**, which loads the associated trip data through the **QueryManager** by querying the **DBMS**. If the requested trip is found, the **StatsManager** computes the aggregated statistics and returns them to the mobile app, which presents them to the user.

If the trip does not exist or does not belong to the user, the backend replies with a **404 NOT_FOUND** response and the app notifies the user.

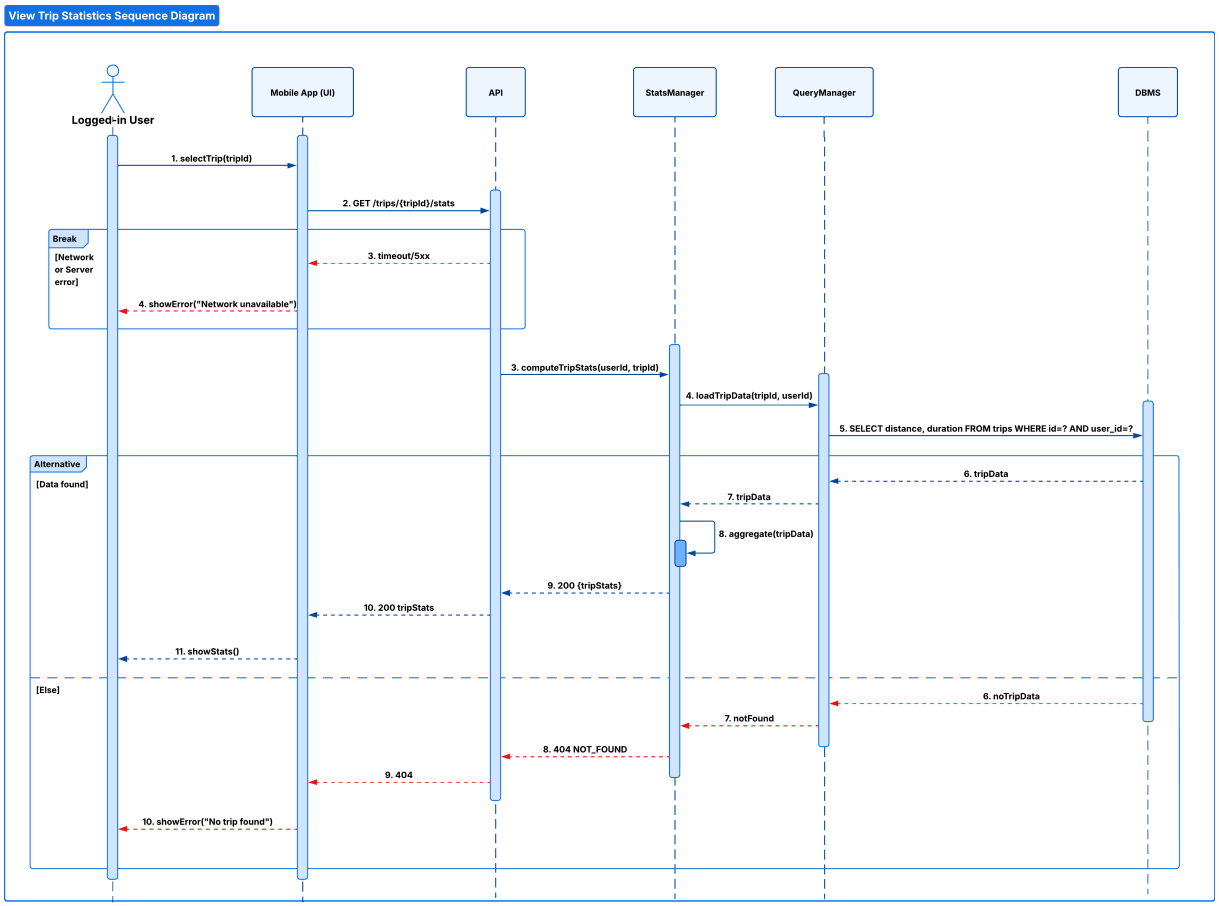

Figure 2.22: View Trip Statistics Sequence Diagram

[UC21] - Edit Personal Profile

After the logged-in user opens the edit form, the mobile app locally validates the submitted fields. If the data is incomplete or invalid, the app immediately notifies the user. If the input is valid, the updated payload is sent to the **API**, which delegates the request to the **UserManager**.

The update is forwarded to the **QueryManager**, which issues the corresponding **UPDATE** operation on the database. If the update succeeds, the modified user profile is returned to the app and displayed to the user.

In case of network or server errors during the request, the mobile app shows an appropriate error message.

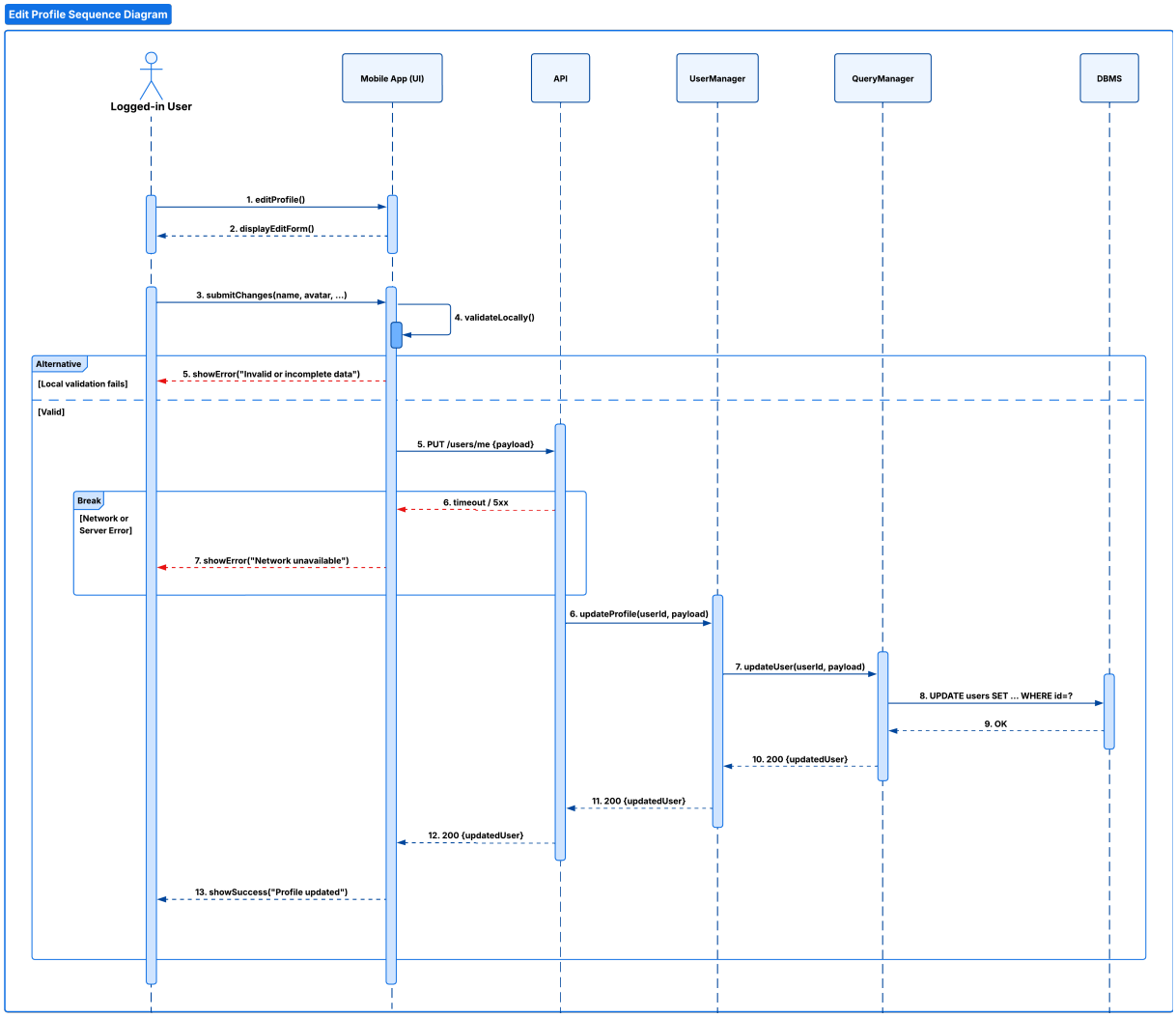

Figure 2.23: Edit Personal Profile Sequence Diagram

2.6. Component Interfaces

2.7. Selected architectural styles and patterns

2.8. Other Design Decisions

3 | User Interface Design

3.1. User Interfaces

All mockups included here are conceptual prototypes that illustrate the expected interaction flow, the structure of each screen, and the behaviour of both logged-in and guest users. While the visual style is representative of the final layout, the graphical design may undergo refinements during implementation.

This section provides a high-level visualization of how users interact with the system across its main features, including authentication, map exploration, path search and selection, trip navigation, obstacle reporting, custom path creation and management, and profile settings. The screens also highlight how certain functionalities change depending on the user state (guest or logged-in), ensuring a clear understanding of the overall user experience.

3.1.1. Welcome Screen

The screen displays the app logo, the application name, and a short tagline summarizing its purpose: discovering bike paths, tracking rides, and joining a cycling community. Three navigation options are provided: Sign In, Log In, and Continue as Guest. The Sign In button directs new users to the registration screen to create an account, while the Log In button allows returning users to access their existing accounts. The Continue as Guest option enables users to explore the app without registering, offering limited functionality. This screen serves as the entry point to all subsequent interactions and does not require any backend communication. Its goal is to guide the user toward authentication or guest access in a clear, minimal, and intuitive way.

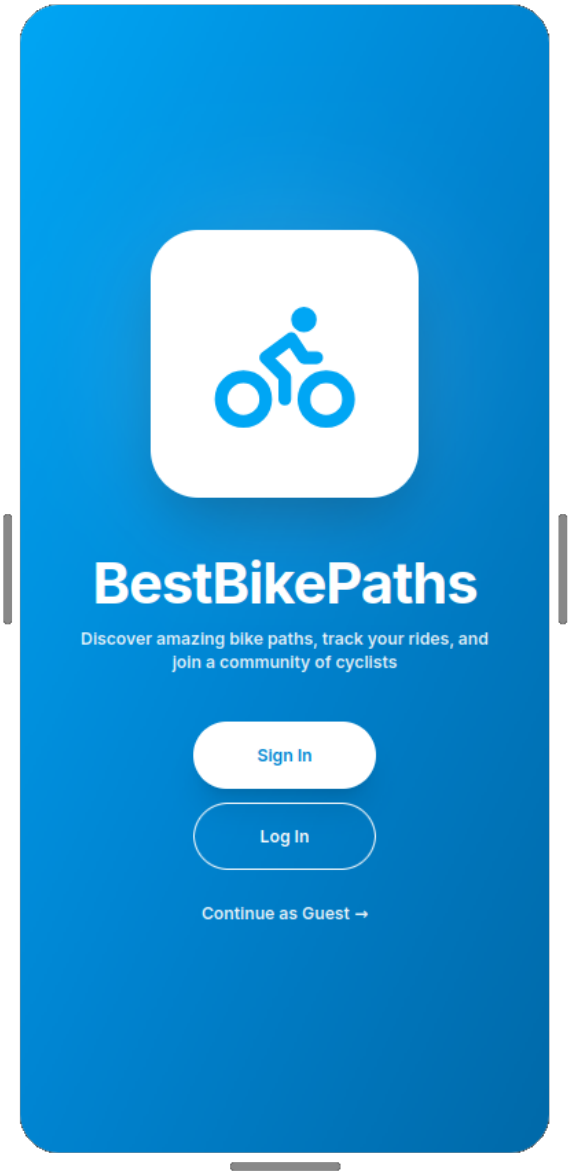

Figure 3.1: Welcome Screen Mockup

3.1.2. Login Screen

After choosing the Log In option from the Welcome Screen, the user is taken to the Login Screen. This interface allows returning users to authenticate by entering their email address and password. Below the login form, users can navigate back to the Welcome Screen. This screen ensures secure access to all BBP features that require authentication, including trip recording with data storage, path creation, reporting, and access to statistics.

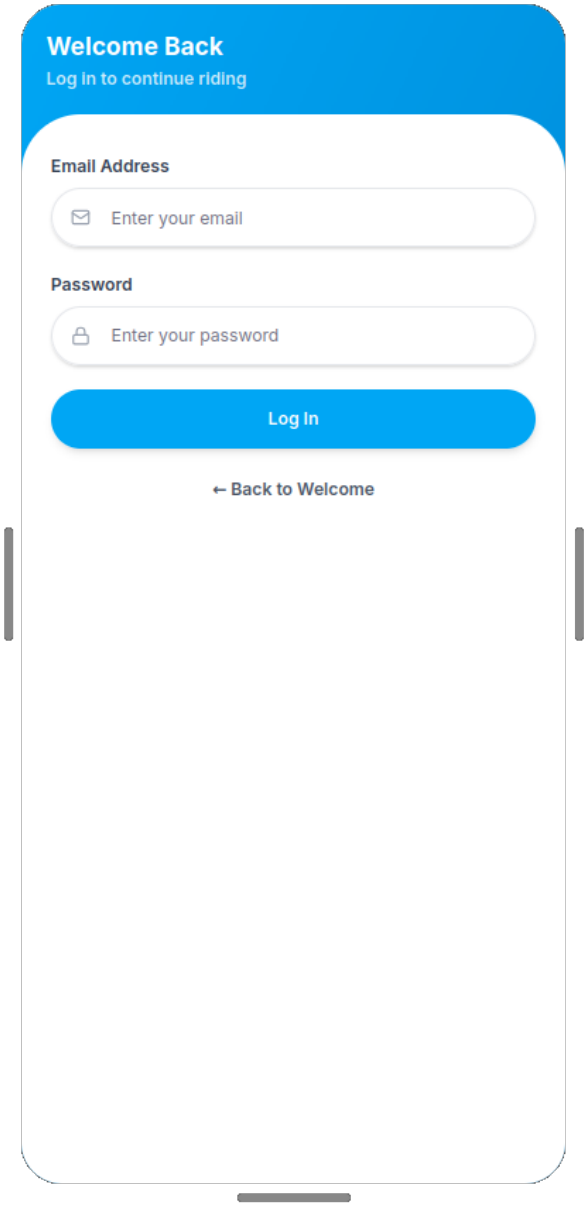

Figure 3.2: Login Screen Mockup

3.1.3. Signup Screen

The Signup Screen, reachable through the welcome page, enables new users to create an account by providing a username, email address, password, and password confirmation. This interface is designed to keep the registration process simple, requiring only the essential information needed to activate a personal profile. Users can switch to the Welcome Screen by pressing the dedicated button. Creating an account unlocks all core functionalities of BBP, such as recording trips, submitting reports, managing custom paths, and accessing personal statistics.

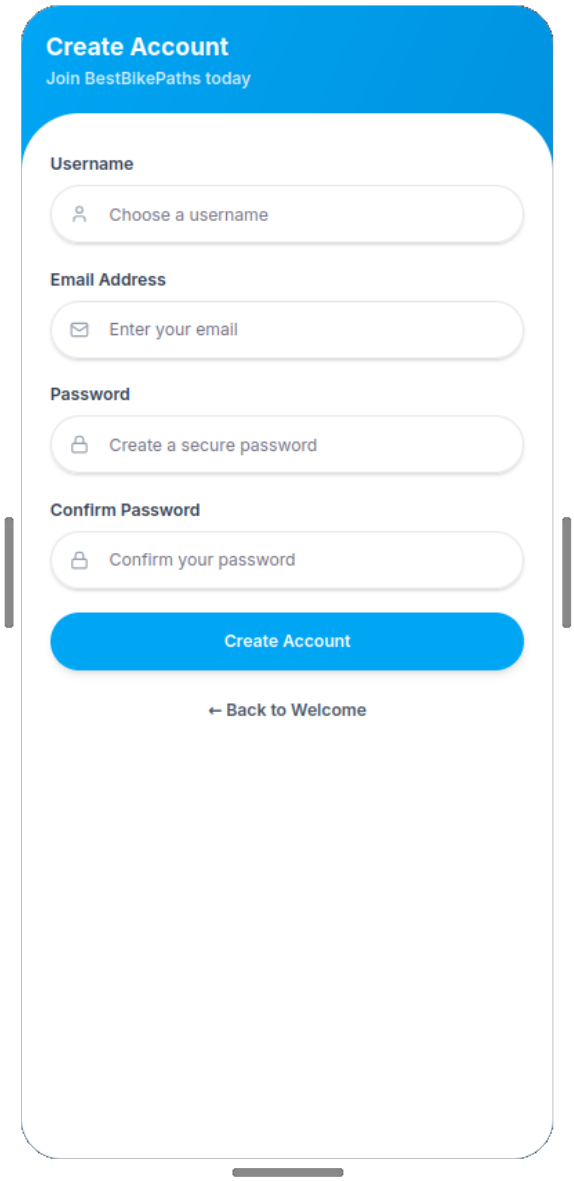

Figure 3.3: Signup Screen Mockup

3.1.4. Home Screen

The Home Screen is the central hub of the BBP application, allowing users to search for bike paths, view the interactive map, and navigate to all major sections of the app.

At the top of the interface, users can enter a Starting point and a Destination using two text fields. By default, the starting point is set to the user's current GPS location, but it can be manually edited if needed. Below the input fields, the Find Paths button initiates the path search process, retrieving all available bike routes that connect the selected points. The map occupies most of the screen, displaying the user's current position and

any computed paths. In the lower-right corner, a floating action button with a plus icon is available for logged-in users. Pressing it opens the path creation flow, allowing them to create a custom bike path.

At the bottom of the screen, a persistent navigation bar provides access to the main sections of the app: Home, Trip History, Path History, and User Profile.

Guest users can fully interact with the map and use the path search form. However, features that require authentication, such as creating new paths, viewing personal trip history, or accessing the user profile, remain disabled. Icons in the bottom navigation bar appear visually greyed out, and a lock symbol indicates restricted access. Tapping a disabled icon triggers a pop-up inviting the user to sign in to unlock all the app's features. The creation floating action button is also greyed out, as path creation is restricted to logged-in users. This interface clearly conveys the distinction between freely accessible features and those requiring authentication.

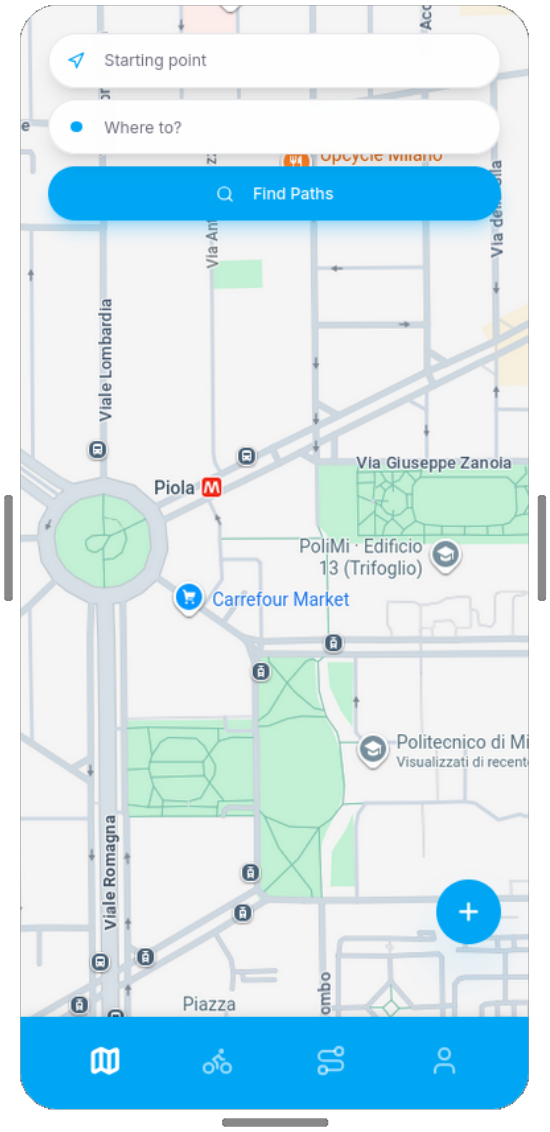

Figure 3.4: Home Screen Mockup

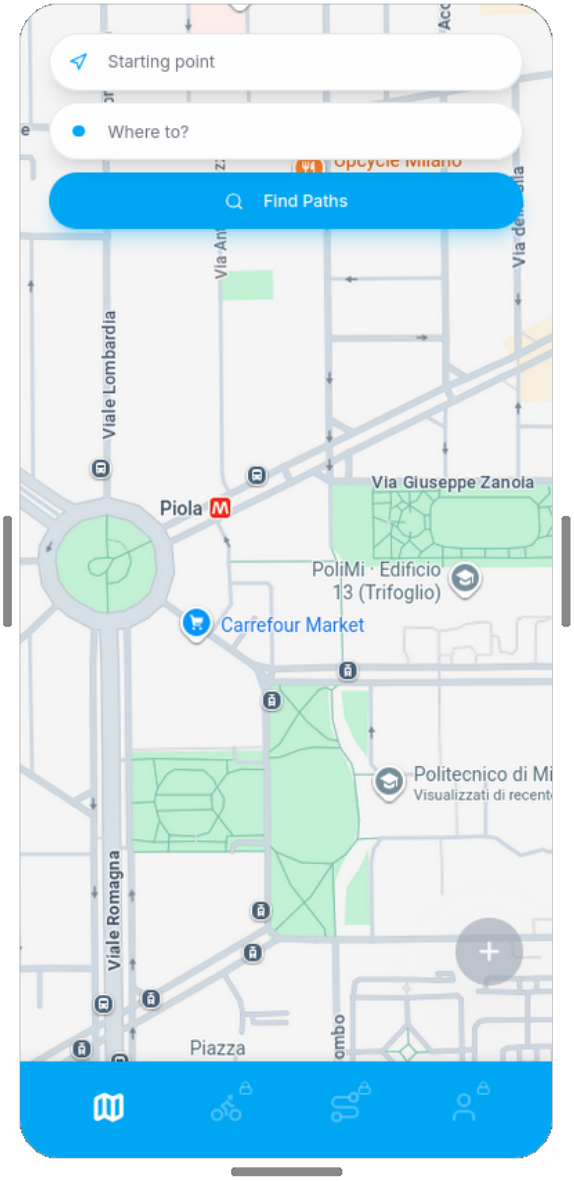

Figure 3.5: Home Screen Mockup for Guest Users

3.1.5. Authentication Pop-up for Guest Users

When a guest user attempts to access a restricted feature, the app displays a modal pop-up overlay informing them that the selected functionality is available only to authenticated users. The pop-up contains a short explanation and two action buttons: one to sign in, and one to continue as a guest without enabling the restricted action.

Behind the pop-up, the rest of the interface is dimmed and temporarily disabled, ensuring that the user's attention remains on the modal and preventing interaction with the under-

lying elements. The modal can be closed either by selecting one of the available actions or by tapping outside the pop-up area, returning the guest user to the current screen and allowing them to continue browsing the map or searching for bike paths uninterrupted.

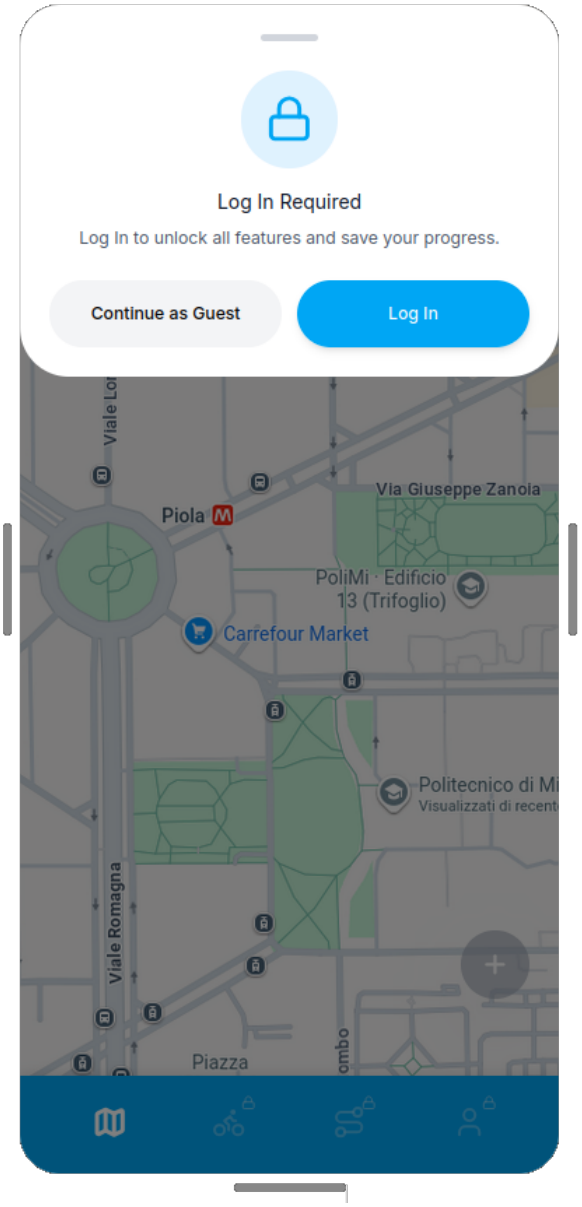

Figure 3.6: Authentication Pop-up Mockup for Guest Users

3.1.6. Search Results

After the user submits a path search from the home screen, the app displays an "Available Paths" panel anchored to the top of the map. The panel lists all suggested paths matching the search criteria. Each item shows the path name, a short description if available, estimated distance and travel time, and its current condition (e.g., Optimal, Maintenance)

together with the number of aggregated reports. The underlying map remains visible as a background reference centered on the searched area, while the user scrolls through the list to compare the alternatives. Tapping the close icon in the top-right corner of the panel dismisses the results and returns the interface to the previous state, where the user can enter a new search and run it again.

The behavior is identical for logged-in and guest users, except for the general restrictions applied to guests.

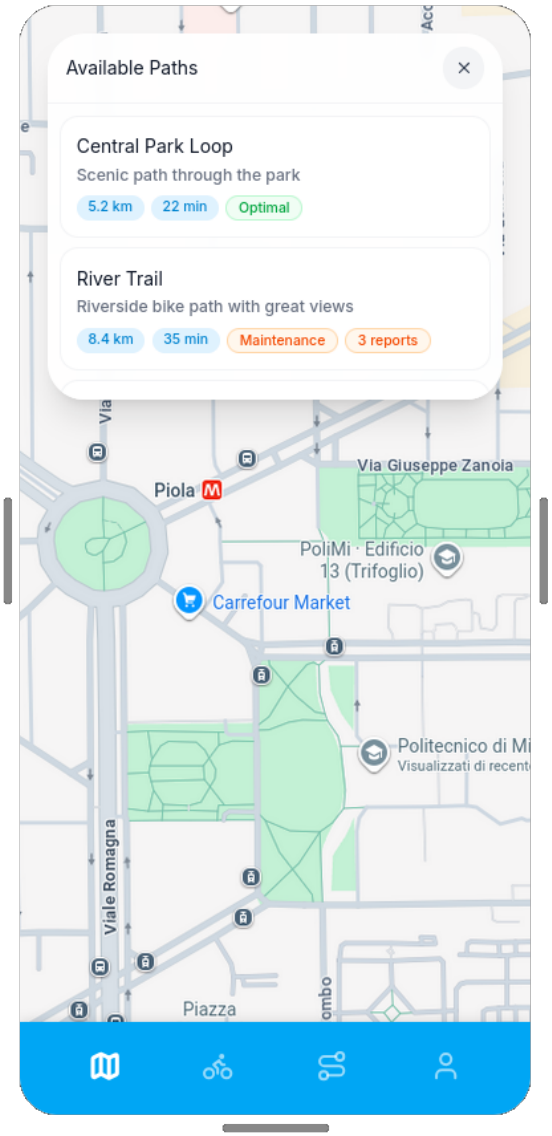

Figure 3.7: Search Results Mockup

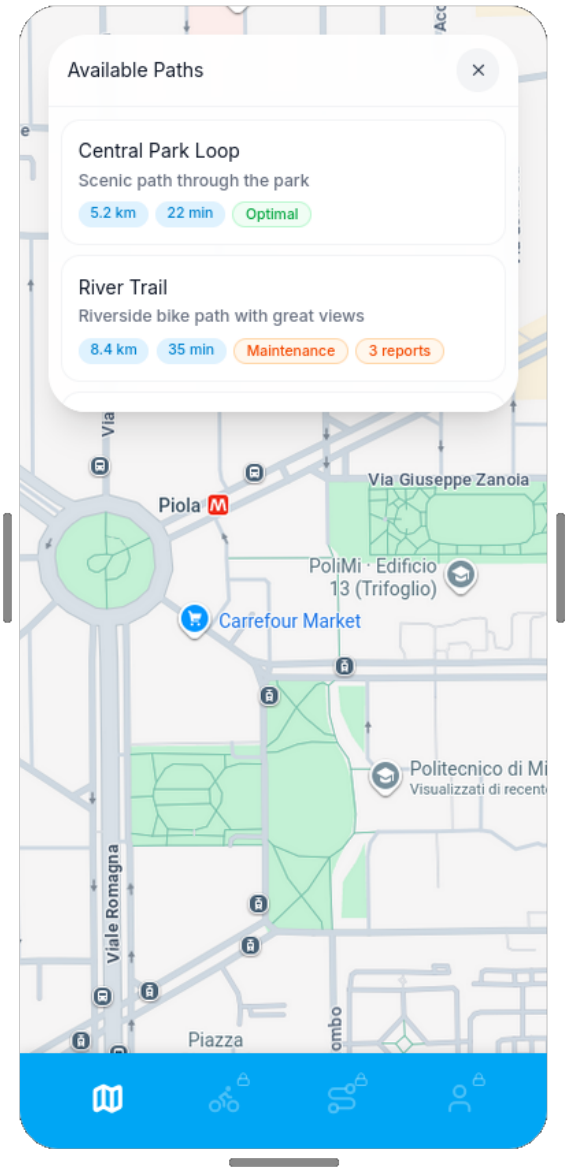

Figure 3.8: Search Results Mockup for Guest Users

3.1.7. Path Selection

When the user taps on one of the suggested paths in the results list, BBP highlights the selected option by applying a blue outline around the corresponding card. This visual cue clearly indicates which path is currently active while keeping the rest of the list unchanged. Tapping a different card updates the selection, tapping on the same card again deselects it, while tapping outside does not alter the current state.

Once a path is selected, the app displays the corresponding route on the map. The route is shown using a bold blue polyline, along with the estimated distance and travel time, giving the user an immediate visual understanding of the full trip. If the selected path includes confirmed reports, their corresponding markers are displayed on the map, allowing the user to identify potentially problematic segments.

A "Start Trip" button appears inside the selected card only when the path origin matches the user's current GPS position. If the origin does not correspond to the current position, the app displays the path preview and its route on the map, but the user can't start a trip from that location.

When the Start Trip button is available and pressed, for logged-in users, the system opens a pop-up requesting whether to enable the Automatic Report Mode. Once the mode is selected, the trip begins.

For guest users, the trip can still be started, but the activity is not stored in the system and no automatic detection features are available.

Pressing the close icon at the top of the results panel hides the list and returns the interface to the standard map view without an active selection.

The behaviour is identical for logged-in and guest users regarding map visualization, path selection, and result browsing. However, only logged-in users can access advanced features from the bottom navigation bar, which appear disabled in guest mode.

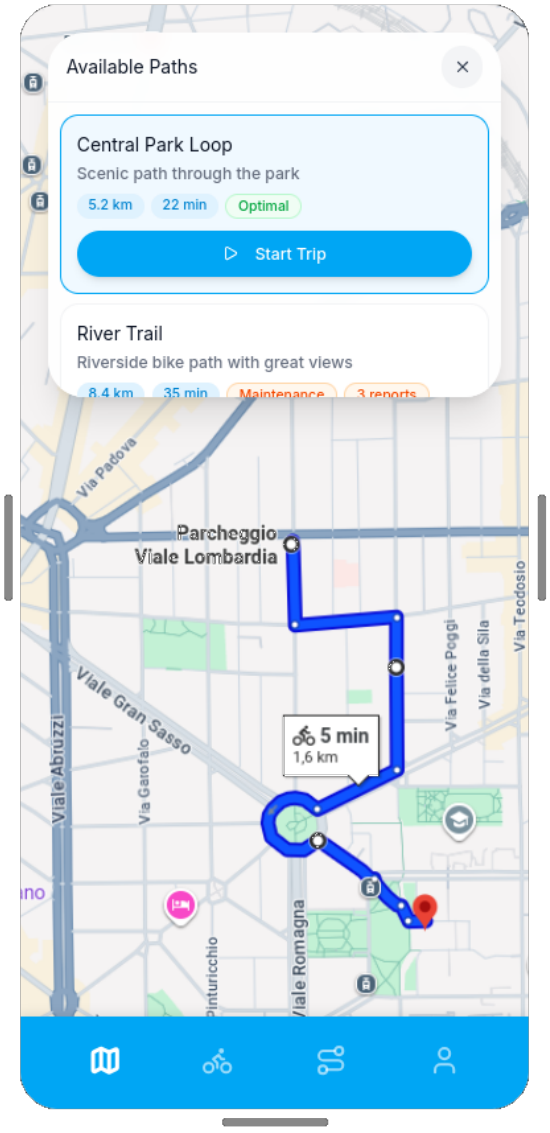

Figure 3.9: Path Selection Mockup

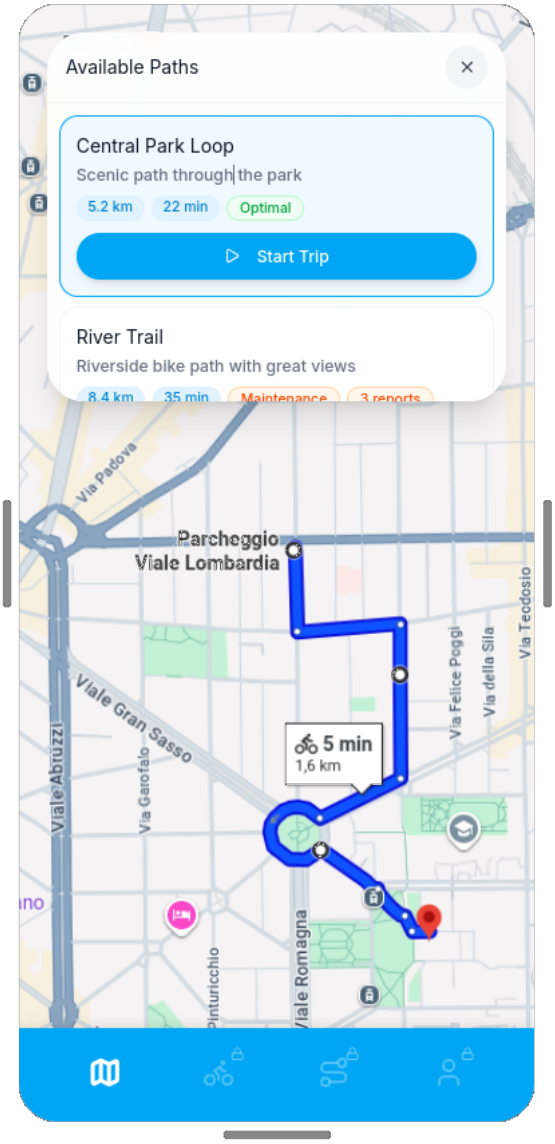

Figure 3.10: Path Selection Mockup for Guest Users

3.1.8. Automatic Mode Activation

After tapping the Start Trip button, logged-in users are shown a pop-up prompting them to enable the Automatic Report Mode. This mode activates the device's sensors (gyroscope and accelerometer) to automatically detect potential obstacles such as potholes or rough surfaces. Whenever an anomaly is detected, BBP prepares a pre-filled report and displays a confirmation prompt for the user to review, edit, or submit.

If the user continues, the trip starts immediately and the system begins tracking the movement along the selected path. Automatic detection does not prevent manual reporting:

users may still send manual reports at any time during the trip through the dedicated interface.

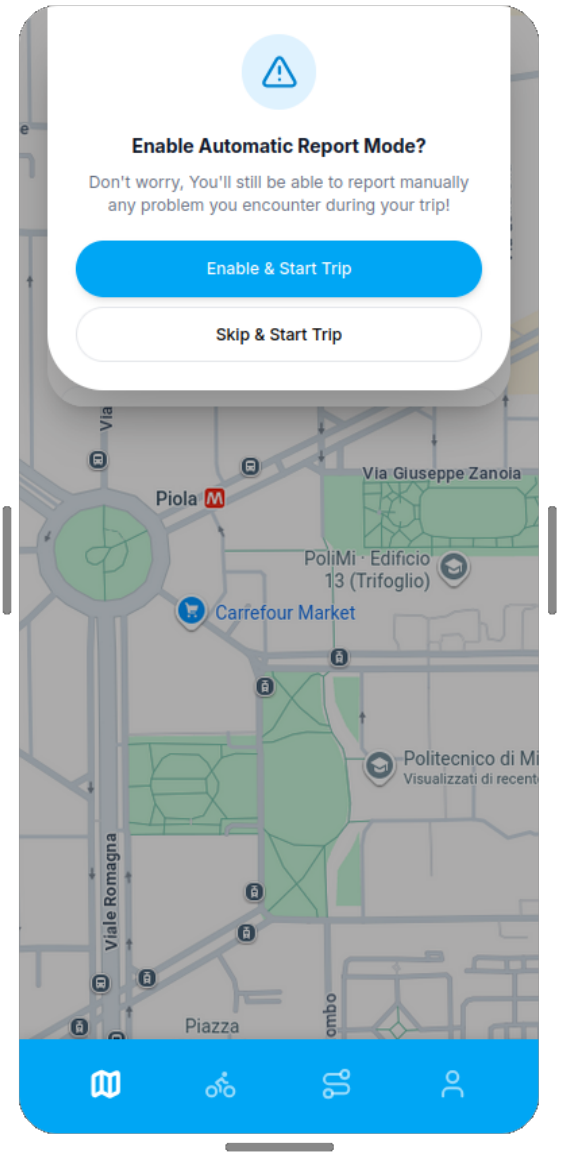

Figure 3.11: Automatic Mode Activation Mockup

3.1.9. Navigation View

During an active trip, the interface switches to a dedicated full-screen navigation mode. The map occupies the entire screen, displaying the user's real-time position and the selected path highlighted with a thick blue route polyline. No bottom navigation elements are accessible during a trip, ensuring that the user remains fully focused on navigation.

At the bottom of the screen, a large Complete Trip button allows the user to manually

end the session at any moment. On the lower-right side, the interface shows a floating report button red and active for logged-in users, enabling manual obstacle reporting. The button is grey and disabled for guest users, who can't submit reports, and if clicked it triggers a pop-up inviting them to sign in. The button's floating position and color clearly distinguish whether reporting is available.

A trip session ends when the user presses the Complete Trip button, the destination is reached or when the system detects that the user has significantly deviated from the planned path. When the trip ends, BBP stops tracking and displays a confirmation message.

For logged-in users, the trip data is stored in their personal history. For guest users, the navigation experience is identical, but no information is saved once the trip is completed.

This view visually reinforces the functional difference between user types: although both can ride along a selected path, only logged-in users can contribute reports and preserve their trip data.

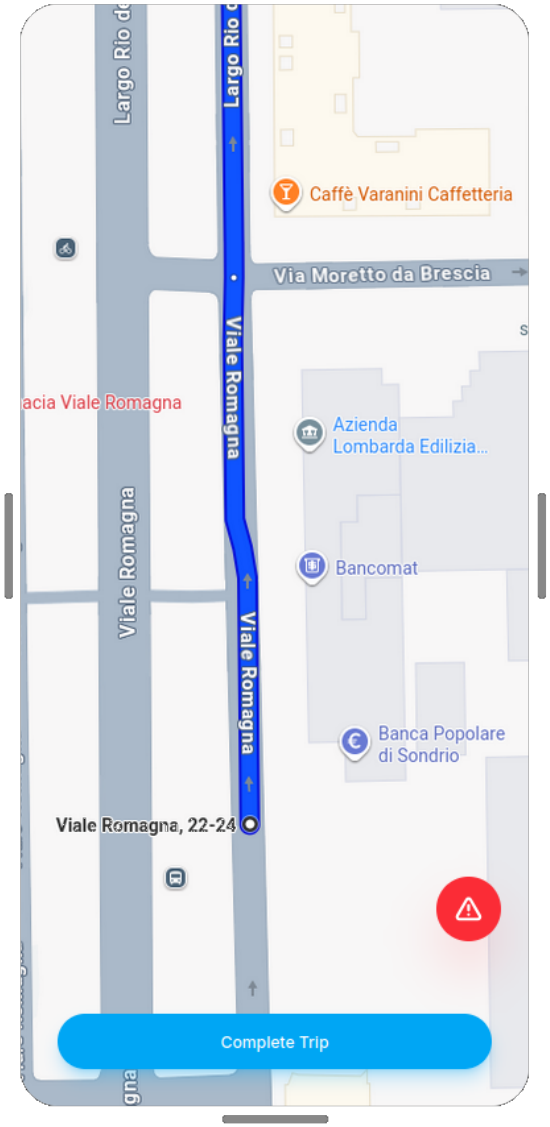

Figure 3.12: Navigation View Mockup

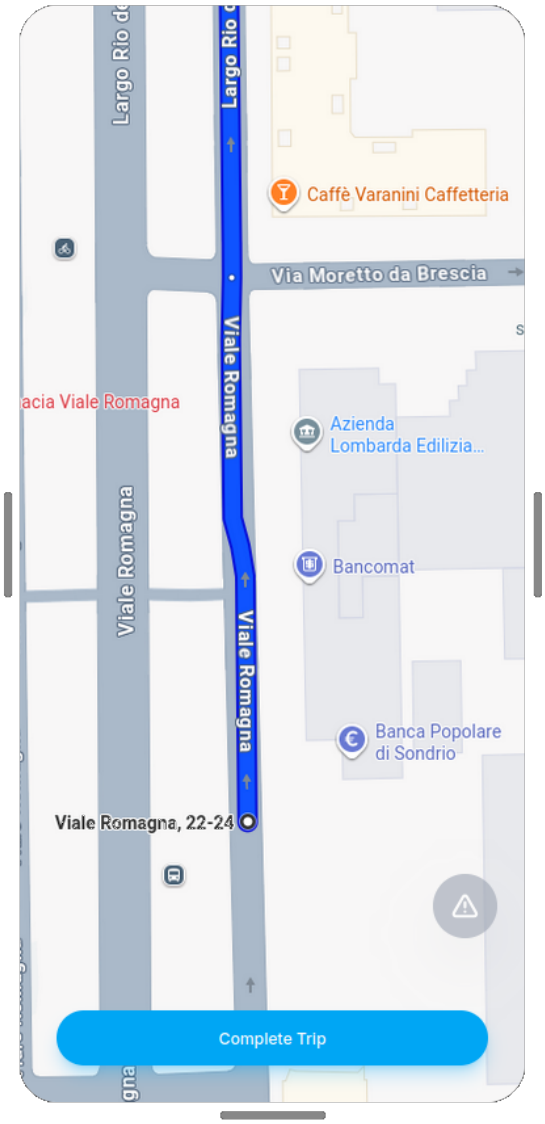

Figure 3.13: Navigation View Mockup for Guest Users

3.1.10. Trip Completion

At the end of a trip, the app displays a completion pop-up that summarizes the outcome of the session. This pop-up appears in all termination scenarios.

For logged-in users, the pop-up shows the message “Great Ride!” and includes a View Stats button. Tapping this button redirects the user to the Trip History screen and automatically opens the detailed summary of the trip that has just been completed. If the user taps outside the dialog, the pop-up simply closes and the map becomes interactive again.

For guest users, the pop-up displays “Trip Complete” along with a reminder that trip statistics are not saved in guest mode. Only one button is shown, which closes the dialog. Since guest trips are never stored, no option to view statistics is provided.

In both cases, the background map is dimmed while the modal is visible, preventing interaction until the user acknowledges the pop-up.

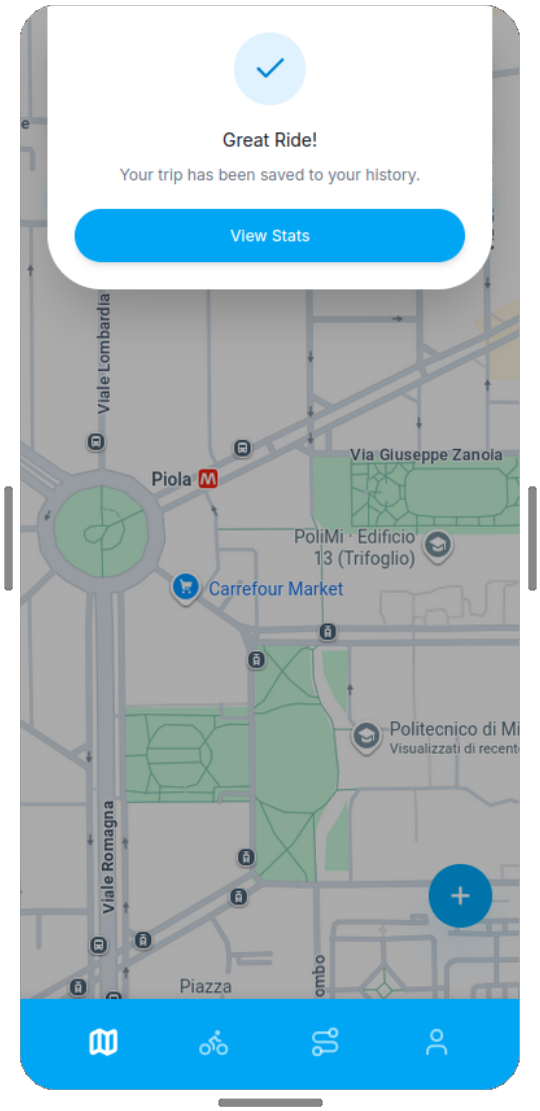

Figure 3.14: Trip Completion Mockup

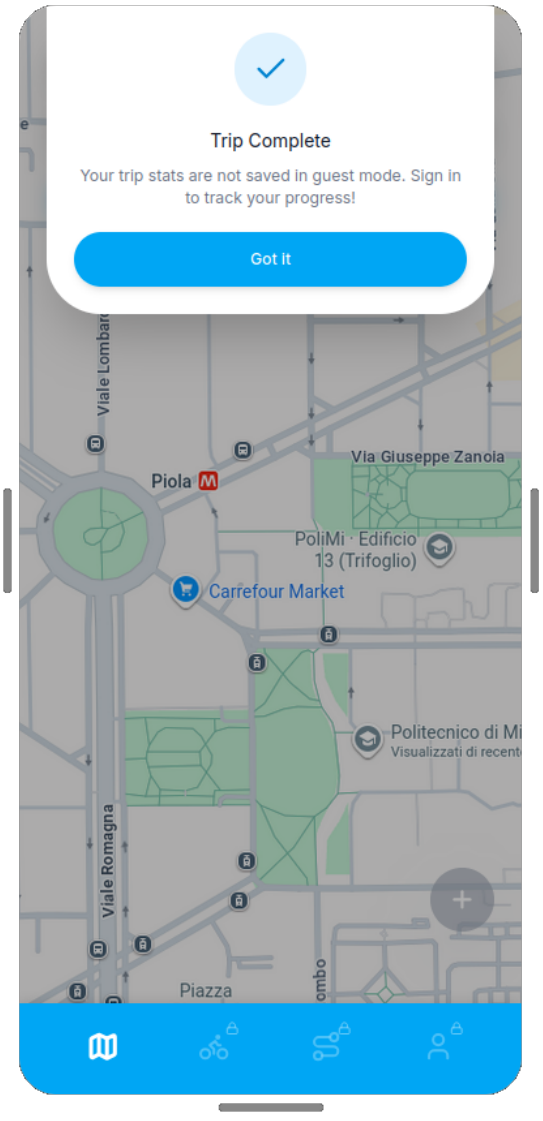

Figure 3.15: Trip Completion Mockup for Guest Users

3.1.11. Report Submission

During an active trip, logged-in users can submit a report through a dedicated pop-up interface. The report dialog appears when the user taps the red report button or when an

automatic detection triggers a report prompt. The pop-up overlays the current navigation view and pauses the map interaction by dimming the background.

The pop-up contains two fields: Path Condition, allowing the user to assign the current status of the affected segment (e.g., Optimal, Requires Maintenance, Closed), and Obstacle Type, specifying the encountered issue (e.g., Obstacle, Path Damage, Work in Progress).

For manual submissions, both fields are initially empty and must be filled in by the user before submitting. For automatic detections, the fields are pre-filled with the values inferred from sensor data, but the user may freely edit the information, confirm it, or discard the report entirely.

Once the report is successfully submitted, the app displays a confirmation pop-up titled "Report Submitted", thanking the user for contributing to the community. The dialog includes a Continue Trip button, which closes the pop-up and returns the user to the navigation view so they can resume riding without interruption. If the user does not interact with the dialog, it will automatically close after a short timeout.

Only logged-in users can access this feature. In guest mode, the report button is visible but appears greyed out and can't be pressed.

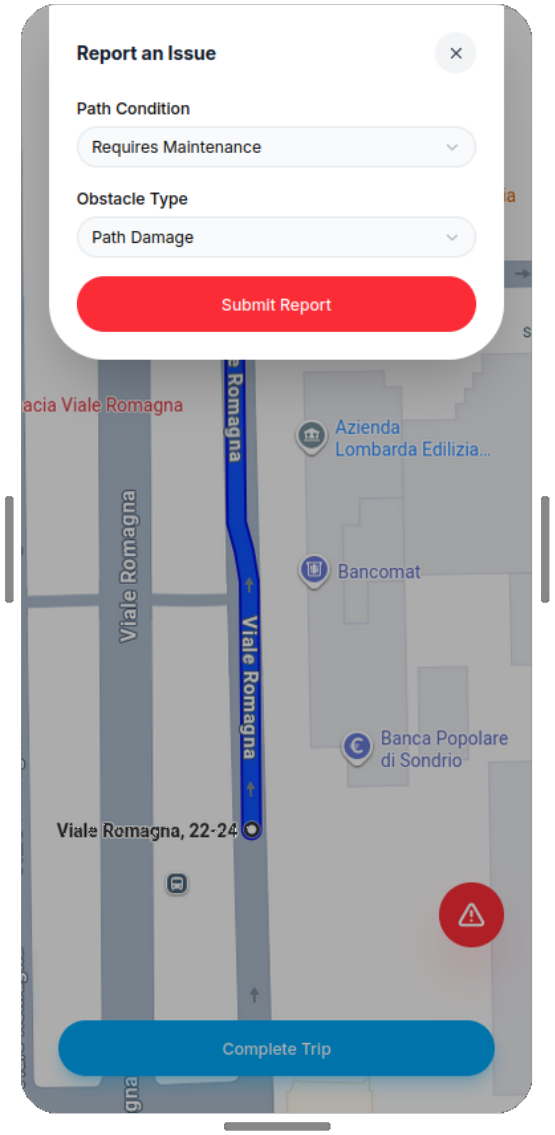

Figure 3.16: Report Submission Mockup

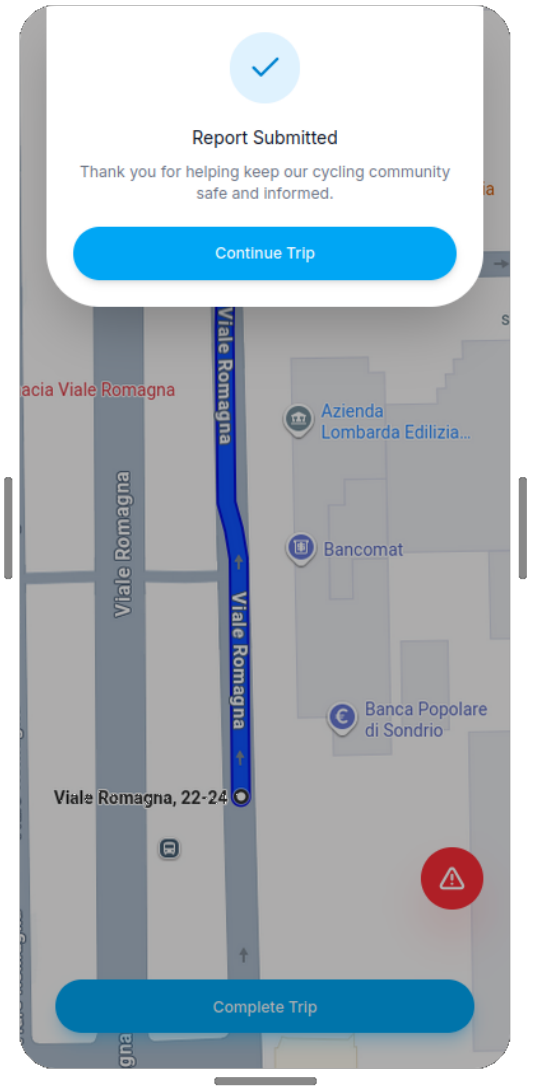

Figure 3.17: Report Submission Success Message Mockup

3.1.12. Report Confirmation

While following a path during an active trip, BBP notifies logged-in users when they approach a location where another cyclist previously submitted a report. In this situation, a confirmation pop-up appears at the top of the screen with the message "Obstacle Detected", informing the user that an obstacle was previously reported at that spot.

The dialog asks the user to confirm whether the issue is still present, offering two options, one to confirm that the report is still valid and another to mark it as no longer present.

The background map is dimmed to draw attention to the dialog while keeping the naviga-

tion visible. If the user does not respond within a short timeout, the pop-up automatically closes so as not to interrupt the ride.

After the user chooses either option, a second pop-up briefly appears to confirm that their response has been recorded, allowing them to immediately resume their trip. This confirmation dialog automatically closes after a short delay if the user does not interact with it

This feature is available only for logged-in users, guest users do not receive confirmation prompts and can't validate reports.

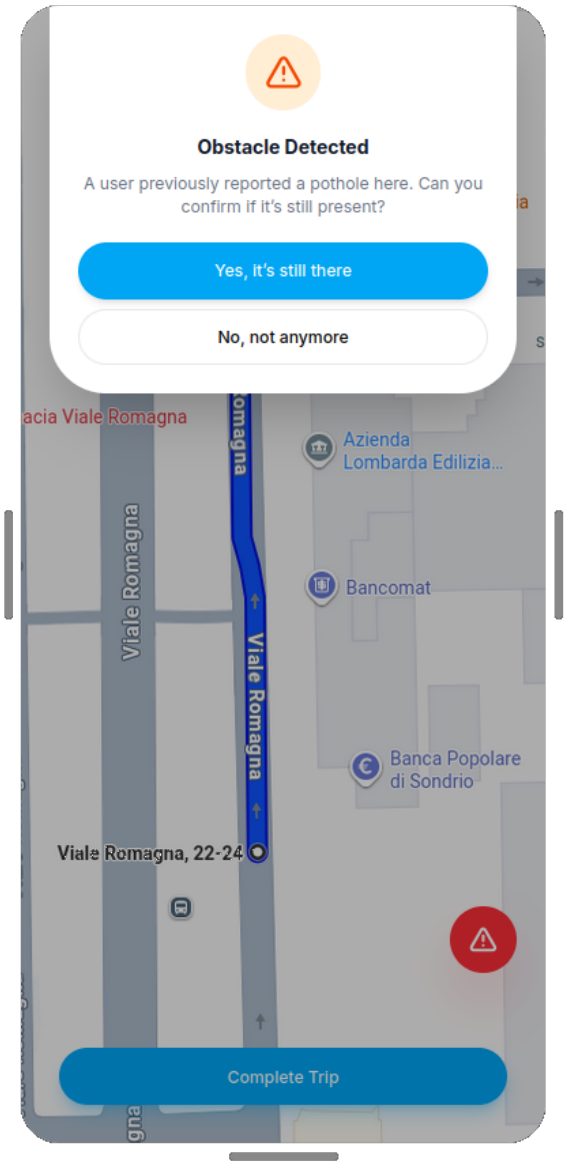

Figure 3.18: Report Confirmation Mockup

3.1.13. Path Creation

When the user selects the path creation feature, a pop-up form appears at the top of the screen, requesting the basic information needed to define a new bike path. The form includes Path Name, where the user enters the title of the new path, Description, an optional text field for adding details about the route, Visibility, allowing the user to choose whether the new path will be public or private. The user can also select the Creation Mode, which determines how the path will be constructed: either manually by drawing it on the map, or automatically by recording the GPS track during a ride.

Once the required fields are completed, the user may proceed by tapping Start Creating, which closes the dialog and starts the selected creation flow. If the user chooses not to continue, the pop-up can be dismissed at any time using the button in the upper-right corner.

Only logged-in users can create new paths, this feature is not available in guest mode.

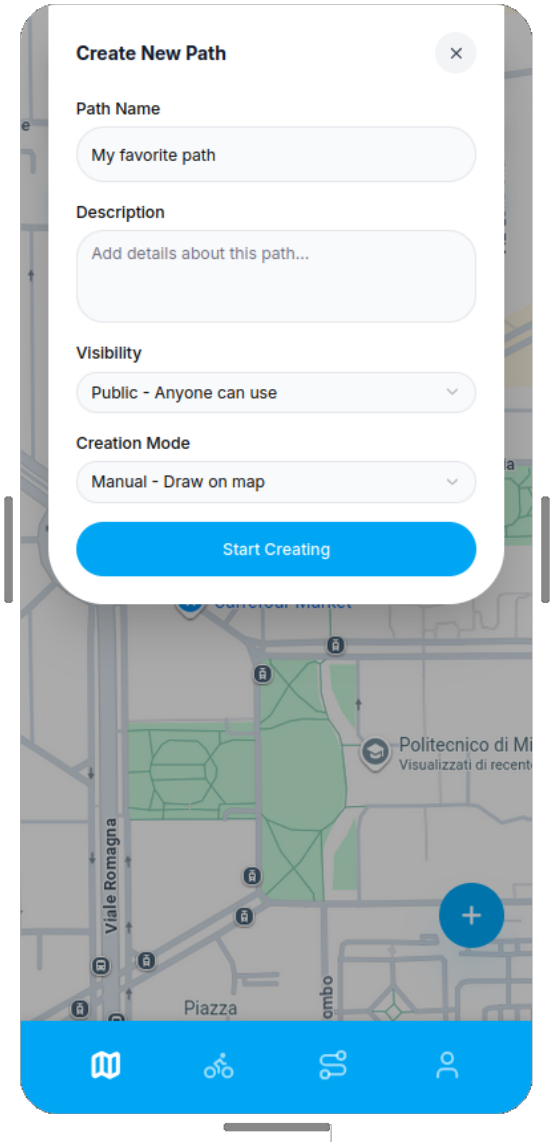

Figure 3.19: Path Creation Mockup

3.1.14. Creation View

When the user begins creating a new path, the app transitions into a dedicated creation view. The interface expands to a full-screen map and hides all navigation elements, allowing the user to focus entirely on constructing the path.

The content of this screen depends on the selected creation mode. In Automatic Mode, the map shows the user's real-time position as they cycle. As the user moves, the app draws a blue polyline representing the segment of the path already recorded. The line continuously updates to reflect the ongoing GPS trace. This view allows the user to visually monitor the path being generated as they ride.

In Manual Mode, the user can manually define the route by interacting with the map. Each waypoint placed by the user extends the blue polyline, progressively shaping the final path. This mode allows full manual control over the geometry of the route.

At any moment, the user may cancel the creation by tapping the button in the upper-left corner, which closes the creation view and discards the current progress.

When the user is satisfied with the path, they can save it by tapping the Save Path button at the bottom of the screen. This action stores the new path together with the metadata previously entered in the creation form.

Only logged-in users can access the creation view and save newly created paths.

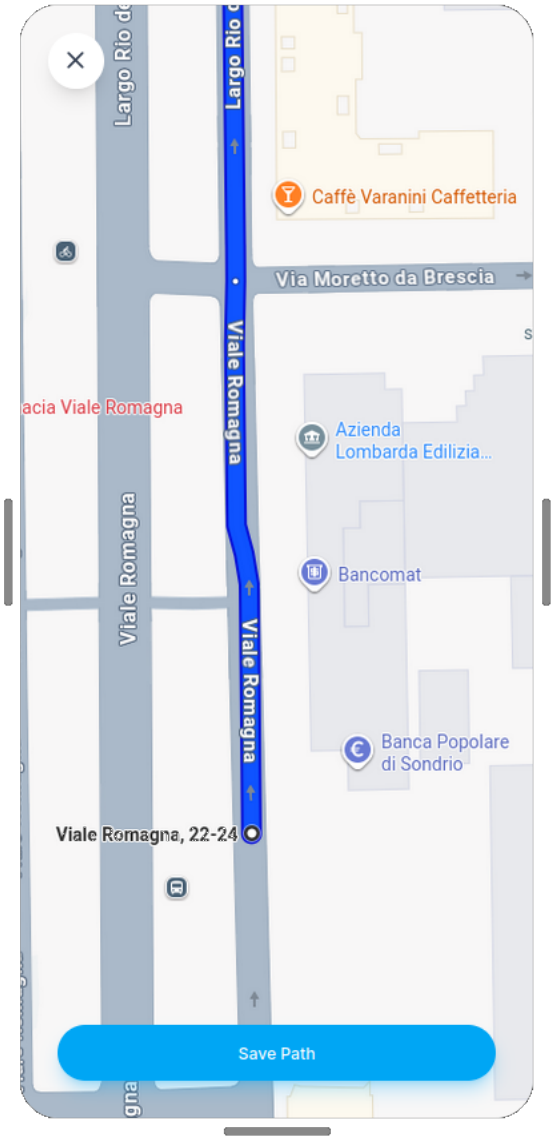

Figure 3.20: Creation View Mockup

3.1.15. Creation Completion

After the user completes the path creation process and saves the new route, the app displays a confirmation pop-up titled “Path Created!”. The dialog informs the user that the new path has been successfully stored and is now ready to use.

The pop-up includes a single action button, “Go to My Paths”, which redirects the user to the section containing all the paths they have created. From there, they can view the newly saved path in detail or manage their collection.

If the user taps outside the dialog, the pop-up simply closes and the app returns to the map view, allowing them to continue exploring the interface.

This confirmation appears only for logged-in users, as guests can’t create or save custom paths.

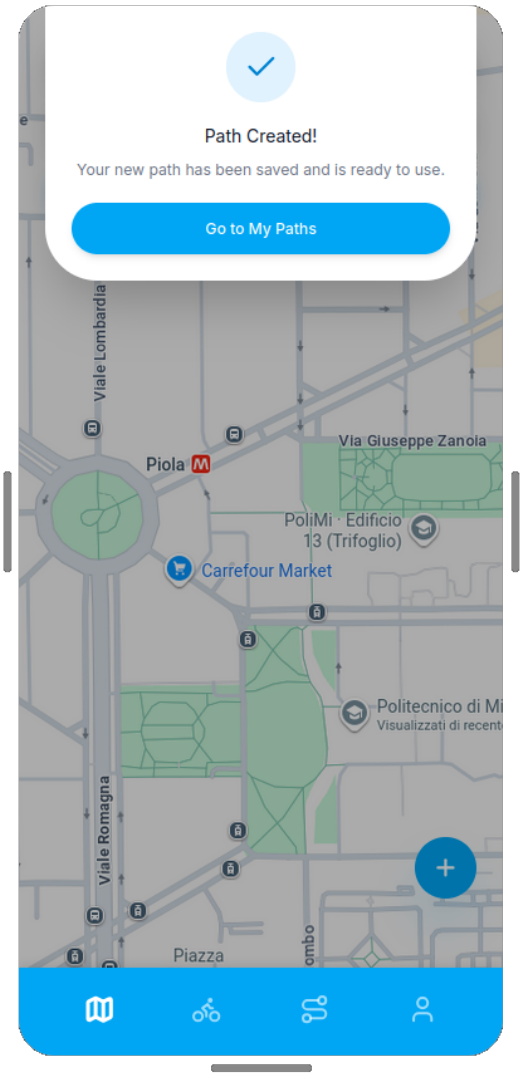

Figure 3.21: Creation Completion Mockup

3.1.16. Trip History Screen

The Trip History screen allows logged-in users to review all their previously recorded cycling activities and is accessible through the second icon in the bottom navigation bar. The top section includes a sorting control placed beside the screen title. Tapping it opens a compact dropdown menu that lets the user reorder the list of trips by date, distance, duration, or alphabetical order. Once a sorting option is selected, the list immediately updates to reflect the new ordering.

All trips are presented as compact cards showing essential information about each session, including the trip name, the total distance, the duration, and the date of completion. A delete icon also appears on each card, allowing users to remove a trip from their history.

When the user presses the delete icon, a confirmation pop-up appears to prevent accidental deletions. When the user taps a trip card, it expands smoothly into a detailed view. In this expanded state, the interface displays a map preview showing the route followed during the session, enriched with a weather badge positioned in the corner of the map. Beneath the map, a summary section reports the date, duration, and the name of the selected path, followed by a performance panel showing additional metrics such as distance, average speed, maximum speed, and elevation.

If the user taps the weather badge, the app reveals an additional panel containing detailed meteorological information, such as temperature, humidity, wind speed, visibility, pressure, and overall weather conditions at the time of the trip. When the recorded activity includes confirmed reports, the corresponding markers appear directly on the map. Selecting one of them opens a small dialog containing the details of the associated issue, allowing the user to inspect what was encountered along the route.

Tapping the trip header again collapses the view, returning the interface to the scrollable list of compact cards. This screen provides a rich and well-structured overview of past cycling activities, enabling users to explore their performance and recall the conditions of each session.

The Trip History is available exclusively to logged-in users, as guest users can't store or view past trips.

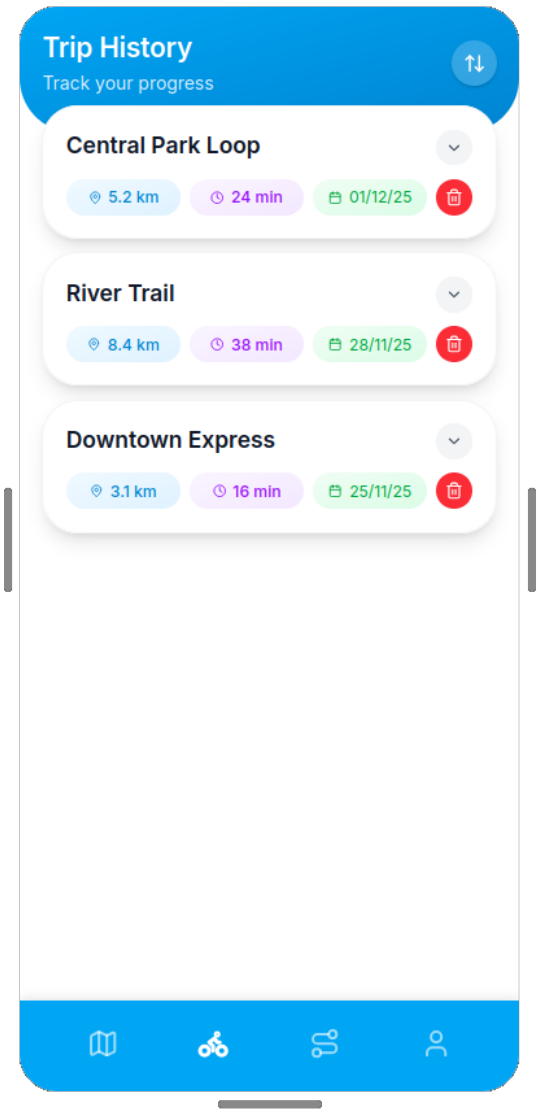

Figure 3.22: Trip History Screen Mockup

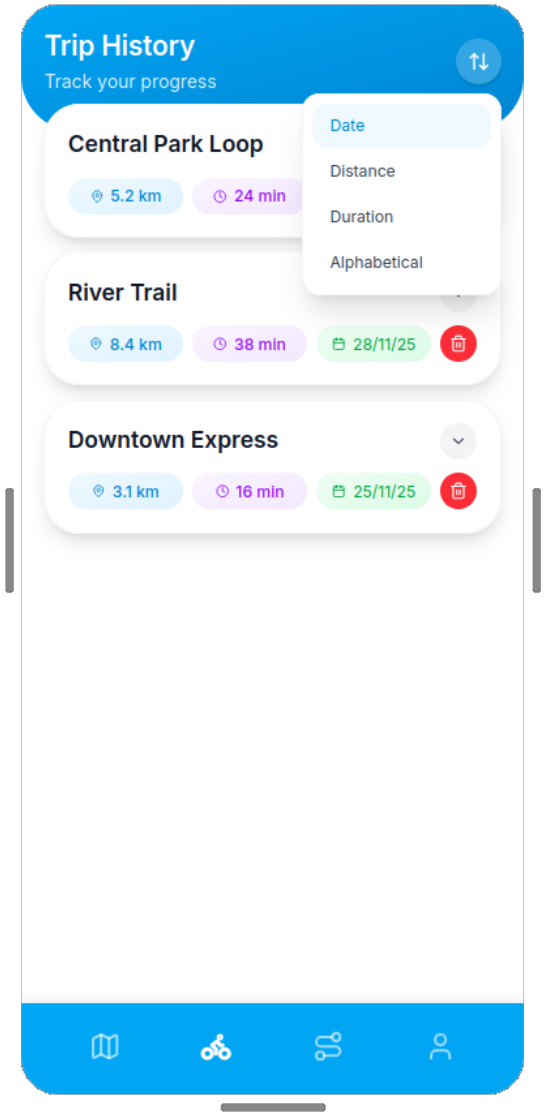

Figure 3.23: Trip Sorting Options Mockup

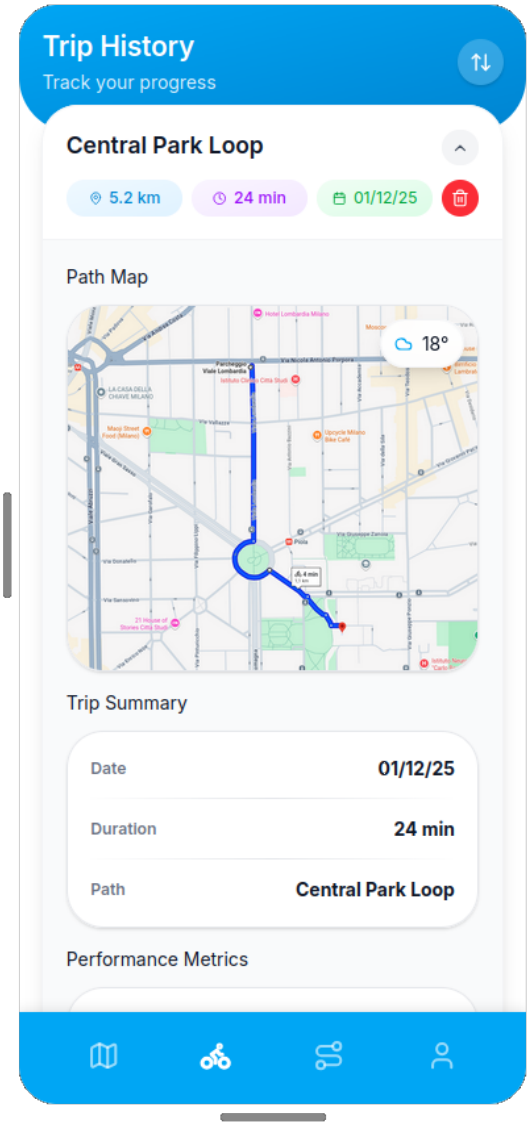

Figure 3.24: Trip History Details Mockup

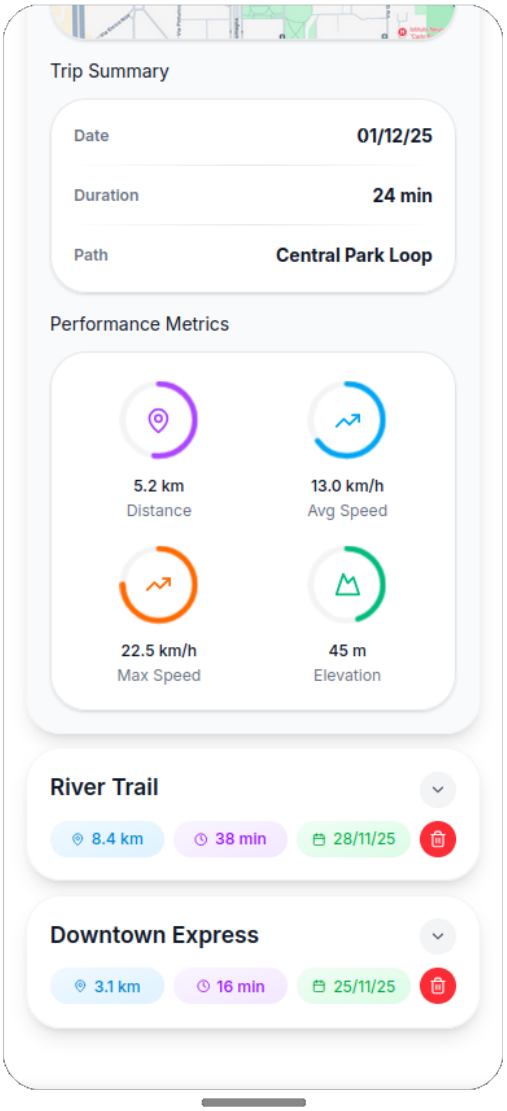

Figure 3.25: Trip Statistics Mockup

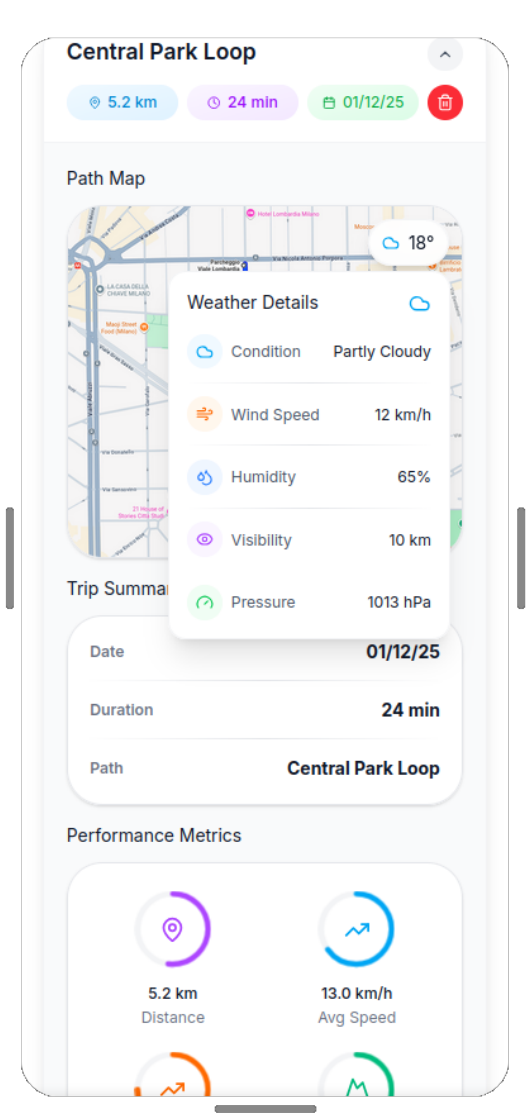

Figure 3.26: Trip Weather Details Mockup

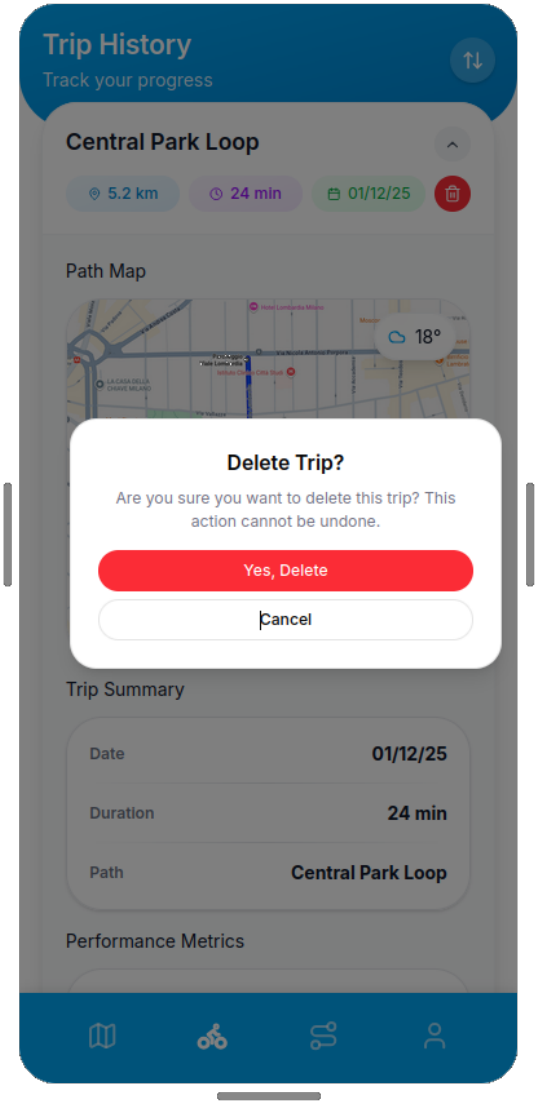

Figure 3.27: Trip Deletion Confirmation Mockup

3.1.17. My Paths Screen

The My Paths screen displays all custom bike paths created by the logged-in user and is accessible through the third icon in the bottom navigation bar. At the top of the page, a small arrow icon next to the screen title opens a compact sorting menu, allowing the user to reorder their paths by date, distance, alphabetical order, or visibility. Once an option is selected, the list is immediately re-sorted.

Each item in the list shows the path title, a short description, the total distance, the creation date, and two management icons: one for toggling visibility and one for deletion.

Tapping a path expands it and reveals a map preview showing the full route. When expanded, the entry also provides a Start This Path button, which redirects the user to the home screen with the selected path already loaded on the map. If the user's current position matches the path's starting point, the trip can be started immediately.

Managing a path triggers dedicated confirmation dialogs. Changing visibility opens a pop-up asking the user to confirm whether they want to switch the path between public and private. The change is applied only after confirmation. Deleting a path opens a separate confirmation dialog warning that the action is permanent, and the path is removed only if the user explicitly confirms.

Collapsing an expanded path restores the compact list layout, allowing users to browse their collection efficiently.

All interactions inside this screen are available exclusively to logged-in users. Guests can't view, create, start, or manage custom paths.

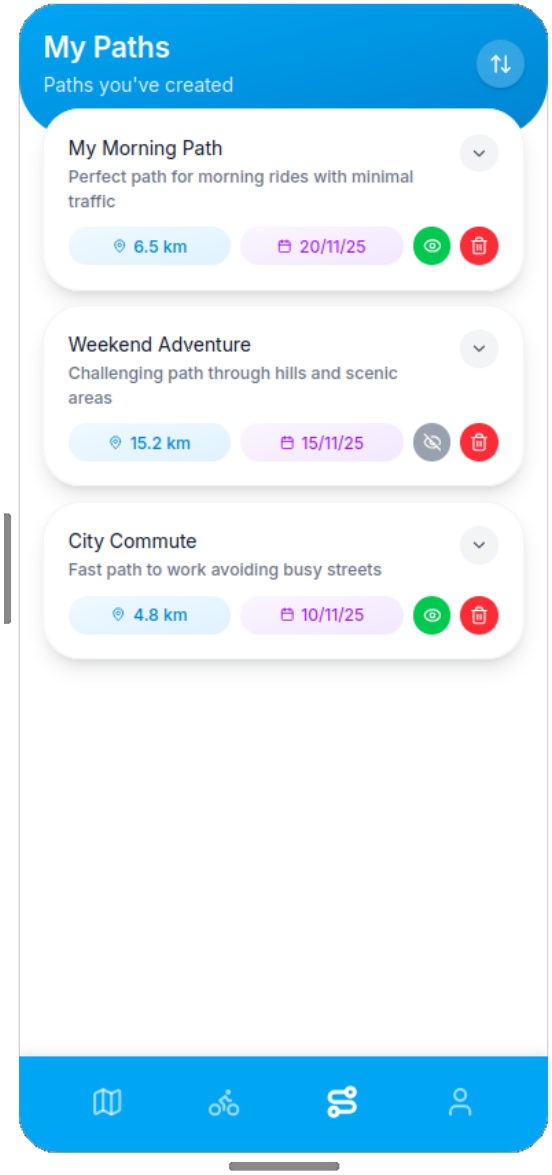

Figure 3.28: My Paths Screen Mockup

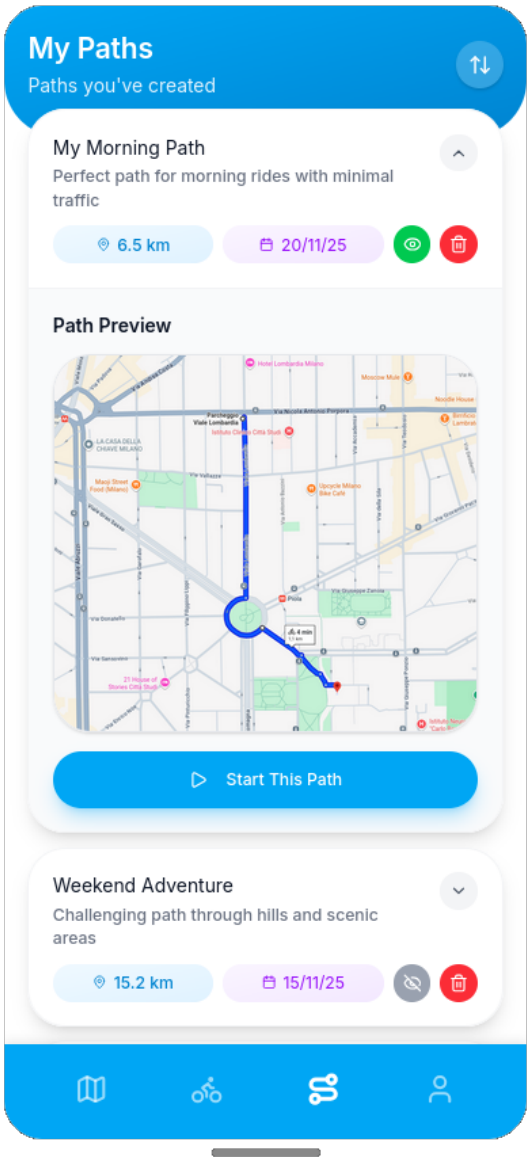

Figure 3.29: Path Details Mockup

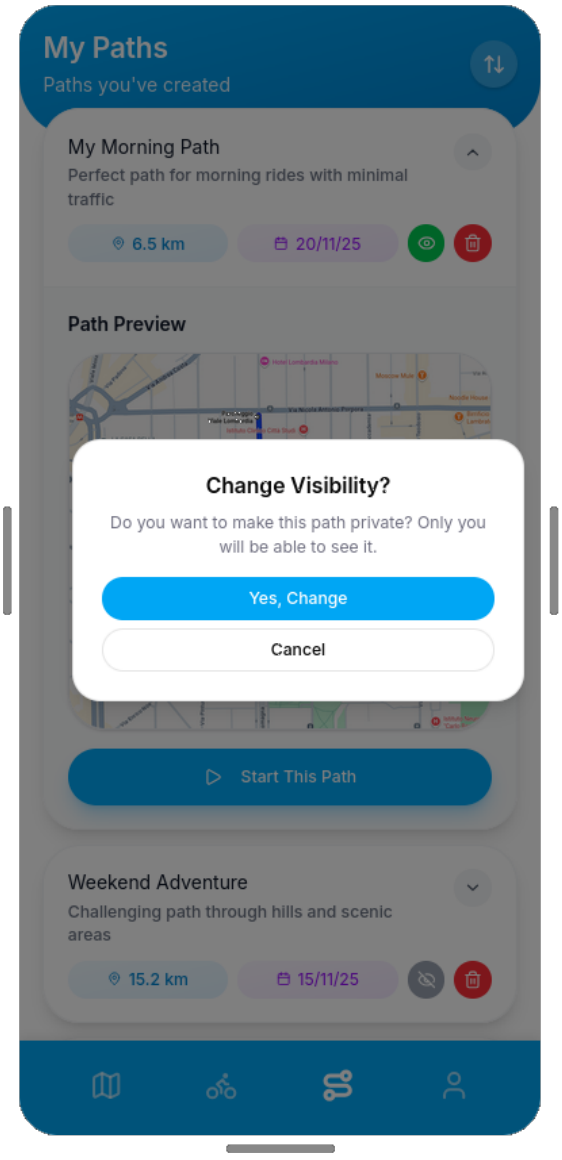

Figure 3.30: Path Visibility Toggle Mockup

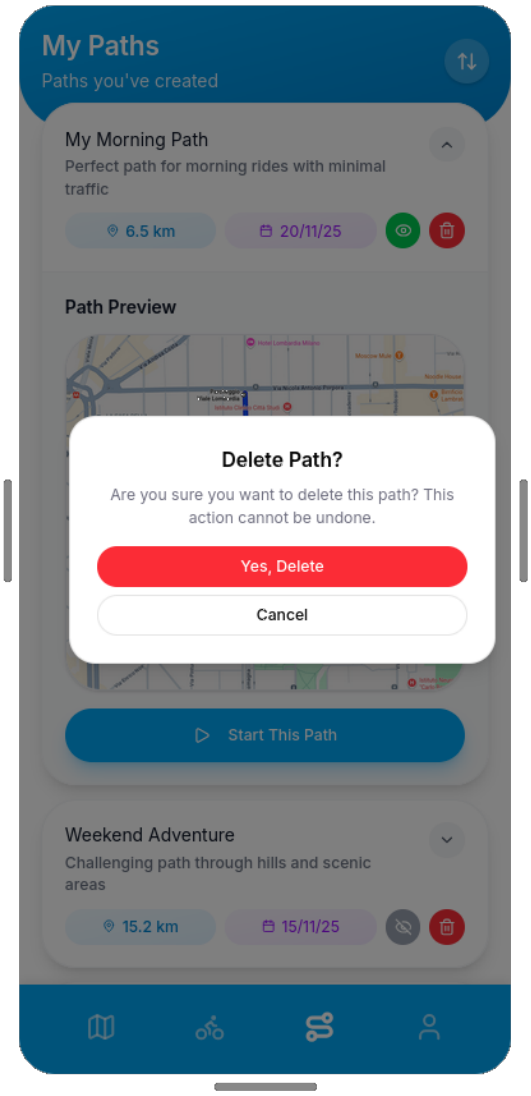

Figure 3.31: Path Deletion Confirmation Mockup

3.1.18. Profile Screen

The Profile screen provides logged-in users with a complete overview of their cycling activity and personal account details. It is accessible through the fourth icon in the bottom navigation bar.

At the top of the page, the user's profile card displays their avatar, name, and email address. A small edit icon next to the name allows the user to update their personal information, such as name or email. A settings icon is available in the top-right corner and redirects to the Settings screen, where account preferences and configuration options

can be adjusted.

Below the profile header, the screen shows a set of Overall Stats, summarising the user's total distance ridden, total number of recorded trips, and the total number of custom paths they have created. These metrics provide a quick snapshot of the user's long-term activity.

Further down, the Activity Stats section offers a more detailed breakdown. A compact dropdown menu allows the user to filter the statistics by time period, such as monthly activity. The metrics displayed include the number of paths and trips completed during the selected period, the total distance travelled, total riding time, average and maximum speed, elevation gain, calories burned, and other performance indicators. Each metric is represented with its own colourful icon for quick visual identification.

All information in this section is available exclusively to logged-in usersGuests cannot access the Profile screen or view personal statistics.

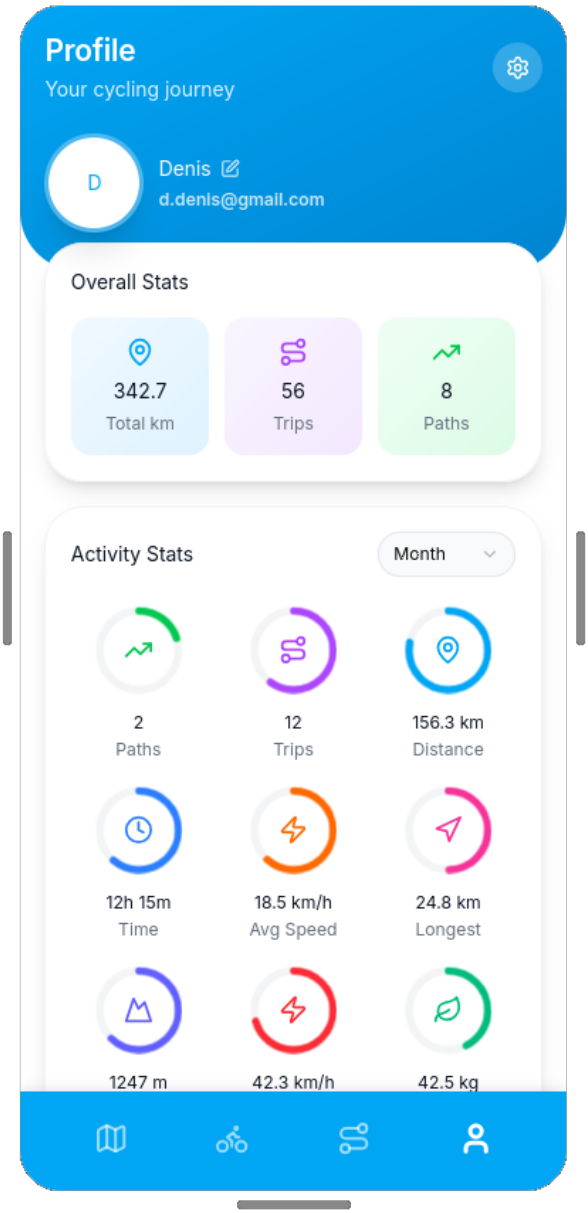

Figure 3.32: Profile Screen Mockup

3.1.19. Personal Information

The Personal Information screen allows logged-in users to update their account details. It includes editable fields for the Username and Email Address, followed by a dedicated section for changing the password.

To update the password, the user must enter their Current Password, then provide a New Password, and finally confirm it in the Confirm Password field. This ensures that the new credentials are correctly entered and match before proceeding.

Once all desired changes have been made, the user can tap Save Changes to submit the updated information. The app validates the input and, if successful, stores the new data and returns the user to the previous screen. A Cancel button is also available, allowing the user to dismiss the form without applying any modifications.

This screen is accessible exclusively to logged-in users and is reached from the Profile screen by tapping the pencil icon next to the user's name. The user can return to the Profile view at any time by tapping the back arrow in the top-left corner.

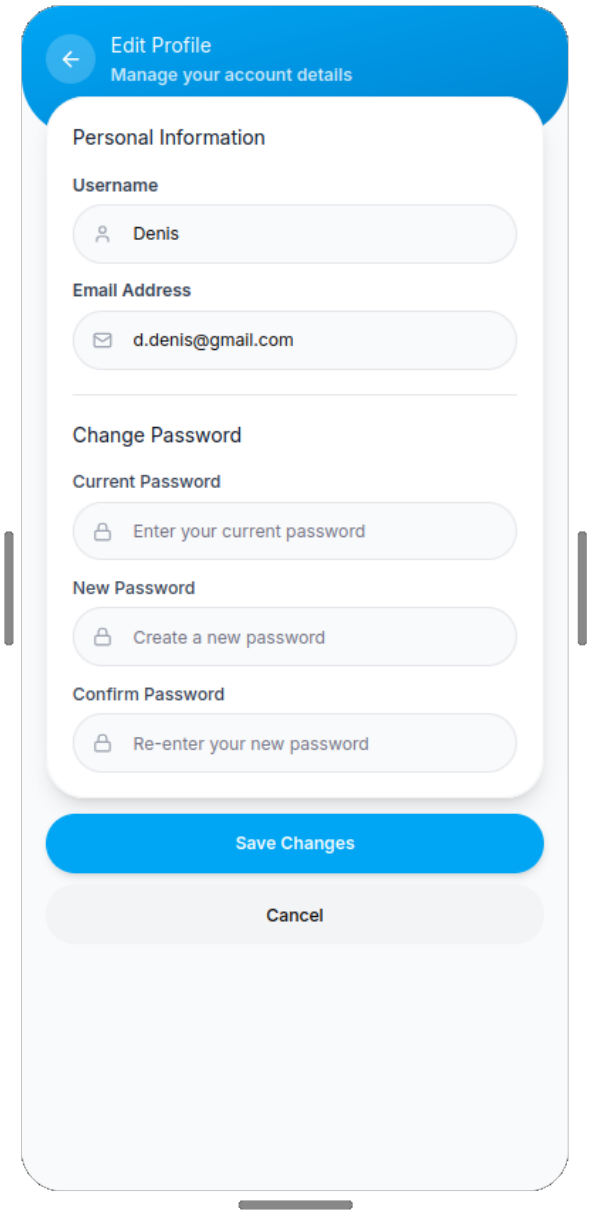

Figure 3.33: Personal Information Mockup

3.1.20. Settings Screen

The Settings screen allows logged-in users to customize their app experience and manage a small set of personal preferences. It is accessible from the Profile screen through the settings icon in the top-right corner.

At the top, the Appearance option lets the user choose the visual theme of the application. The current theme is displayed on the right side of the row, and tapping the selector opens a small menu where the user can switch between the available modes (e.g. Light or Dark).

Below it, the Default Privacy setting controls the visibility assigned to newly created paths. The user can choose whether new paths should be public or private by default. This preference applies automatically during path creation but can still be overridden manually for each individual path.

The Get Help section provides quick access to the support channel, allowing the user to contact the team if assistance is needed.

At the bottom of the screen, the Sign Out button logs the user out of their account and returns the application to guest mode.

This screen is available exclusively to logged-in users, as guest users do not manage appearance preferences, default visibility settings, or account actions.

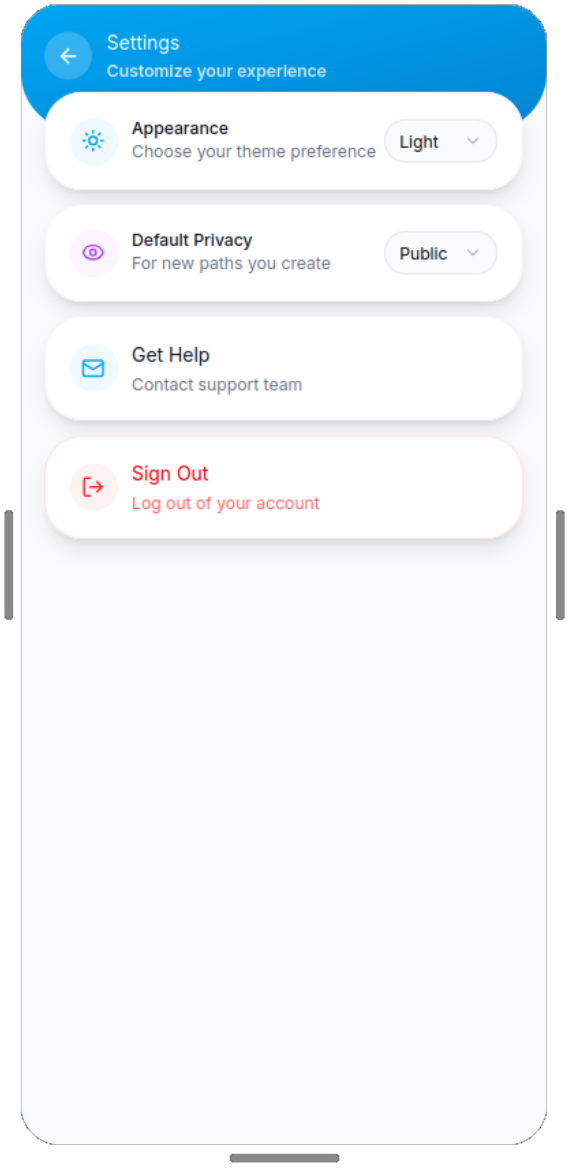

Figure 3.34: Settings Screen Mockup

3.1.21. Error Pop-ups

The app includes a global error-handling mechanism that provides consistent feedback whenever an unexpected issue occurs. In these cases, a dedicated Error pop-up appears at the top of the screen, displaying a clear title and a short description of what went wrong. The message shown in the dialog is dynamically replaced with the specific error text relevant to the situation, such as network failures, permission issues, or invalid operations.

The pop-up dims the underlying interface to draw attention to the alert and includes a single Close button that dismisses the dialog and returns the user to the previous screen.

Tapping outside the pop-up also closes the notification.

This error dialog can appear at any point in the app, including during path searches, trip management, report submission, navigation, or when modifying custom paths.

Figure 3.35: Error Pop-up Mockup

4 | Requirements Traceability

5 | Implementation, Integration and Test Plan

5.1. Overview

This chapter outlines the strategies adopted for the implementation, integration, and testing of the **Best Bike Paths (BBP)** platform. The primary objective is to ensure that the system meets the functional and non-functional requirements defined in the RASD document.

The chosen integration approach is **Bottom-Up**. This strategy involves developing and testing starting from low-level components (Data Layer and independent services) and then moving up towards more complex business logic components, finally reaching the presentation layer. This method allows for isolating and resolving bugs in the early stages, ensuring solid foundations for the higher-level modules.

5.2. Implementation Plan

The implementation of the BBP backend will follow a three-tier architecture (Data, Application, Presentation), developed iteratively.

5.2.1. Development Environment and Tools

Before starting the coding of modules, the development environment will be set up:

- **Version Control**: Use of Git with a feature-branch workflow.
- **DBMS Setup**: Configuration of the PostgreSQL instance and definition of relational schemas.
- **CI/CD**: Configuration of pipelines for automated building and testing.

5.2.2. Implementation Order

The implementation order of software components will be as follows:

1. **Data Access Layer**: Implementation of the `QueryManager` and database configuration.
2. **Core Services**: Development of `AuthManager` and `UserManager` for identity management.
3. **External Services Integration**: Implementation of the `WeatherManager` for communication with external weather APIs.
4. **Business Logic Services**: Sequential development of `PathManager`, `TripManager`, `ReportManager`, and `StatsManager`.
5. **Interface Layer**: Implementation of the `API Gateway` and definition of REST endpoints.
6. **Presentation Layer**: Development of the `Mobile App` (developed in parallel using Mock APIs in the initial stages).

5.3. Integration Plan

Component integration will strictly follow the Bottom-Up approach. Each phase involves integrating one or more modules into the existing subsystem, verifying their correct functioning through specific test drivers that simulate calls from higher levels.

The first step consists of integrating the `DBMS` with the `QueryManager`. Since all subsequent managers depend on data access, this subsystem constitutes the foundation of the architecture. The test driver will simulate query requests (CRUD) to verify the correct connection and manipulation of persistent data.

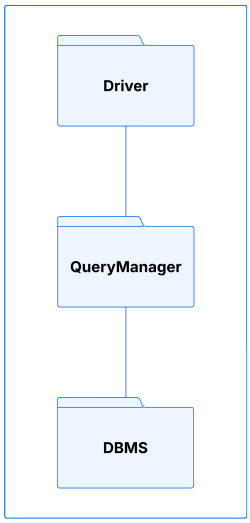

Figure 5.1: Step 1: DBMS and QueryManager Integration

The `AuthManager` and `UserManager` are integrated next. These components are fundamental to ensure that subsequent operations are performed by authenticated users. The driver will simulate registration, login, logout, and profile management flows, verifying that the `QueryManager` correctly persists credentials and user data and that token generation, validation, and error conditions (e.g., duplicate e-mails, invalid credentials) are handled consistently.

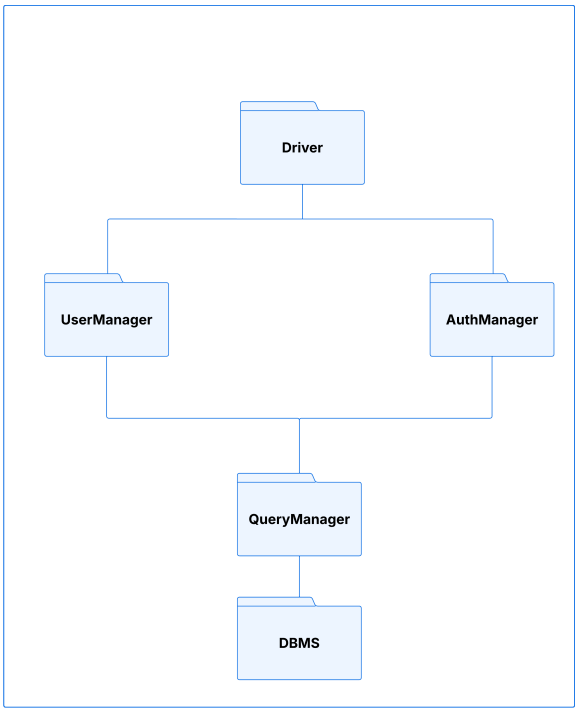

Figure 5.2: Step 2: AuthManager and UserManager Integration

In parallel, the `WeatherManager` is integrated. As a component that primarily interacts with an external Weather Service API and has minimal internal dependencies, it can be tested independently. The driver will simulate successful and failing API calls to verify the correct parsing of meteorological data, the handling of timeouts and HTTP errors, and the propagation of meaningful error codes when the external service is unavailable.

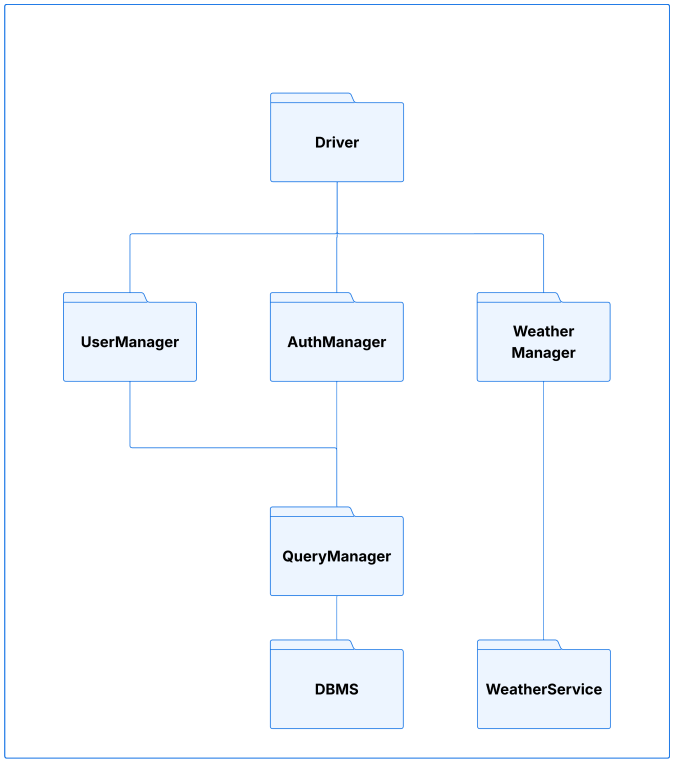

Figure 5.3: Step 3: WeatherManager Integration

We proceed with the integration of the `PathManager`. This component is crucial for BBP's core logic (path management). The driver will test the creation of new paths and the route calculation process, which relies on geospatial data retrieved via the `QueryManager`. Integration tests will also verify that the `PathManager` correctly computes and ranks candidate routes using stored path and report information, and properly handles corner cases such as missing or incomplete data.

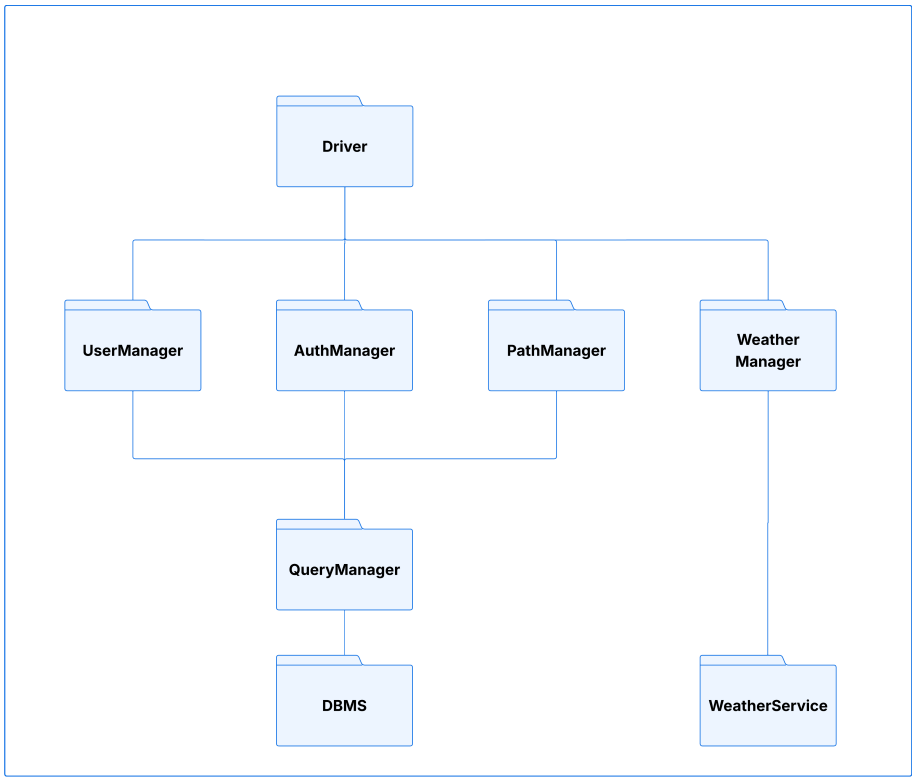

Figure 5.4: Step 4: PathManager Integration

The `TripManager` is added next. This module manages user trips and relies on both the `PathManager` and the `WeatherManager` to associate route and weather data with each trip. Integration tests will verify that raw trip samples and metadata are correctly stored, that each trip is linked to the selected path and corresponding weather snapshot, and that a complete trip summary is generated upon closure, even if the external weather service is temporarily unavailable.

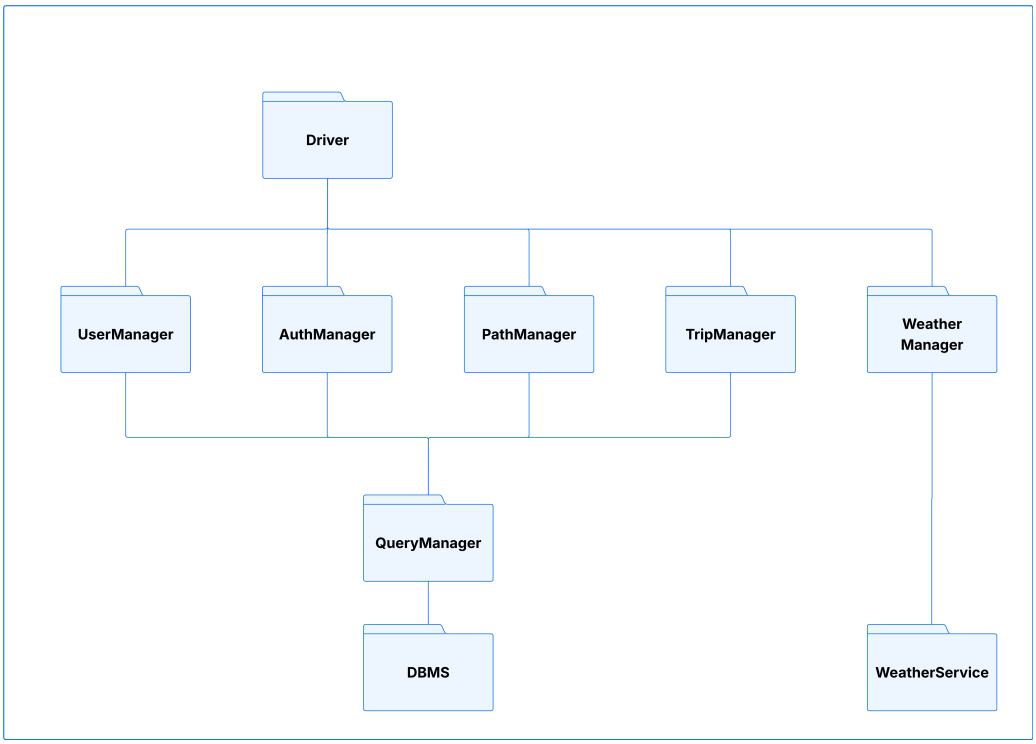

Figure 5.5: Step 5: TripManager Integration

After the `TripManager`, we integrate the `ReportManager`. This module allows users to report obstacles. Integration tests will verify that reports are correctly linked to existing users and paths, that attempts to create reports for non-existing entities and that updates and confirmations are properly propagated to the `PathManager`.

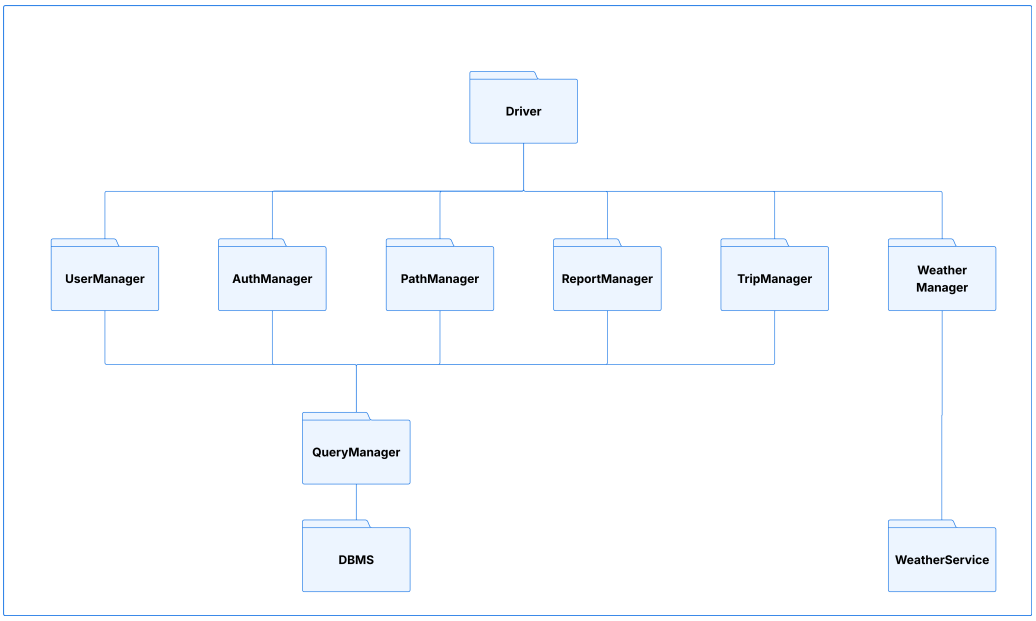

Figure 5.6: Step 6: ReportManager Integration

The backend system is completed with the addition of the `StatsManager`. This component aggregates data generated by the user's device to provide statistics. Being the final logical module, its integration allows testing complex data flows traversing the entire application domain. Integration tests will exercise end-to-end interactions involving trips, paths, reports, and confirmations to ensure that the computed aggregates are consistent with the underlying data and remain efficient on realistic data volumes.

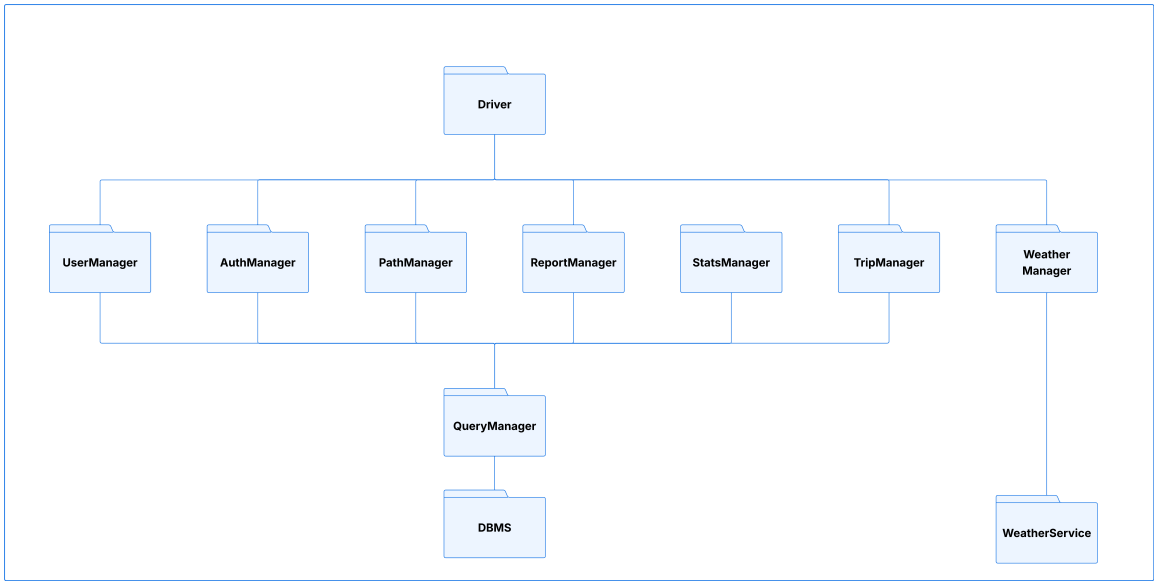

Figure 5.7: Step 7: StatsManager Integration (Complete Backend System)

The `API Gateway` is introduced, acting as the single entry point for the system. At this stage, the test driver no longer directly invokes individual Managers but sends REST HTTP requests to the API Gateway, which handles routing to the correct components (`Auth`, `User`, `Trip`, etc.).

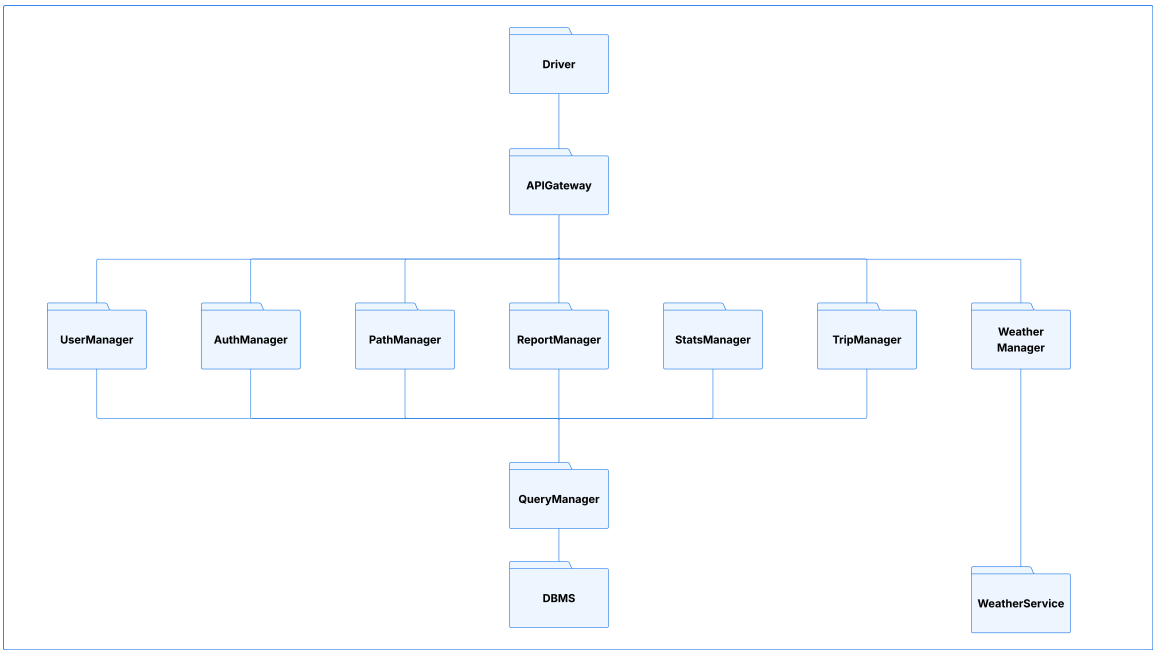

Figure 5.8: Step 8: API Gateway Integration

Finally, the test driver is removed and from now on the interaction is performed between the mobile device, the API Gateway, and all backend services, including real device sensors (GPS). At this final stage, integration tests will be executed end-to-end by exercising the main use cases from the mobile client, checking that interactions involving device sensors, the API Gateway, backend services, and the database behave as described in the Runtime View sequence diagrams.

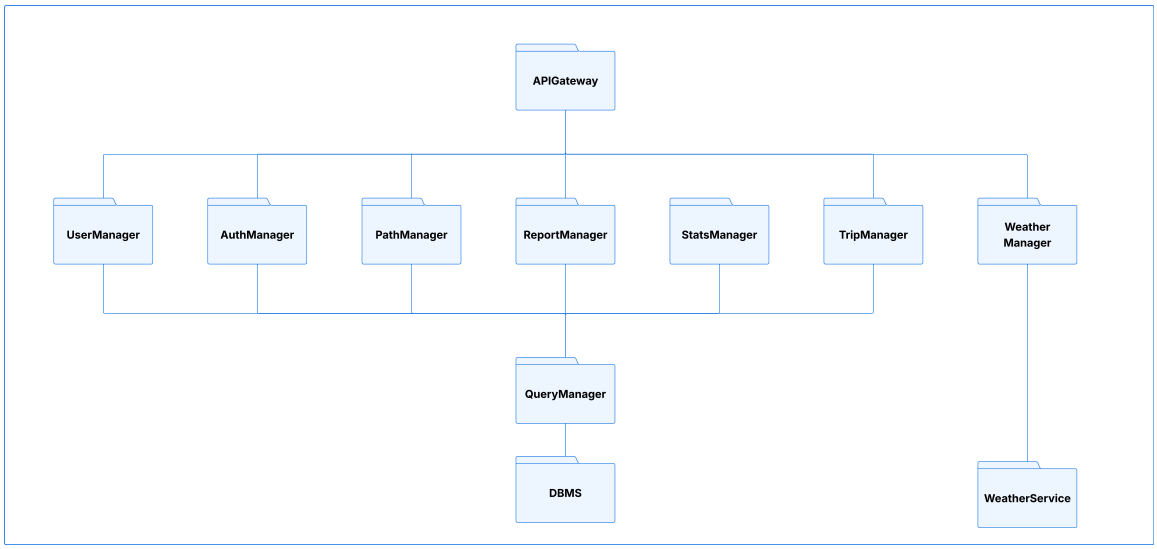

Figure 5.9: Step 9: Complete System Integration

5.4. Test Plan

Testing activities will be conducted during every development phase to ensure software quality. The testing strategy covers individual units, module interactions, and full system behavior.

5.4.1. Unit Testing

Unit testing is the first line of defense against software defects. In this project, we will employ Jest as our primary testing framework to isolate and verify the correctness of individual components, specifically the Managers and utility classes. The goal is to ensure that the internal logic of methods and classes functions exactly as intended before they interact with other parts of the system. For components that rely on external dependencies, such as the database or the external weather service, we will utilize Mock Objects. This approach allows us to simulate the behavior of these dependencies, ensuring that our tests remain focused, fast, and deterministic.

5.4.2. Integration Testing

Integration testing will be conducted incrementally following the steps described in Section 5.3. The primary objective of this phase is to verify that the interfaces between different components communicate correctly and that data flows seamlessly across module boundaries. We will focus on verifying the correct passing of parameters, the proper

handling of exceptions that may propagate between modules, and the referential integrity of data as it moves from the business logic layer to the persistence layer. By testing these interactions early and often, we can identify interface mismatches and communication errors that unit tests might miss.

5.4.3. System Testing

System testing will be executed on the complete system. This phase involves a rigorous validation of the application against the requirements specified in the RASD. Functional testing will cover all use cases to ensure that the user workflows, such as creating a trip, reporting an obstacle, or viewing statistics, behave as expected. Additionally, we will perform stress testing by simulating a high volume of concurrent requests to critical components like the `PathManager` and `TripManager`, ensuring the system remains stable under heavy load. Finally, performance testing will measure the response times of the API Gateway, with a particular focus on resource-intensive operations like route calculation and statistics retrieval, to guarantee a responsive user experience.

6 | Effort Spent

This section provides a breakdown of the number of hours each group member dedicated to completing this document. The work distribution is tracked per section and task.

Section	Ianosel Bianca	Simone Errigo	Vajihe Gholami	Total Hours
Introduction	0 hours	0 hours	0 hours	0 hours
Architectural Design	0 hours	0 hours	0 hours	0 hours
User Interface Design	0 hours	0 hours	0 hours	0 hours
Requirements Traceability	0 hours	0 hours	0 hours	0 hours
Implementation, Integration & Test	0 hours	0 hours	0 hours	0 hours
Final Review & Editing	0 hours	0 hours	0 hours	0 hours
Total Hours	**0 hours**	**0 hours**	**0 hours**	**0 hours**

Table 6.1: Time spent on document preparation

Bibliography

[1] ISO/IEC/IEEE. Systems and software engineering — life cycle processes — requirements engineering, 2018.

[2] M. Rossi, E. Di Nitto, and M. Camilli. Software engineering 2 rasd and dd assignment specification, Academic Year 2025/2026.

[3] M. Rossi, E. Di Nitto, and M. Camilli. Slides of the software engineering 2 course. WeBeep platform, Academic Year 2025/2026.

List of Figures

List of Tables