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: 07.01.2026
Project Link: Errigo-Gholami-Ianosel (github.com)

Contents

1 Introduction

In recent years, cycling has gained increasing relevance as a sustainable, healthy, and efficient means of transportation. Despite its benefits, cyclists often face difficulties in identifying safe and reliable routes. The **Best Bike Paths (BBP)** system has been conceived to address these challenges by using community-contributed data and sensor information to help users choose safer and more reliable cycling routes.

1.1. Purpose

The purpose of this document is to translate the system requirements defined in the **Requirement Analysis and Specification Document (RASD)** into architectural, component-level, and interaction design choices. This document provides guidance for developers and testers by defining a structured approach to the design, integration, and validation of system components, ensuring alignment with the project goals and user requirements.

1.2. Scope

The scope of the **Best Bike Paths (BBP)** platform is to create and maintain a comprehensive inventory of bike paths by supporting users in recording, analyzing, and sharing cycling activities and path conditions. It allows logged-in users to record their trips, automatically collecting GPS and sensor data or manually inputting information, which is then analyzed to provide statistics and identify obstacles. Crucially, BBP incorporates a mechanism for validating and merging this user-reported data, ensuring the quality and freshness of path information before making it publishable. Finally, the system enables all users, logged in or not, to search for the best-scored path between two points, leveraging the validated, community-contributed inventory to optimize their cycling routes.
This document focuses on the architectural and component-level design of the system, describing how its elements interact and cooperate to satisfy the requirements defined for the platform. The complete and formal definition of system requirements is provided in the **RASD**.

1.3. Definitions, Acronyms, Abbreviations

1.3.1. Definitions

- **HTTPS:** Secures all communication between the app and the server using encryption.
- **REST API:** The set of HTTP-based interfaces exposed by the backend service, used by the mobile client to exchange data in JSON format.
- **DAO Pattern:** A design pattern that isolates data access logic from the core business logic, improving modularity and maintainability.

1.3.2. Acronyms

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

1.4. Revision history

- Version 1.0 (07 January 2026);

1.5. Document Structure

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

1. **Introduction:** provides an overview of the document, outlining its purpose, scope, and relevance to the project.
2. **Architectural Design:** explains the technical architecture of the system, including the components, layers, and their interactions.
3. **User Interface Design:** details the visual and functional design of the user interfaces, emphasizing usability and user experience.
4. **Requirements Traceability:** maps the requirements defined in the RASD to the architectural components and design decisions presented in this document.
5. **Implementation, Integration and Test Plan:** describes the approach to system development, integration strategies, and the planned testing methods to ensure quality and reliability.
6. **References:** lists all sources (including Software) and documents used in the preparation of this document.
7. **Effort Spent:** summarizes the distribution of time and effort invested by each team member in completing the document and its sections.

2 | Architectural Design

2.1. Architectural Overview: High-level components and their interactions

The architecture of the Best Bike Paths (BBP) system follows a classic **three-tier structure**, separating the software into a presentation layer, an application 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 HTTPS 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 application layer. The app interacts with device-level subsystems such as the GPS module and external sensors (when available), elements that 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 application 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.
- **TripManager**: handles the lifecycle of a cycling trip and produces trip summaries enriched with contextual weather data, trip statistics, and reports encountered during the trip.
- **PathManager**: responsible for creating paths, computing routes, ranking candidate paths according to their condition and effectiveness, and managing visibility and deletion.
- **ReportManager**: responsible for storing and aggregating reports, retrieving reports by path, managing confirmations, and updating path-condition indicators.
- **StatsManager**: computes and stores user statistics and per-trip metrics, including overall aggregations.
- **QueryManager**: centralises database access and exposes CRUD-style operations to all Managers, enforcing a uniform data-access interface.

The application layer also relies on supporting services that encapsulate third-party integrations:

- **WeatherService**: adapter for the external weather provider used to fetch and aggregate meteorological data for trip summaries and history retrieval.
- **GeocodingService**: integrates with an external geocoding provider to resolve user-provided origins and destinations into coordinates for path search.
- **SnappingService**: maps raw segments to road geometry to improve path accuracy during manual path creation before storage and analysis.

All modules are accessed through the **API Entrypoint**, 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 application logic and reduces duplication across components.

2.2. Component View

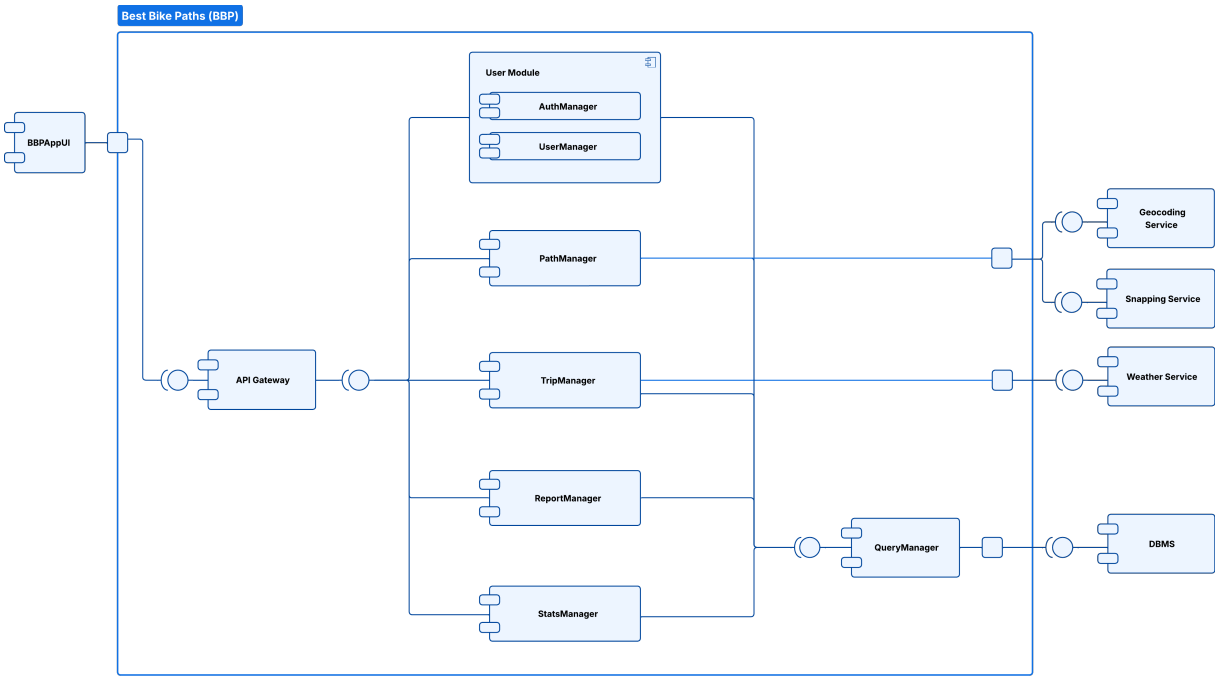

Figure 2.1: Component View Diagram

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.

API Entrypoint

The **API Entrypoint** represents the single entry point for all interactions between the BBP mobile app and the backend. It is a logical layer implemented inside the backend application. It centralizes request routing toward the Managers, applies authentication middleware on protected endpoints, performs input validation at the boundary, and translates domain errors into HTTP responses.
The API Entrypoint exposes the following logical sub-APIs:

- **AuthAPI**: endpoints for login, token refresh, and token validation.
- **UserAPI**: endpoints for registration and profile retrieval.
- **TripAPI**: endpoints for creating, listing, and deleting trips.
- **PathAPI**: endpoints for route computation and retrieval of path metadata.
- **ReportAPI**: endpoints for submitting and retrieving obstacle reports.
- **StatsAPI**: endpoints for retrieving user statistics and trip metrics.

User Module

The **User Module** groups two Managers that collectively handle all user-related flows exposed by the backend, including registration, login, logout, token refresh, and profile management. It centralises credential checks, uniqueness constraints, and the enforcement of authentication rules at the application layer.

- **AuthManager**, responsible for credential verification, access/refresh token issuance and rotation, and session invalidation (logout). Refresh tokens are persisted and revoked through the **QueryManager**.
- **UserManager**, responsible for user registration and profile updates, including validation of editable fields and prevention of duplicate email/username.

Both Managers use the **QueryManager** for data retrieval and persistence, while the **API Entrypoint** applies authentication middleware to protect profile endpoints.

TripManager

The **TripManager** coordinates the lifecycle of a cycling session. GPS samples and live tracking stay on the mobile app during the trip; the backend is contacted only at stop time. Upon trip completion, the app sends the final payload containing origin/destination, timestamps, and trip segments. The TripManager persists the trip data through the **QueryManager** and enriches the stored record with environmental data retrieved through the **WeatherService** and metrics computed by the **StatsManager**. When a user retrieves their trip history, the TripManager also triggers on-demand enrichment: it requests missing weather data, statistics, and associated reports through the **ReportManager**, then persists any newly computed values.

PathManager

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. For path search requests, it resolves textual origins and destinations through the **GeocodingService** before querying the database. It also manages path creation in both manual and automatic modes, updates visibility, and handles deletion requests. When the client draws a manual path, the PathManager invokes the **SnappingService** to align coordinates to the road network. It also is responsible for creating new paths, either from user-drawn segments or from GPS samples recorded during automatic path creation. For both modes, it interacts with the **SnappingService** to snap drawn paths to the road network. This service exposes the routing logic to the API Entrypoint and interacts with the **QueryManager** to retrieve pre-aggregated segment conditions alongside path data needed for route computation.

ReportManager

The **ReportManager** 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 **PathManager** when updated conditions affect route evaluation. The **TripManager** may also invoke the ReportManager during trip history retrieval to enrich trip summaries with associated reports.

StatsManager

The **StatsManager** optimizes system performance by using a state-aware recomputation strategy that minimizes redundant processing. For overall statistics, it compares the user's current total trip count against the value stored from the last calculation; it only triggers a full aggregation of data if the **QueryManager** reports a change in the trip count, otherwise it simply retrieves the previously stored results.
For individual trip statistics, the manager utilizes a lazy initialization pattern by asking the **QueryManager** if a processed record already exists for a specific trip ID. If the statistics are found, they are returned immediately; if not, the manager computes and persists them for the first time. This dual-layered approach ensures that data is only processed when necessary, effectively eliminating double computations by relying on the **QueryManager** to track the state of processed data. The **TripManager** calls the StatsManager at trip finalization and also on demand when trip history retrieval detects missing statistics.

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.

WeatherService

The **WeatherService** wraps calls to an external provider and aggregates meteorological data by sampling coordinates along a trip route, including path segment polylines and origin/destination when available. The resulting snapshot is returned to the **TripManager**, which stores it through the **QueryManager**. The TripManager also reuses the WeatherService during trip history retrieval when stored data is missing.

GeocodingService

The **GeocodingService** wraps calls to an external provider to translate location strings into coordinates. It returns resolved coordinates to the **PathManager** and isolates third-party integration concerns.

SnappingService

The **SnappingService** interfaces with an internal routing engine to snap user-drawn paths to the road network, ensuring accurate route representation. It returns the adjusted path to the **PathManager**.

2.3. Deployment View

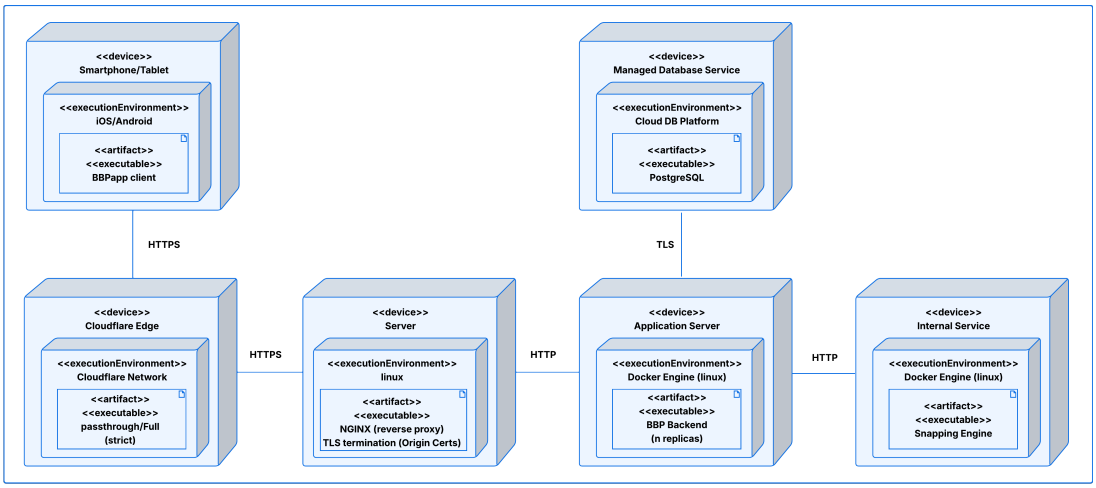

Figure 2.2: Deployment View of the BBP System

The deployment view describes the hardware and software infrastructure supporting the *BBP* system. Each tier is deployed on logically separated infrastructure components 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. While the client executes local presentation logic, no security-critical or authoritative business logic is enforced at this level. The devices capture user input and location, display results, and forward authenticated or unauthenticated HTTPS requests toward the Application Tier.

- **Application Tier:** this tier is responsible for handling incoming traffic, enforcing security, and executing the core business logic of the system. External requests first pass through the *Cloudflare Edge* and then reach a shared *NGINX* reverse proxy

running on the same VPS as the application containers. NGINX is the only component exposing ports **80/443**, terminates TLS using Cloudflare Origin Certificates, applies basic filtering and rate limiting, and forwards traffic to the backend containers over a private Docker network using HTTP. The backend application itself runs in one or more stateless Docker containers (replicas), allowing horizontal scaling. The snapping engine runs as a separate internal container and is reachable only over the private Docker network. Since authentication tokens are included in each request header, no server-side session state is maintained, enabling horizontal scaling and dynamic replica management.

- **Data Tier:** the Data Tier consists of a managed PostgreSQL database. This database stores all persistent system data such as user information, paths, trips, and analytics. Backend servers connect to the managed service over the network. 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.

2.4. 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, 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 Entrypoint**, which routes each request to the appropriate **Manager**. Persistence operations are centralised in the **QueryManager**. When required, Managers also interact with dedicated **Services** to complete the requested operation.
The sequence diagrams that follow cover the core use cases and show how responsibilities are distributed at runtime.

[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: username, email, and password. 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 Entrypoint**, which forwards it to the **UserManager**. The **UserManager** checks whether the email is

already in use by querying the DBMS through the **QueryManager**. If the email already exists, the system returns an error and the mobile app notifies the user.
If the email is not already in use, the **UserManager** stores the new user in the database (with the password handled securely). The flow ends successfully with a new user created.

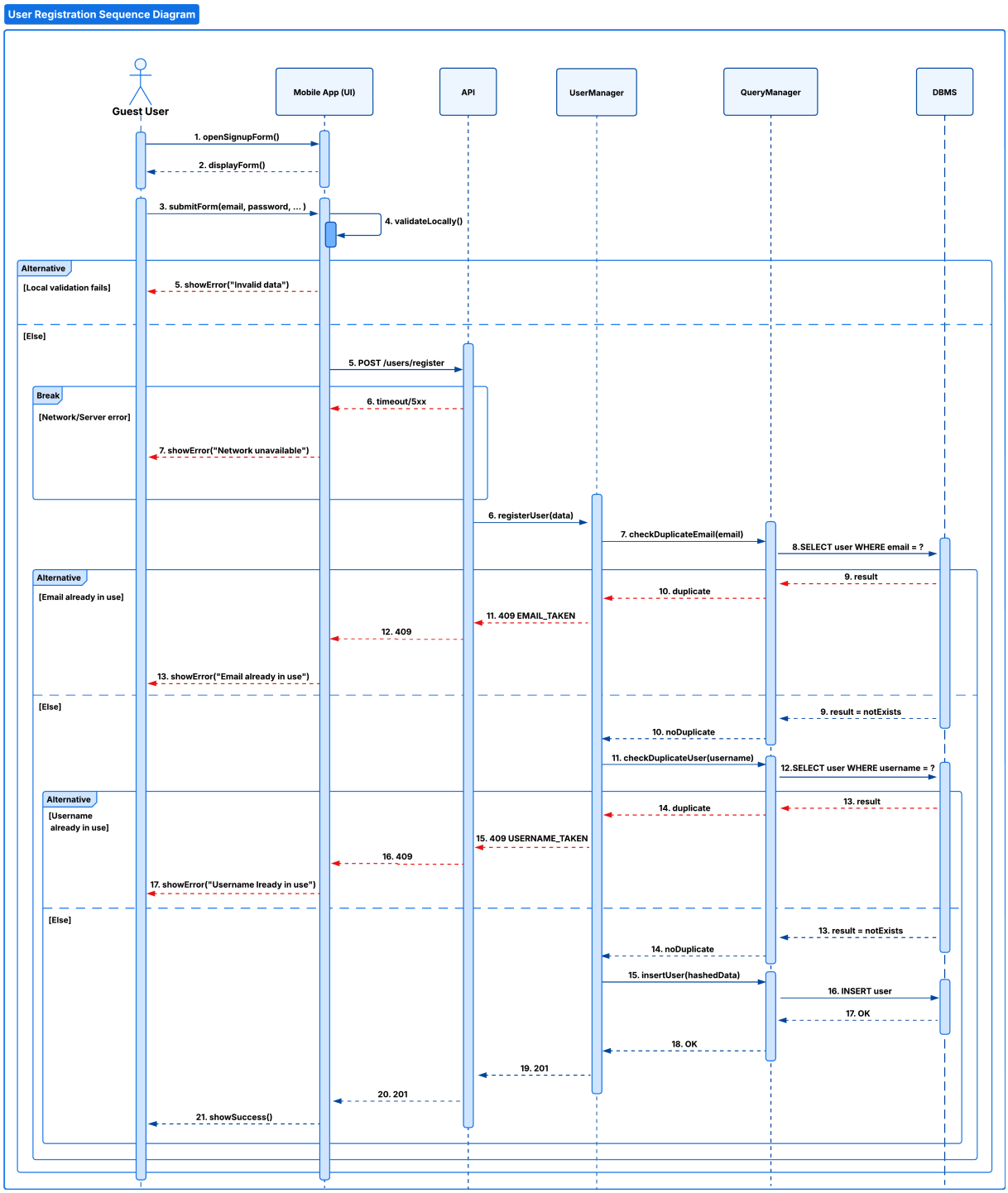

Figure 2.3: User Registration Sequence Diagram

[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 Entrypoint**.
The **API Entrypoint** 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** 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** response and the corresponding error message on the client.
When the credentials are correct, the **AuthManager** generates an **access token** and a **refresh token**, stores the refresh token in the **DBMS**, and returns both tokens together with the user information to the mobile app. The app stores the tokens 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.

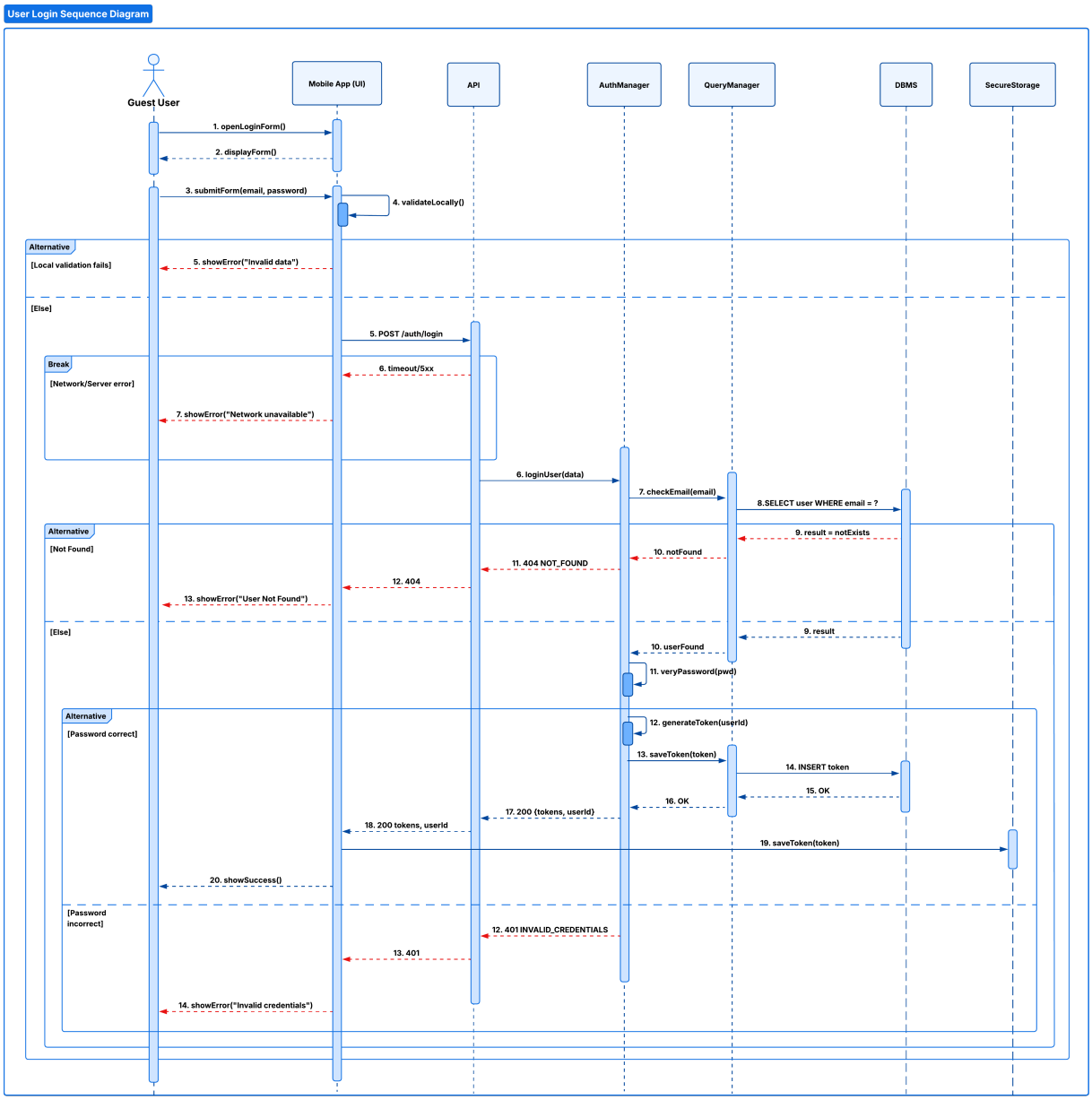

Figure 2.4: User Log In Sequence Diagram

[UC3] - User Log Out

When a logged-in user initiates the logout operation from the mobile app, the client first removes the locally stored tokens from the device's **secure storage**. This ensures the user is logged out on the client side even if the network request fails.

After clearing local data, the mobile app sends an HTTP request to the logout endpoint exposed by the **API Entrypoint**. The **API Entrypoint** forwards the request to the **AuthManager**, which invalidates the current session by deleting the corresponding **refresh token** through the **QueryManager**. The **QueryManager** executes a **DELETE** operation on the database to remove the stored token.

If the operation succeeds, the server returns a **204 NO_CONTENT** response. If a network or server error occurs (timeout or **5xx**), the local logout remains effective and the server revocation can be retried later.

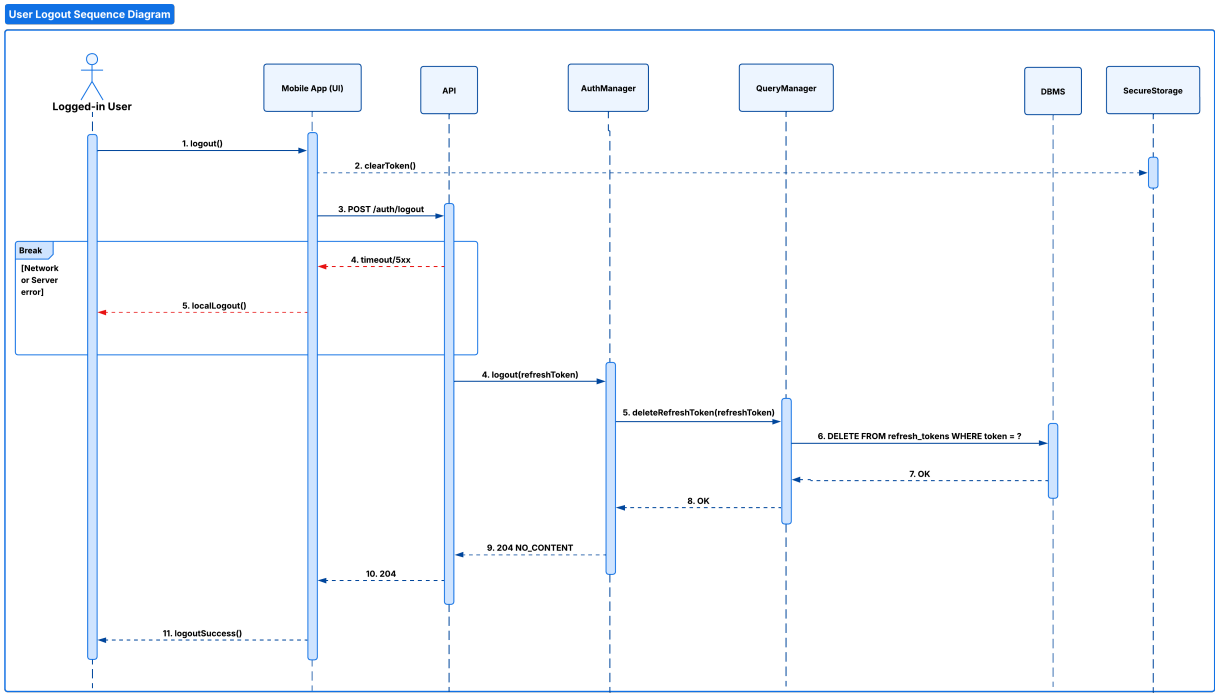

Figure 2.5: User Log Out Sequence Diagram

[UC4] - 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 with a **PATCH** request to the **API Entrypoint**, which delegates the request to the **UserManager**. The diagram illustrates the email update flow, but the same sequence applies to username changes as well. Before applying the update, the **UserManager** checks for duplicate email or username values to prevent conflicts.

The update is forwarded to the **QueryManager**, which issues the corresponding **UPDATE** operation on the database. If the update succeeds, the backend returns a confirmation message.

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

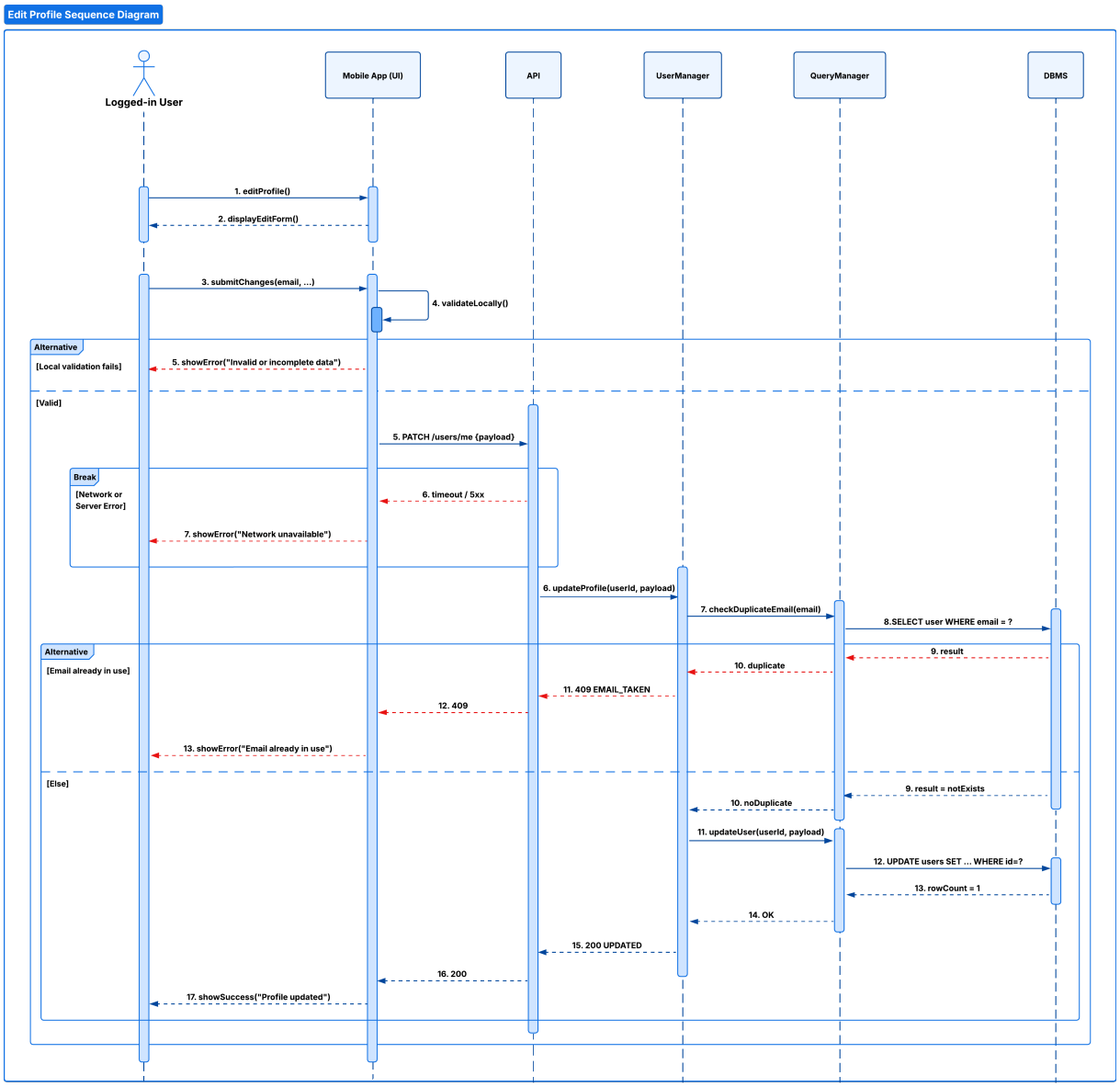

Figure 2.6: Edit Personal Profile Sequence Diagram

[UC5] - Search for a Path

A user searches for bike paths between two locations by providing a starting point and a destination through the mobile app. Before contacting the backend, the app performs a **local validation** to ensure that both fields are not empty. If the validation fails, the user is immediately notified.
When the input is valid, the mobile app sends a search request to the backend through the **API Entrypoint**. The API Entrypoint forwards the request to the **PathManager**, which is responsible for path computation. The **PathManager** first resolves the origin and destination through the **GeocodingService** to obtain coordinates. It then retrieves the relevant path segments and related information from the database by interacting with the **QueryManager**. Once the data is available, it computes the best path or set of paths according to the system criteria and user constraints.
If one or more valid routes are found, the **PathManager** returns the results to the **API Entrypoint**, which responds to the mobile app with a **200 OK** message containing the suggested paths. The app then displays the available routes to the user. If no suitable route can be computed, the **PathManager** signals a **NO_ROUTE** condition, resulting in a **404** response that is propagated back to the mobile app, which notifies the user accordingly.
In the event of network failures or server-side errors, the interaction is interrupted and the mobile app shows a generic "Network unavailable" error message.

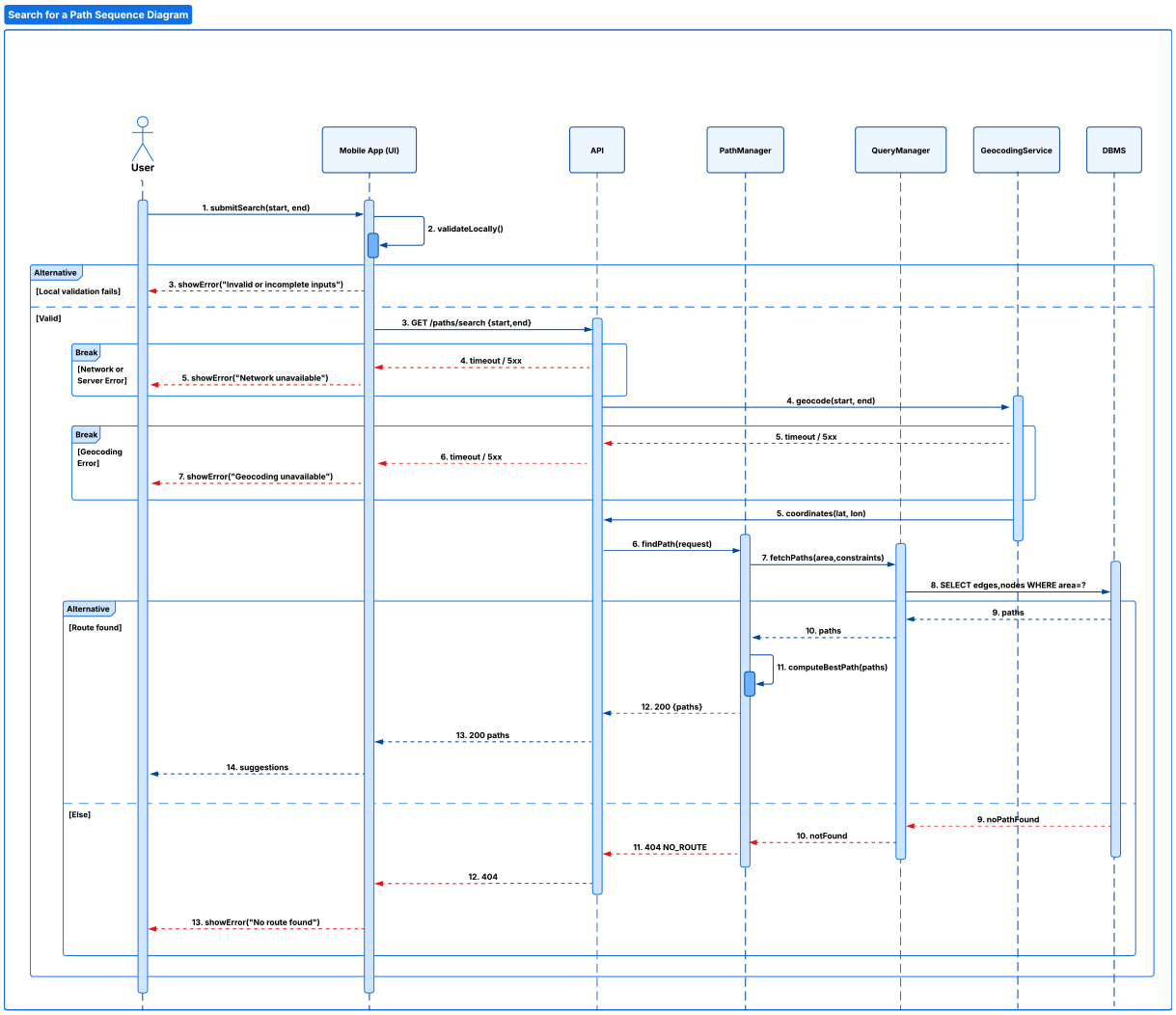

Figure 2.7: Search for a Path Sequence Diagram

[UC6] - Select a Path

After receiving a list of suggested routes, the user can select a specific path to see its details on a map, and inspect its associated reports. When the user selects a path, the mobile app sends a request containing the selected path identifier to the backend through the **API Entrypoint**.
Path details (e.g., segments and status) are obtained during the search phase, while reports are refreshed every time a path is selected to ensure the information is always up to date.
The **API Entrypoint** forwards the request to the **ReportManager**, which retrieves the corresponding reports by querying the database through the **QueryManager**.
If reports are found, the ReportManager returns them to the API Entrypoint, which responds with a **200 OK** status and forwards the information to the mobile app. The app then displays the list of reports to the user.
If no report is available for the selected path, the backend returns an empty list with a **200 OK** response. Network or server failures are handled consistently with other interactions.

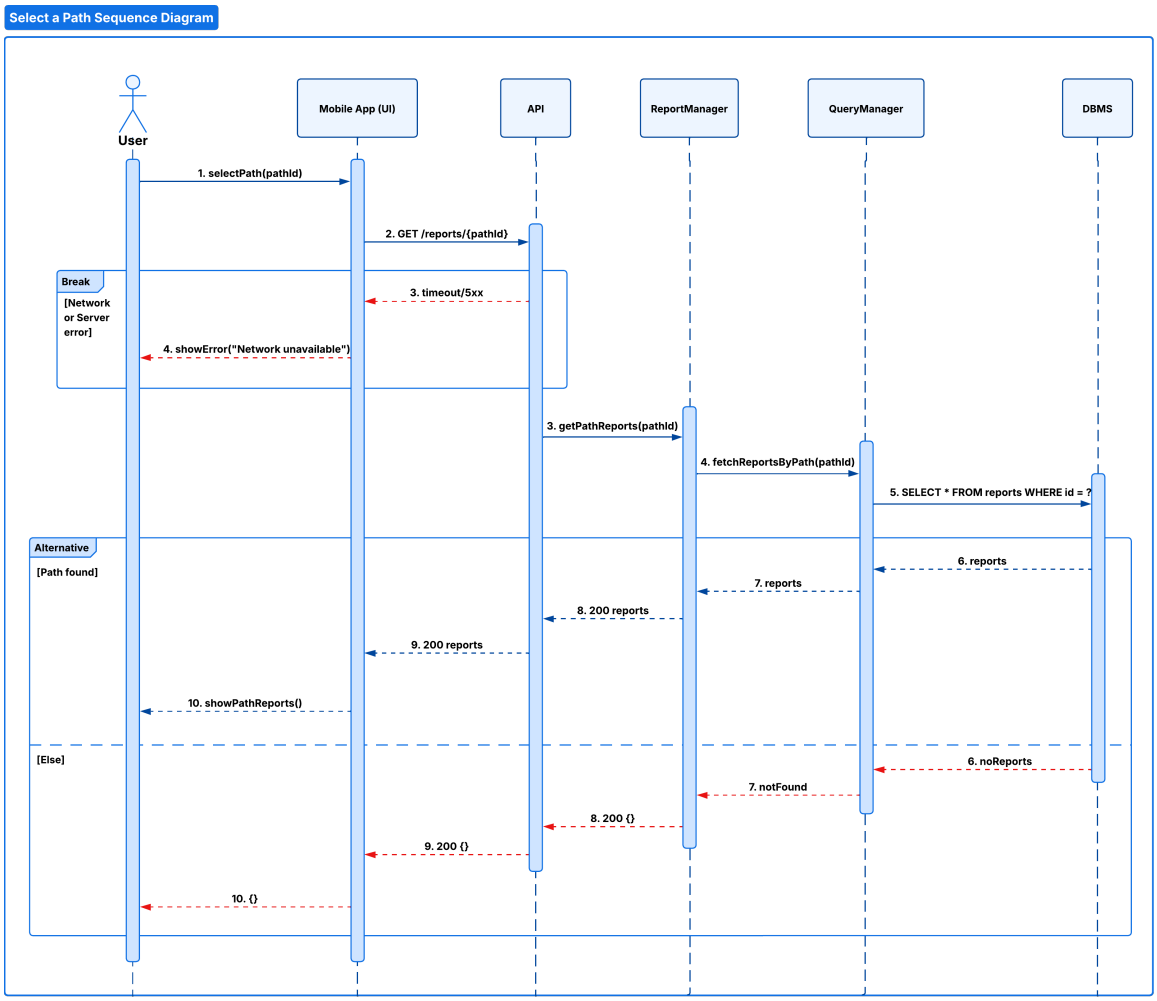

Figure 2.8: Select a Path Sequence Diagram

[UC7] - Create a Path in Manual Mode

When a logged-in user decides to create a new path, the interaction starts by opening the path creation page in the mobile app. The user is first asked to provide the required metadata and to select the **manual creation mode**. Once the metadata is submitted, the mobile app performs an initial **local validation**.
If the metadata is invalid or incomplete, the app immediately notifies the user. Otherwise, the app displays an interactive map that allows the user to manually draw the path by defining its segments. As segments are drawn, the app can call the backend snapping endpoint (**POST /paths/snap**) to align the coordinates to the road network via the **SnappingService**. Any error during the snapping process is non-blocking, so the flow continues and the user is not interrupted if snapping fails at any moment. After the drawing phase, the user submits the generated segments. Then, the mobile app sends a **POST** request to the backend through the **API Entrypoint**. The entrypoint forwards the request to the **PathManager**, which handles the creation logic.
The PathManager delegates the persistence of the new path to the **QueryManager**, which executes the corresponding **INSERT** operation on the **DBMS**, associating the path with the authenticated user. Upon successful insertion, the PathManager returns the generated path identifier to the API Entrypoint, which responds with a **201 Created** status. The mobile app receives the response and displays a confirmation message indicating that the path has been successfully created. In case of network or server errors, the app notifies the user accordingly.

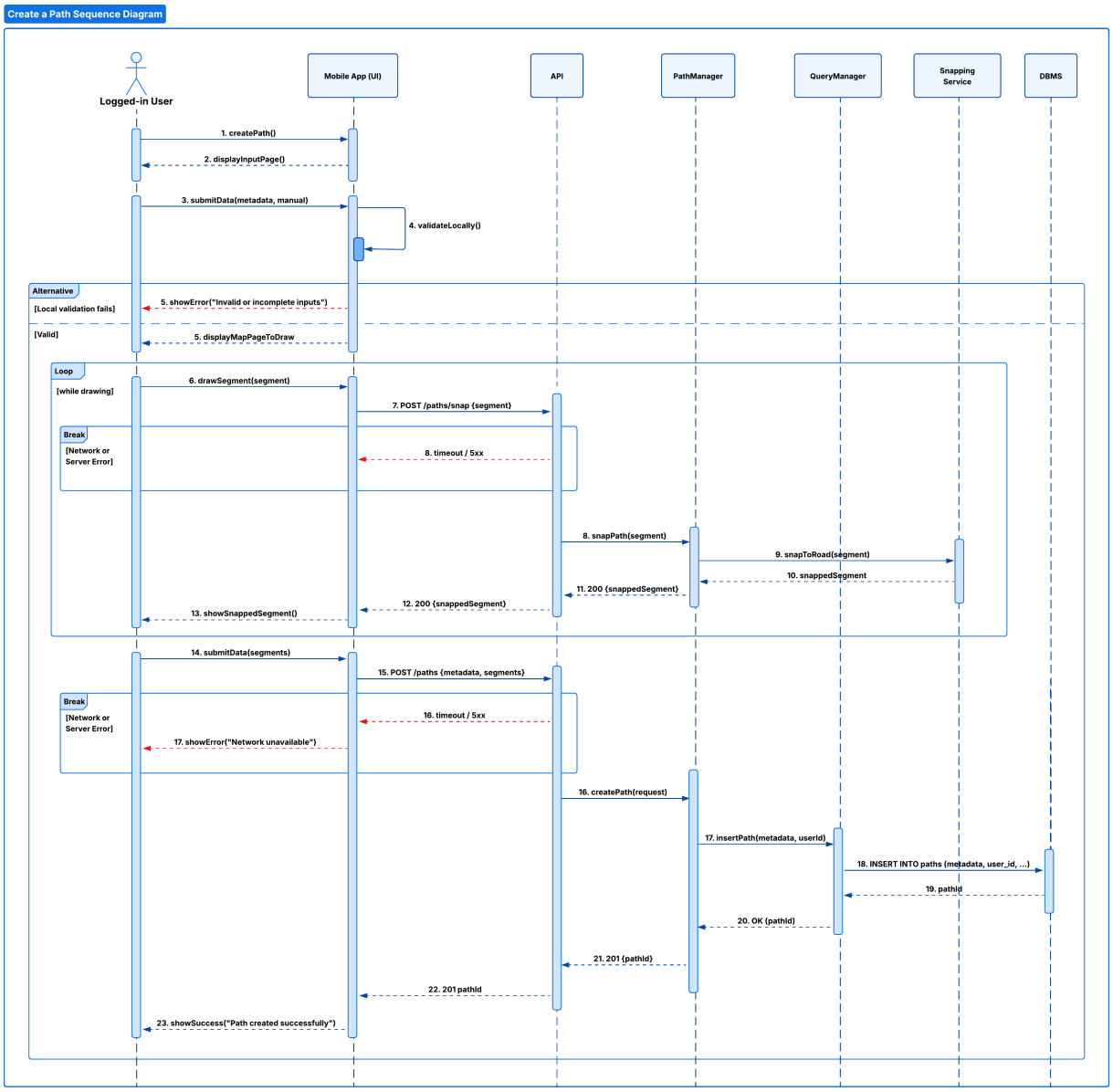

Figure 2.9: Create a Path in Manual Mode Sequence Diagram

[UC8] - Create a Path in Automatic Mode

When a logged-in user chooses to create a new path in **automatic mode**, the mobile app first displays a form to collect the required metadata and performs a preliminary **local validation**. If the validation fails, the user is immediately notified.
If the input is valid, the **GPS module** continuously provides location samples (latitude, longitude, speed, timestamp) while the user is moving. The mobile app locally stores these samples throughout the tracking session.
If a GPS signal error occurs during movement, the app detects the issue, interrupts the process, and informs the user. When the user completes the recording session, the collected samples undergo a final **local validation**. If the data is invalid, the user is notified.
If both metadata and recorded samples are valid, the mobile app sends a **POST** request to the backend through the **API Entrypoint**. The entripoint forwards the request to the **PathManager**, which stores the new path by delegating persistence to the **QueryManager**. After the path is successfully inserted into the **DBMS**, the PathManager returns the generated path identifier to the API Entrypoint, which responds with a **201 Created** status. The mobile app then displays a success message confirming that the automatically generated path has been saved.
Network or server failures are handled consistently with other interactions.

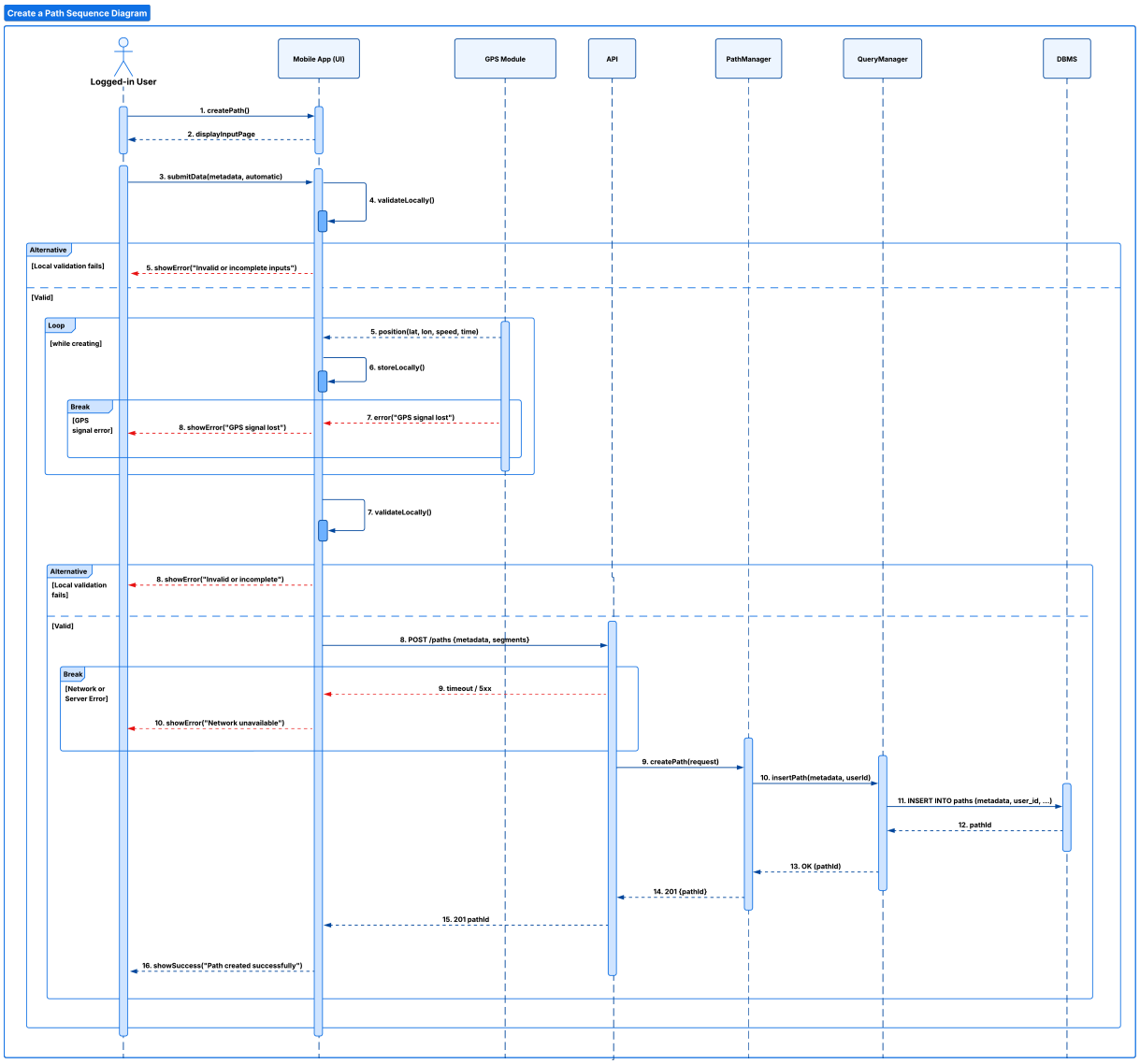

Figure 2.10: Create a Path in Automatic Mode Sequence Diagram

[UC9] - View Path Library

A logged-in user wants to view the list of paths previously created. The interaction starts when the user opens the personal path library section in the mobile application. The app sends a request to the backend through the **API Entrypoint** to retrieve all paths associated with the authenticated user. The **API Entrypoint** forwards the request to the **PathManager**, which retrieves the list of paths owned by the user by querying the database through the **QueryManager**. The QueryManager executes the corresponding **SELECT** operation on the **DBMS**.

If one or more paths are found, the list is returned to the mobile app and displayed to the user. If no paths exist for the given user, the backend returns a **200 OK** response with an empty list, and the mobile app notifies the user that no paths are available.

In case of network or server errors, the request is interrupted and the mobile app displays an appropriate error message.

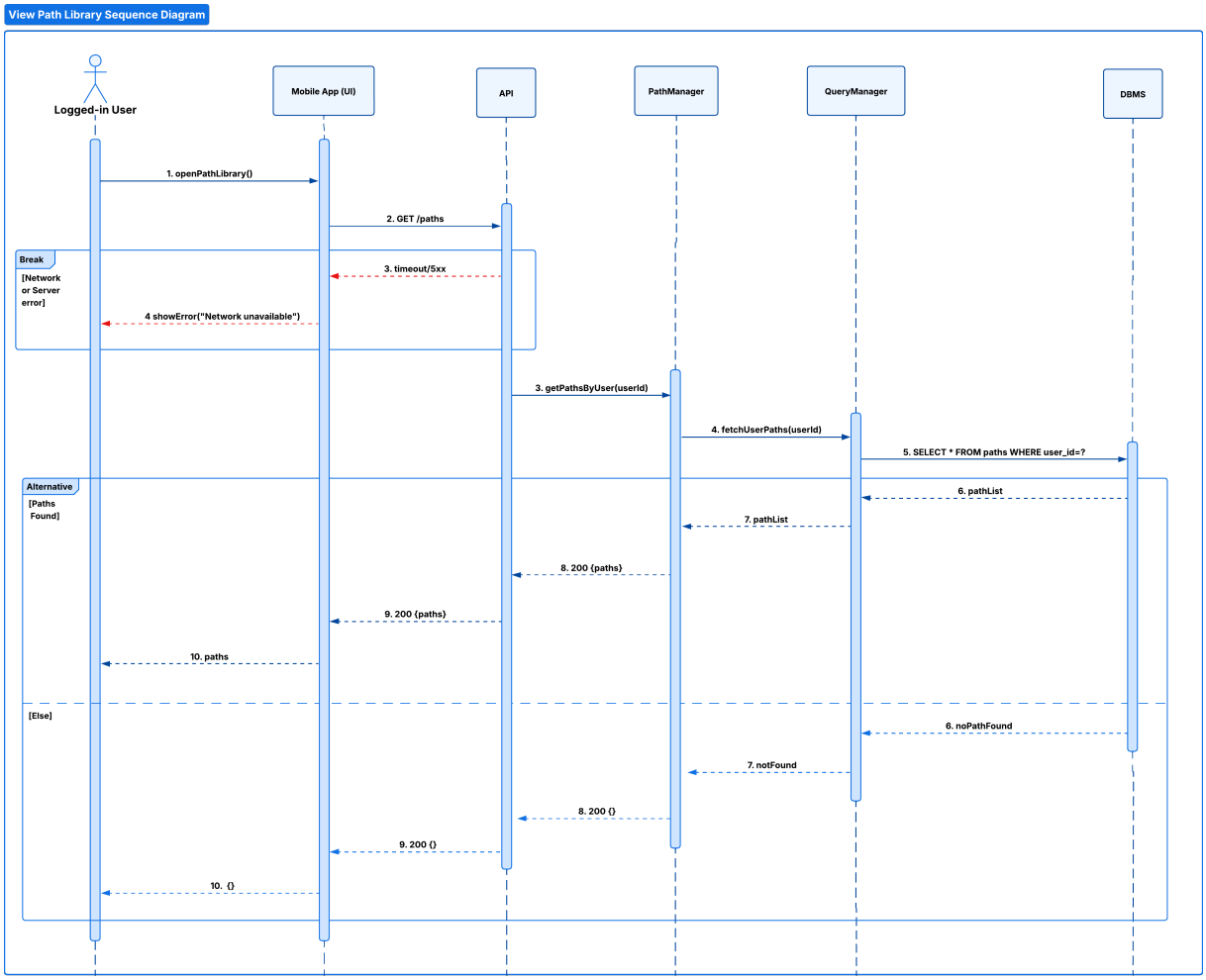

Figure 2.11: View Path Library Sequence Diagram

[UC10] - Manage Path Visibility

A logged-in user wants to change the visibility of one of his previously created paths. The interaction begins when the user selects a path and requests a visibility update from the mobile application. The app sends a **PATCH** request to the backend through the **API Entrypoint**, specifying the desired visibility value.
The **API Entrypoint** forwards the request to the **PathManager**, which first verifies that the requesting user is the owner of the selected path. This ownership check is performed by querying the database through the **QueryManager**. If the path does not exist, the backend returns a **404 NOT_FOUND** response. If the user is not the owner, a **403 FORBIDDEN** response is generated.
If the ownership verification succeeds, the **PathManager** updates the visibility attribute of the path by issuing an **UPDATE** operation through the **QueryManager**. Upon successful completion, the backend returns a **200 OK** response, and the mobile app confirms the update to the user. Network or server-side errors result in the interruption of the flow and the display of an appropriate error message on the client.

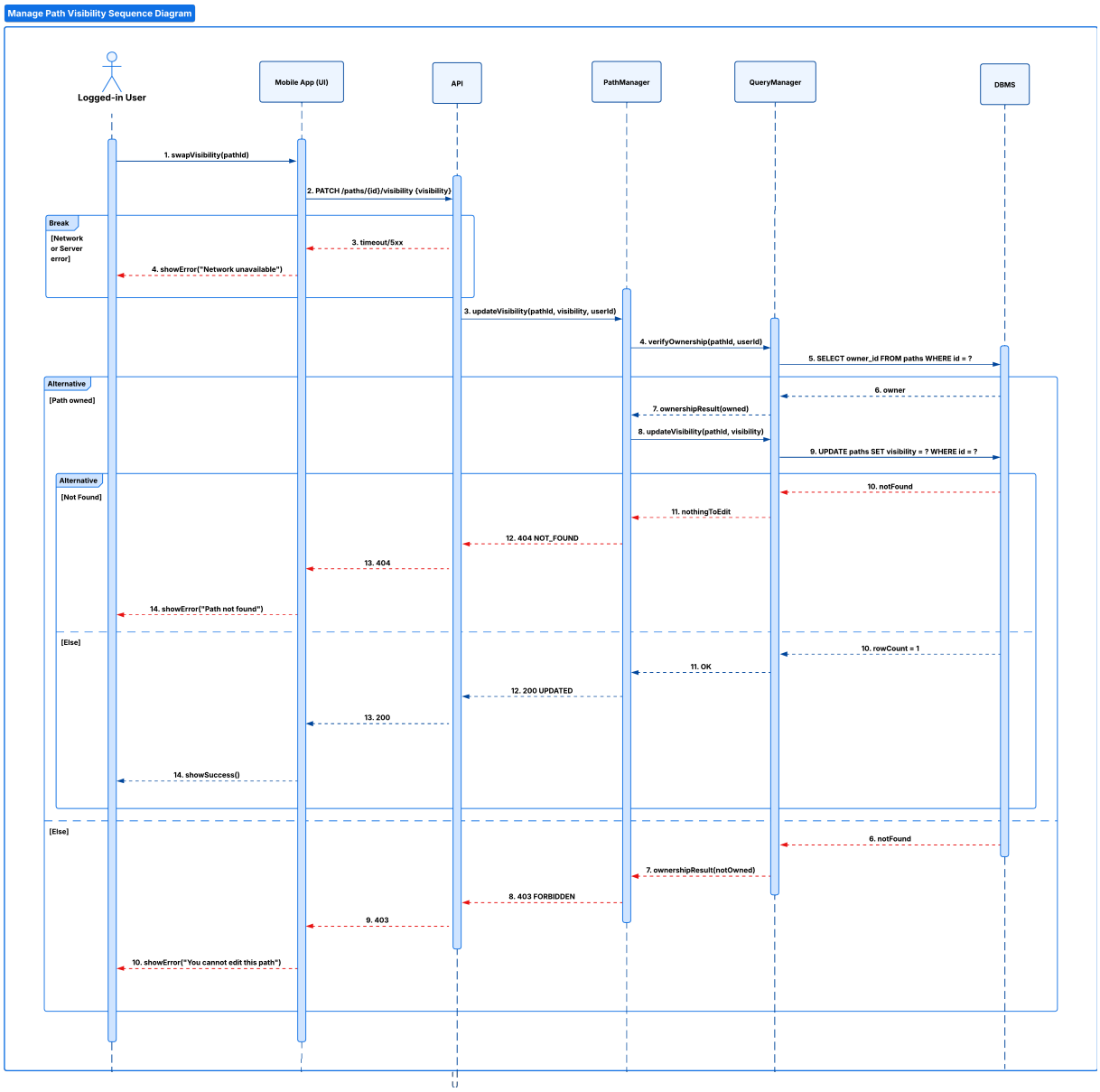

Figure 2.12: Manage Path Visibility Sequence Diagram

[UC11] - Delete a Path

When a logged-in user wants to permanently delete one of his paths, he first selects a path to delete from the mobile application. The app sends a **DELETE** request to the backend through the **API Entrypoint**.
The **API Entrypoint** forwards the request to the **PathManager**, which first verifies that the path exists and that the requesting user is its owner. This verification is performed by querying the database through the **QueryManager**. If the path does not exist, a **404 NOT_FOUND** response is generated. If the user is not authorised to delete the path, the backend returns a **403 FORBIDDEN** response.
If the ownership check succeeds, the **PathManager** performs the deletion by issuing a cascade delete operation through the **QueryManager**, which removes the path from the **DBMS**. Upon successful deletion, the backend returns a **204 NO_CONTENT** response. The mobile app then confirms the successful removal of the path to the user. If a network or server error occurs during the process, the app displays a corresponding error message.

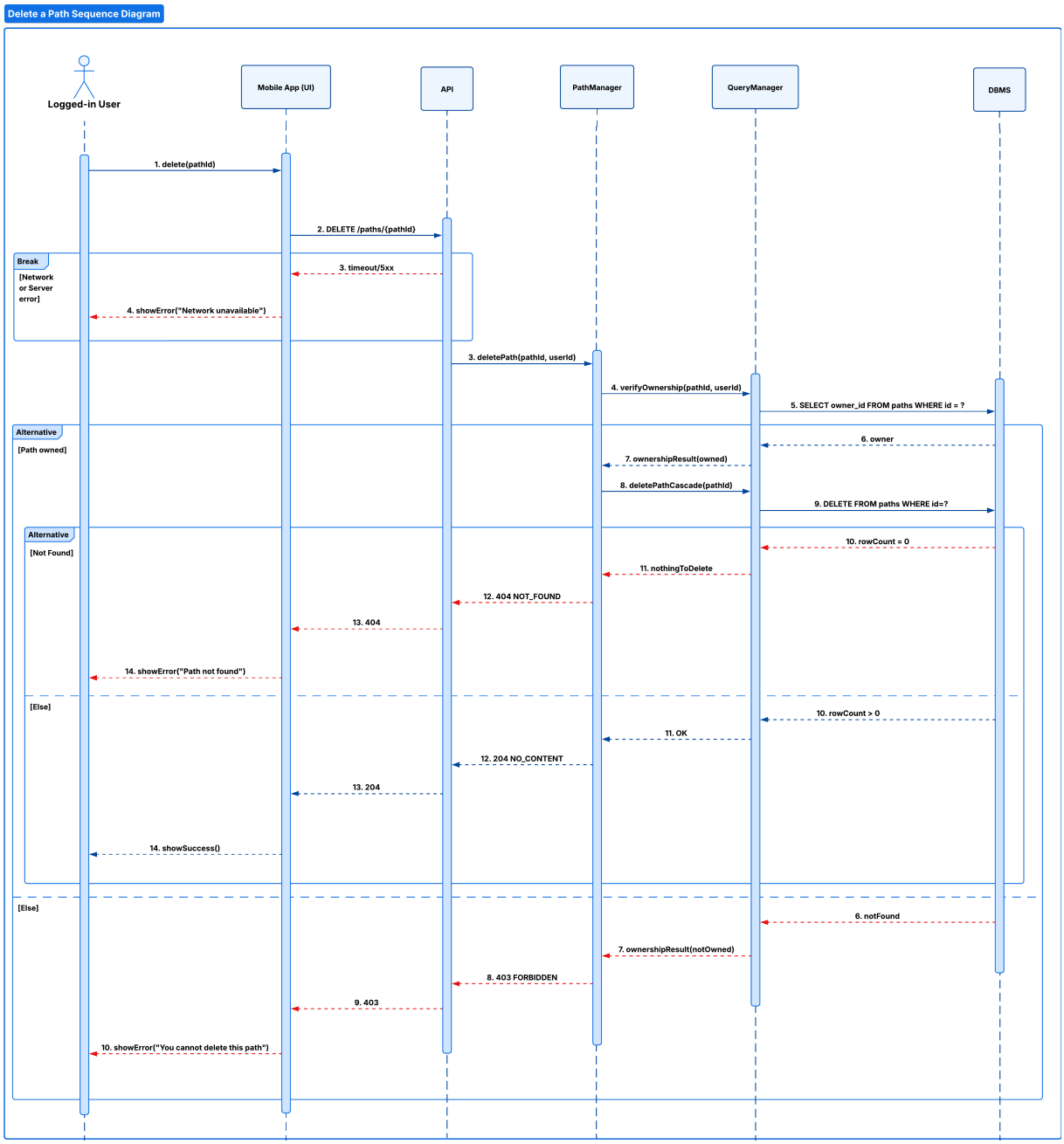

Figure 2.13: Delete a Path Sequence Diagram

[UC12] - Start a Trip as Guest User

When a guest user starts a trip using the BBP mobile app, he first selects a path from the available options. The mobile app displays the selected path on the map and continuously receives position updates from the **GPS module** while the user is moving and updates the map accordingly.
If at any point the GPS signal is lost, the GPS module reports the error and the mobile app displays a corresponding warning to the user, interrupting the normal tracking flow. No interaction with the backend occurs in this use case. Guest trips are not associated with a persistent trip identifier and are not stored by the system.

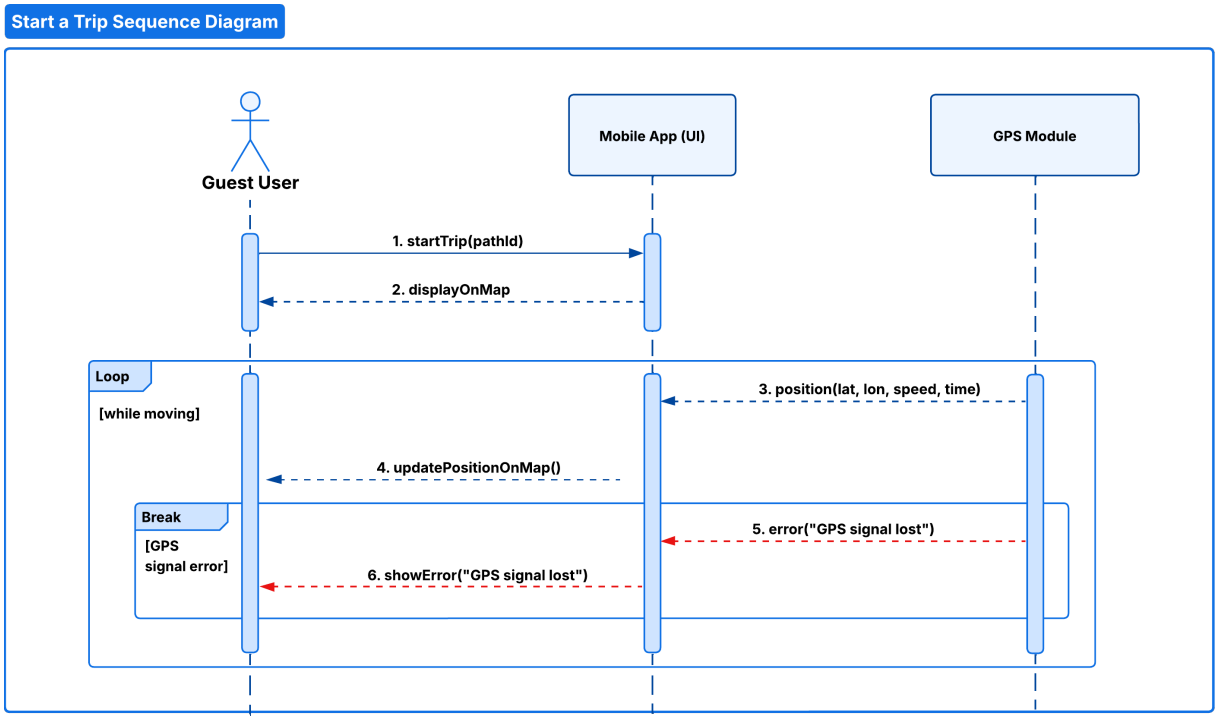

Figure 2.14: Start a Trip as Guest User Sequence Diagram

[UC13] - 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 from the mobile app. The selected path is displayed on the map, and the user is prompted to enable **automatic mode**. The user explicitly declines this option, choosing to proceed in manual mode. GPS tracking starts and the device periodically provides position updates.
During the trip, the mobile app stores the collected samples locally and updates the map in real time. A trip identifier is generated and maintained locally by the app during tracking and is not sent to the backend at this stage. If the GPS signal is lost at any point, the GPS module reports the error and the mobile app displays a corresponding warning to the user, interrupting the tracking process.

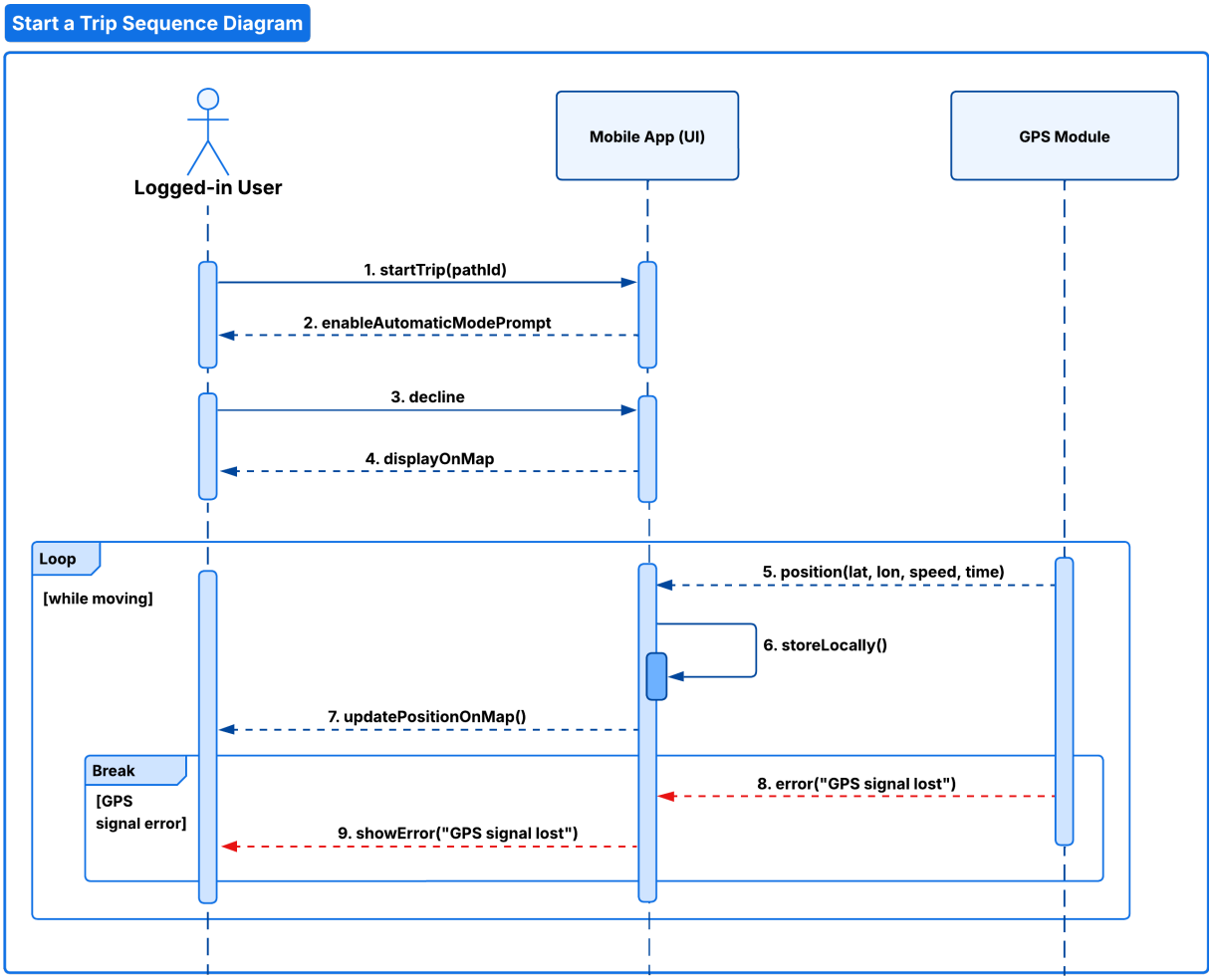

Figure 2.15: Start a Trip in Manual Mode as a Logged-in User Sequence Diagram

[UC14] - Start a Trip in Automatic Mode as a Logged-in user

When a logged-in user selects a path and chooses to enable **automatic mode**, the mobile app displays the selected path and attempts to establish a connection with the required **external sensors**. If the connection to the external sensors fails, the mobile app immediately notifies the user with an error message and the trip does not start.
If the connection succeeds, the GPS module periodically sends position updates, which the mobile app stores locally and reflects on the map in real time. The trip identifier is generated and maintained locally during tracking and is sent to the backend only when the trip is stopped and finalised. During the trip, if the GPS signal is lost, the tracking loop is interrupted and the user is notified. If instead the connection to the external sensors drops while the trip is ongoing, the app stops the automatic mode and displays a corresponding error message.

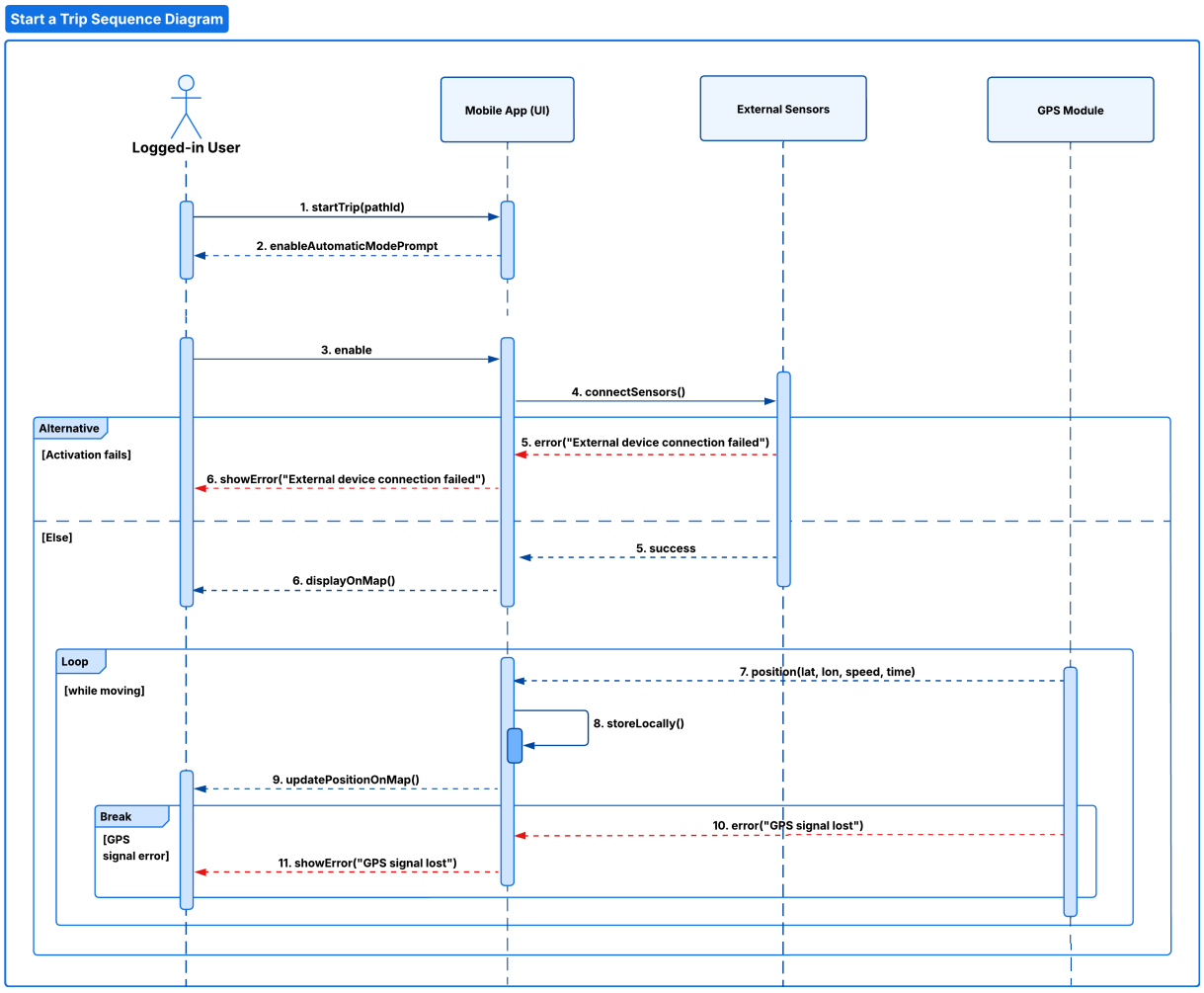

Figure 2.16: Start a Trip in Automatic Mode as a Logged-in User Sequence Diagram

[UC15] - Stop a Trip as Guest User

This use case describes how a guest user stops an ongoing trip. Since guest trips are not stored on the backend, the entire interaction is handled locally by the mobile application. The process starts when the user selects the **Stop Trip** action. The mobile app ends the trip visualisation and updates the user interface accordingly. No network communication is performed and no data is persisted, as guest trips are not associated with a backend record.

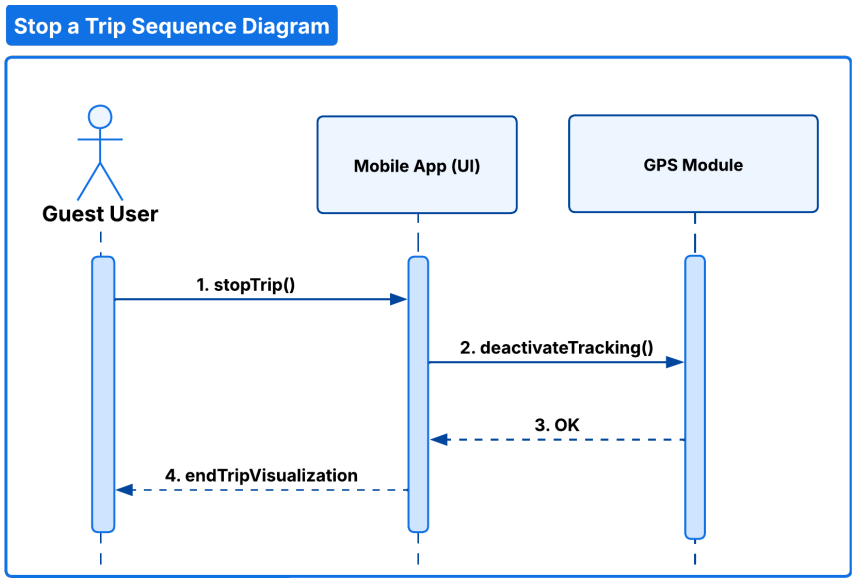

Figure 2.17: Stop a Trip as a Guest User Sequence Diagram

[UC16] - Stop a Trip as a Logged-in User

A logged-in user wants to stop an ongoing trip. The interaction begins when the user selects the **Stop Trip** action from the mobile app. The app first terminates any active data acquisition and, if the trip was started in automatic mode, external sensor connections. The collected data is then validated. If the validation succeeds, the mobile app sends a finalisation request to the backend through the **API Entrypoint**. The **TripManager** processes the request.

When the trip is valid and the external service is reachable, the **TripManager** retrieves a weather snapshot through the **WeatherService** based on the trip samples. It also invokes the **StatsManager** to compute and aggregate trip statistics, which are stored together with the trip data. The complete trip summary is persisted through the **QueryManager**. Upon successful completion, the backend responds with a **201 Created** status. The mobile app displays a success message to the user. Network failures or server-side errors trigger the corresponding alternative flows.

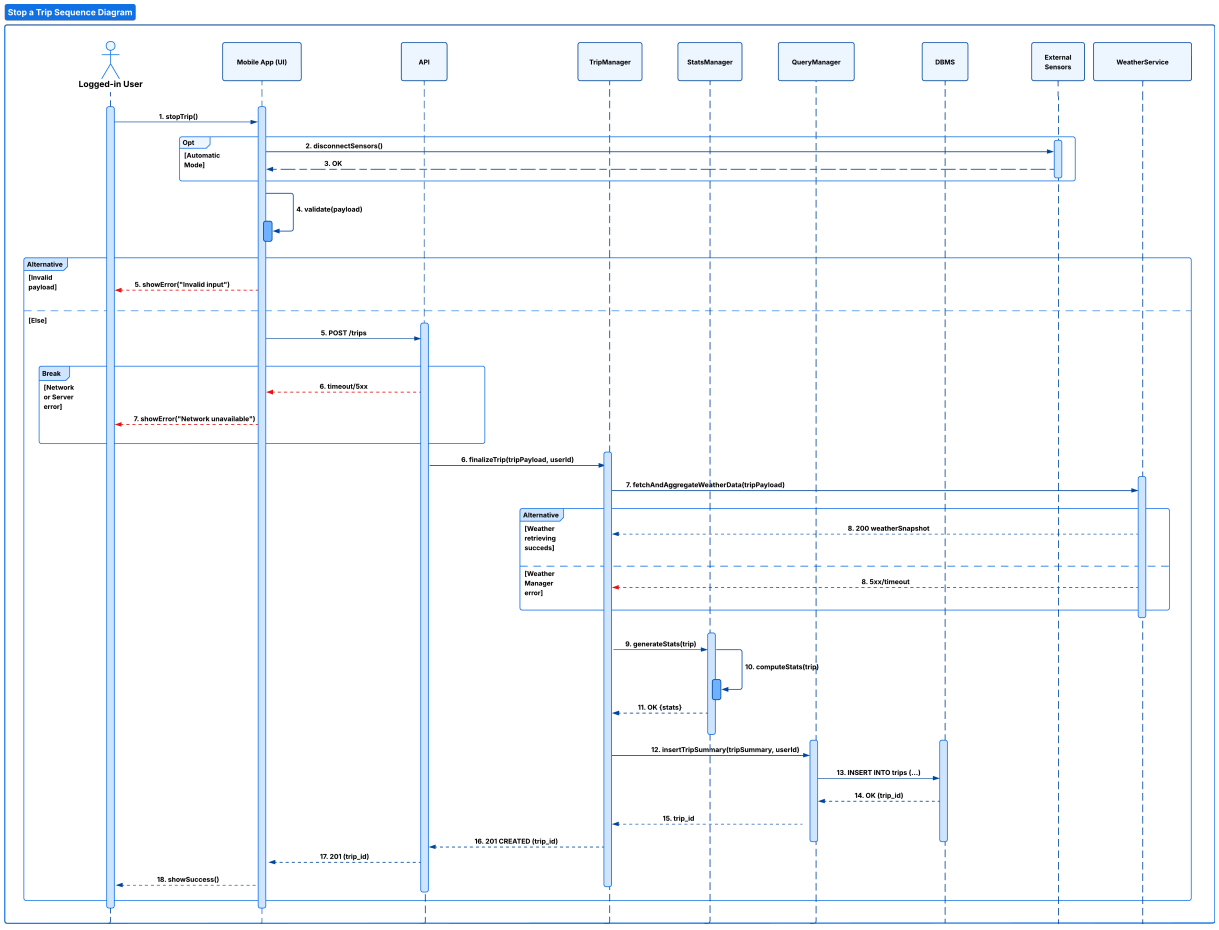

Figure 2.18: Stop a Trip as a Logged-in User Sequence Diagram

[UC17] - Make a Report in Manual Mode

When a logged-in user wants to manually report an issue encountered on a path, the user opens the report creation form on the mobile application. The app retrieves the current position from the **GPS module**. If the position cannot be retrieved, the app immediately displays an error message and the reporting process is interrupted.
The user fills in the report form by providing report details, then submits the report. Before contacting the backend, the mobile app performs a **local validation** to ensure that all mandatory fields are correctly filled. If validation fails, the app shows an error message and no request is sent. When the input is valid, the mobile app sends a **POST request** containing the report payload to the backend through the **API Entrypoint**. In case of network timeouts or server-side failures, the app notifies the user accordingly.
The **API Entrypoint** forwards the request to the **ReportManager**, which creates the report by storing the user identifier, the associated path, the position, and the report data through the **QueryManager**. The **DBMS** generates a new report identifier and confirms the insertion. Upon success, the backend returns a **201 Created** response with the report identifier, and the mobile app displays a confirmation message to the user.

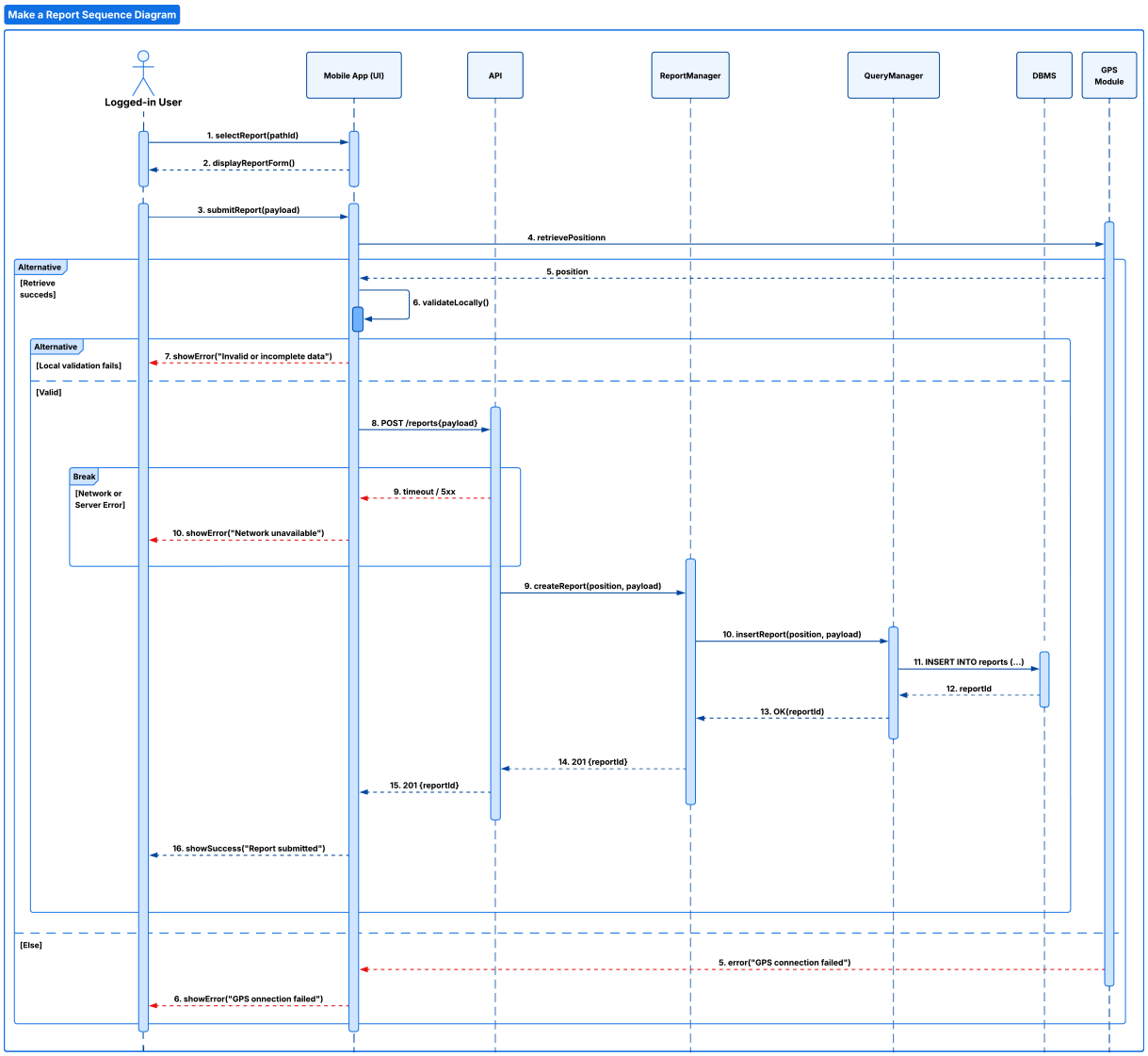

Figure 2.19: Make a Report in Manual Mode Sequence Diagram

[UC18] - Make a Report in Automatic Mode

During an ongoing trip with automatic mode enabled, external sensors may automatically detect an obstacle or issue on the path. When such a detection occurs, the mobile app retrieves the current position from the **GPS module**. If the GPS position cannot be obtained, the app displays an error message and the reporting flow terminates.
If the position retrieval succeeds, the app displays a pre-filled report form containing the detected issue information. The user can review, modify, or complete the report details before submission. After confirmation, the mobile app performs a **local validation** of the generated report data. If the payload is invalid or incomplete, an error is shown and the backend is not contacted.
When validation succeeds, the app sends the report payload to the backend via the **API Entrypoint**. Network or server errors result in a timeout and an appropriate error message is displayed to the user. The **ReportManager** receives the request and creates a new report by delegating its persistence to the **QueryManager**, which inserts the record into the **DBMS**. After successful insertion, the backend responds with a **201 Created** status. Finally, the mobile app informs the user that the report has been submitted successfully.

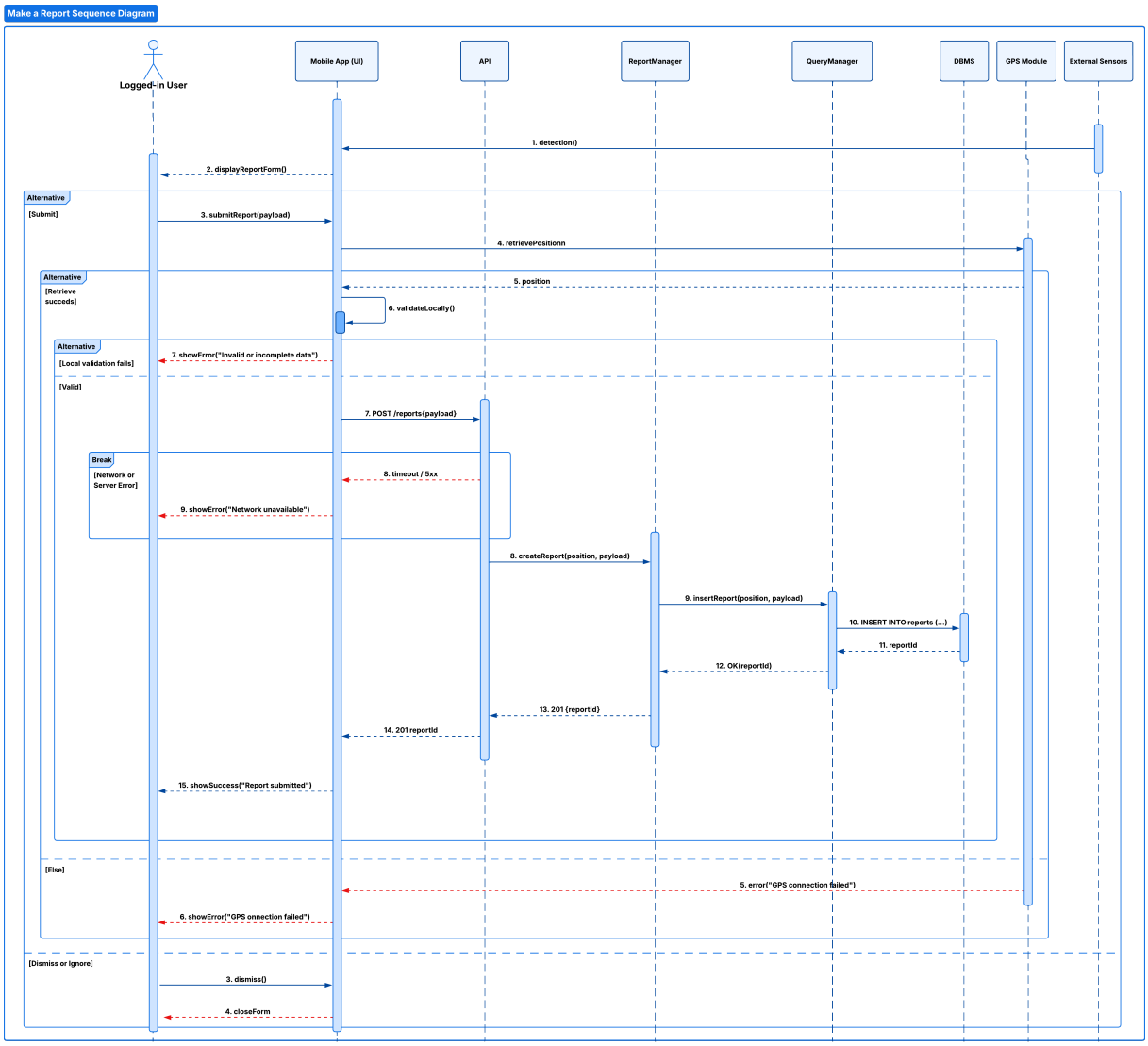

Figure 2.20: Make a Report in Automatic Mode Sequence Diagram

[UC19] - Confirm a Report

While a logged-in user is travelling, the system may notify him of an existing report located nearby. The mobile app displays a confirmation form, allowing the user to either confirm, reject, or dismiss the reported issue. If the user ignores the notification, the form is dismissed after a short period of time and no further action is taken. If the user submits a decision, the mobile app performs a **local validation** to ensure that the input is valid. When validation succeeds, the app sends the confirmation (or rejection) request to the backend through the **API Entrypoint**. In case of network or server errors, an appropriate error message is shown to the user. The **API Entrypoint** forwards the request to the **ReportManager**, which creates a report entry. The confirmation is persisted through the **QueryManager** by inserting the record into the **DBMS**.
If the operation completes successfully, the backend responds with a **201 Created** status and returns the confirmation identifier. The mobile app then notifies the user that the confirmation has been submitted successfully.

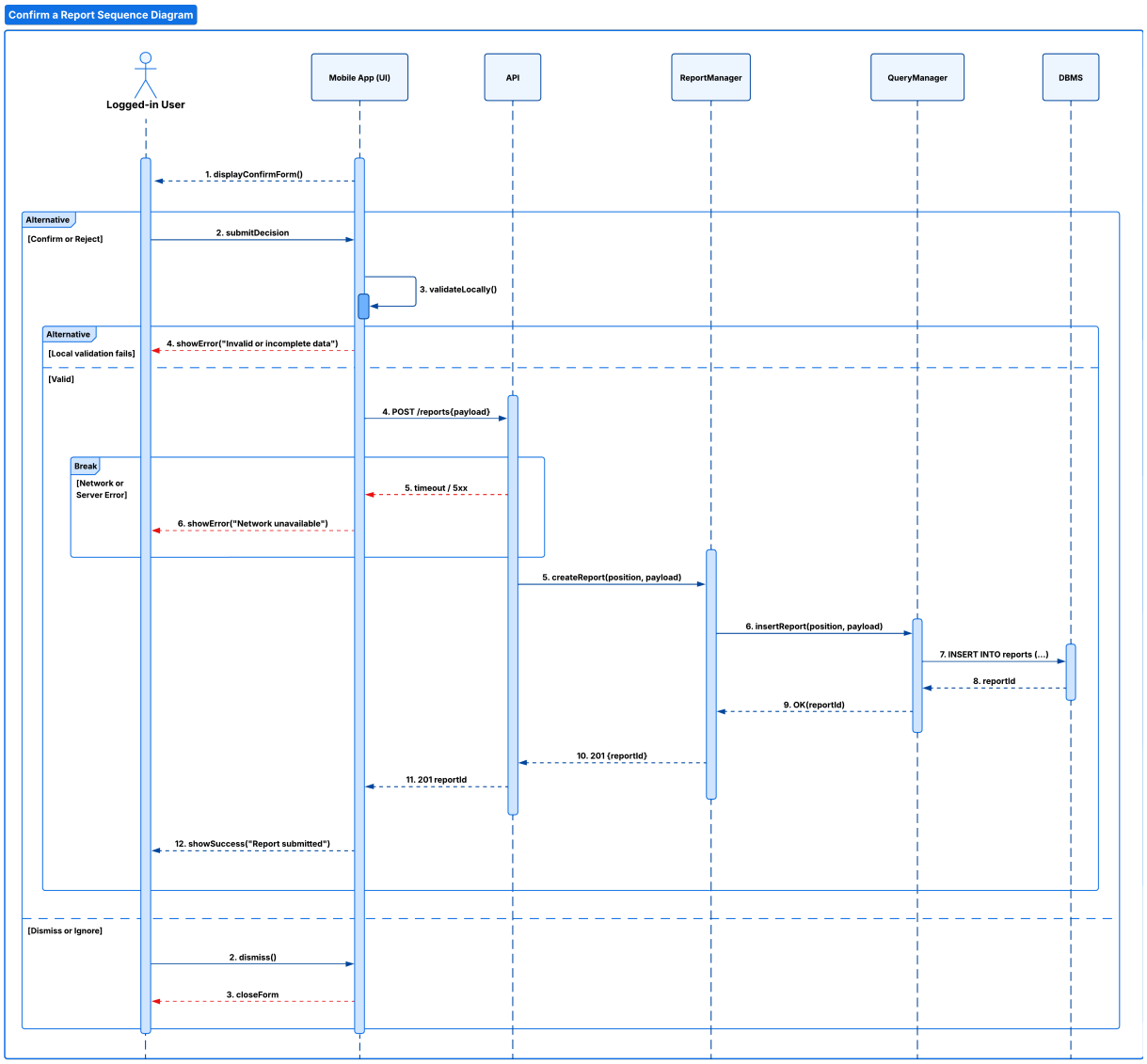

Figure 2.21: Confirm a Report Sequence Diagram

[UC20] - View Trip History

When a logged-in user opens the trip history section from the mobile application, the mobile app sends a request to the backend through the **API Entrypoint** to retrieve the list of trips associated with the user. The **API** forwards the request to the **TripManager**, which retrieves the user identifier and queries the database through the **QueryManager**. The **DBMS** returns the list of trips linked to the user account. For each trip in the list, the **TripManager** may enrich missing data. If weather data is missing, it invokes the **WeatherService** to compute a snapshot and stores it back through the **QueryManager**. It also requests the relevant reports via the **ReportManager** so they can be shown on the map. If statistics are missing, it delegates to the **StatsManager**, which computes and aggregates the statistics and then persists them via the **QueryManager** before responding.

If one or more trips are found, the backend responds with a **200 OK** status containing the trip list, and the mobile app displays the history to the user. If no trips are found, the backend responds with a **200 OK** status containing an empty list, and the mobile app shows an empty state message.

In case of network or server failures, the request times out and the mobile app notifies the user that the service is unavailable.

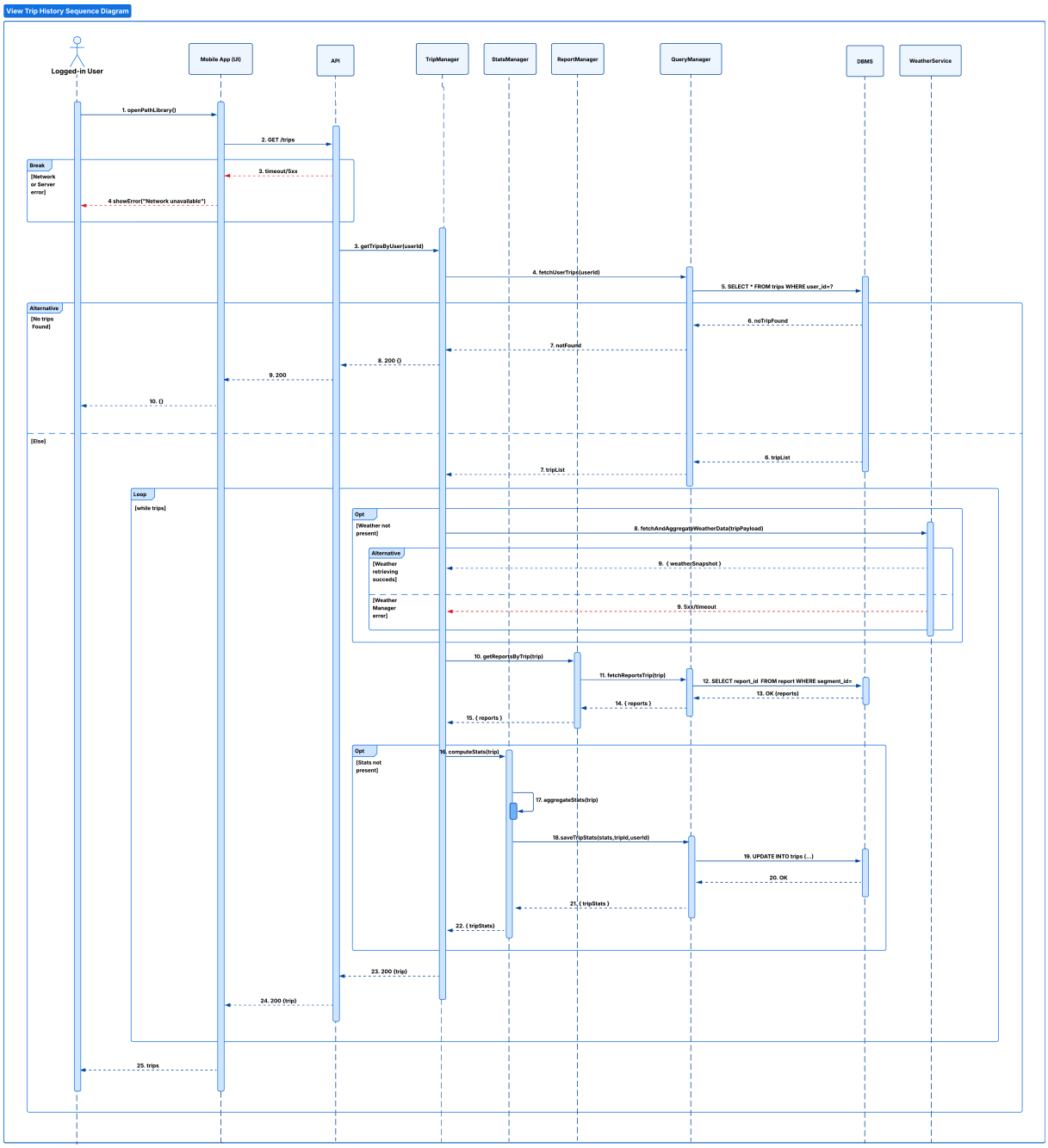

Figure 2.22: View Trip History Sequence Diagram

[UC21] - Delete a Trip

A logged-in user wants to permanently delete one of his recorded trips. He selects a trip to delete from the mobile application. The app sends a **DELETE** request to the backend through the **API Entrypoint**.
The **API Entrypoint** forwards the request to the **TripManager**, which first verifies that the trip exists and that the requesting user is its owner. This verification is performed by querying the database through the **QueryManager**. If the trip does not exist, the backend answers with a **404 NOT_FOUND** response. If the user is not the owner of the trip, a **403 FORBIDDEN** response is generated.
If the ownership check succeeds, the **TripManager** performs the deletion by issuing a cascade delete operation through the **QueryManager**, which removes the trip from the **DBMS**. Upon successful deletion, the backend returns a **204 NO_CONTENT** response. The mobile app then confirms the successful removal of the trip to the user. If a network or server error occurs during the process, the app displays a corresponding error message.

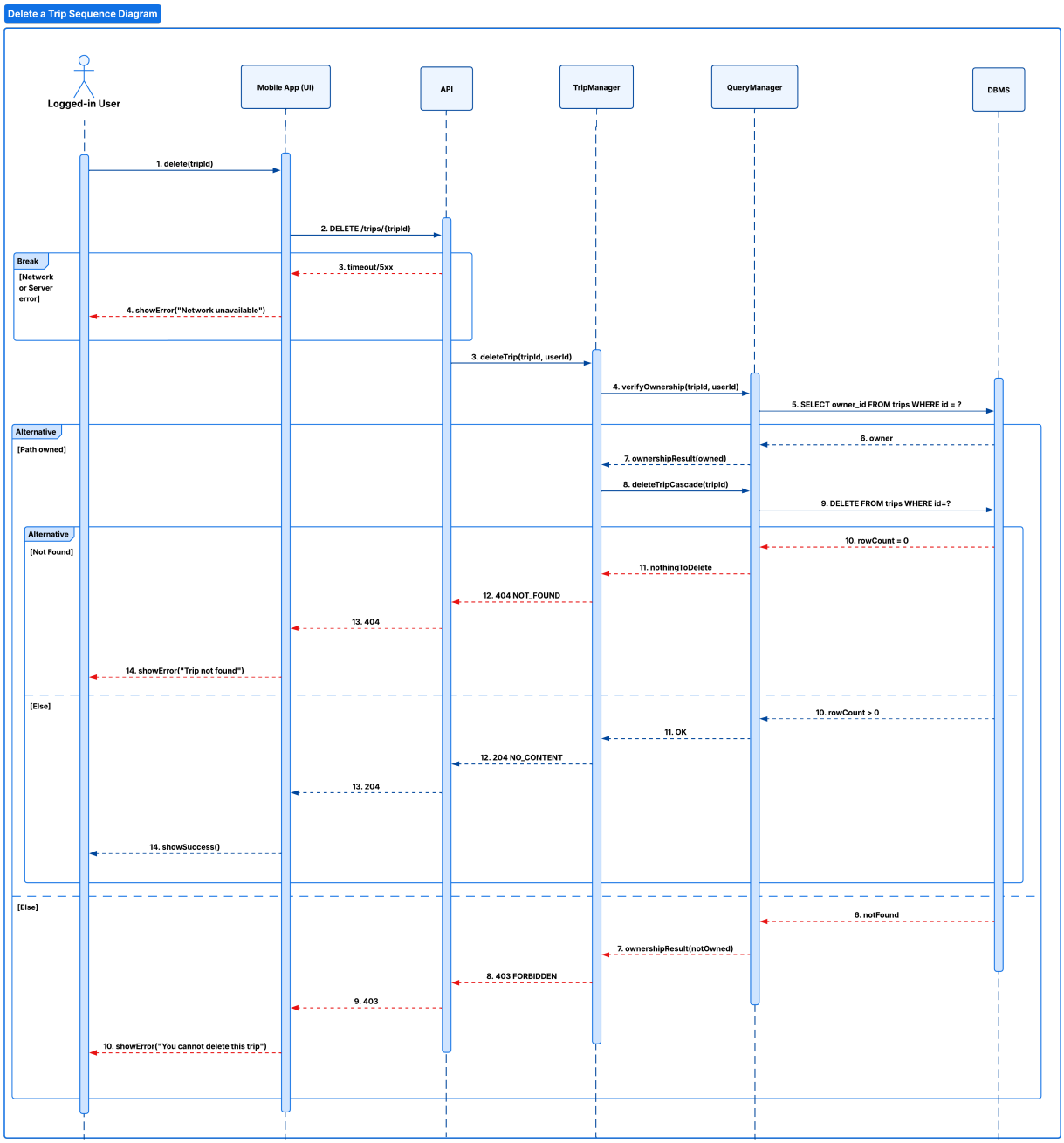

Figure 2.23: Delete a Trip Sequence Diagram

[UC22] - View Overall Statistics

A logged-in user opens the overall statistics section from the mobile app. The application sends a request to the backend through the **API Entrypoint** to retrieve aggregated statistics across all user trips.
The **API** delegates the request to the **StatsManager**, which first checks whether previously computed overall statistics are still valid by comparing the stored trip count with the current number of trips retrieved through the **QueryManager**.
If valid statistics already exist, they are loaded from the database and returned to the mobile app with a **200 OK** response. If the stored data is missing or outdated, the **StatsManager** retrieves all trip records from the **DBMS**, computes the aggregated metrics (e.g., total distance, total duration, average speed), stores the updated statistics, and returns the results to the client.
If no trips exist for the user, the backend responds with a **404 NOT_FOUND** status, which is propagated to the mobile app. Any network or server-side failure results in a timeout and an error message displayed on the client.

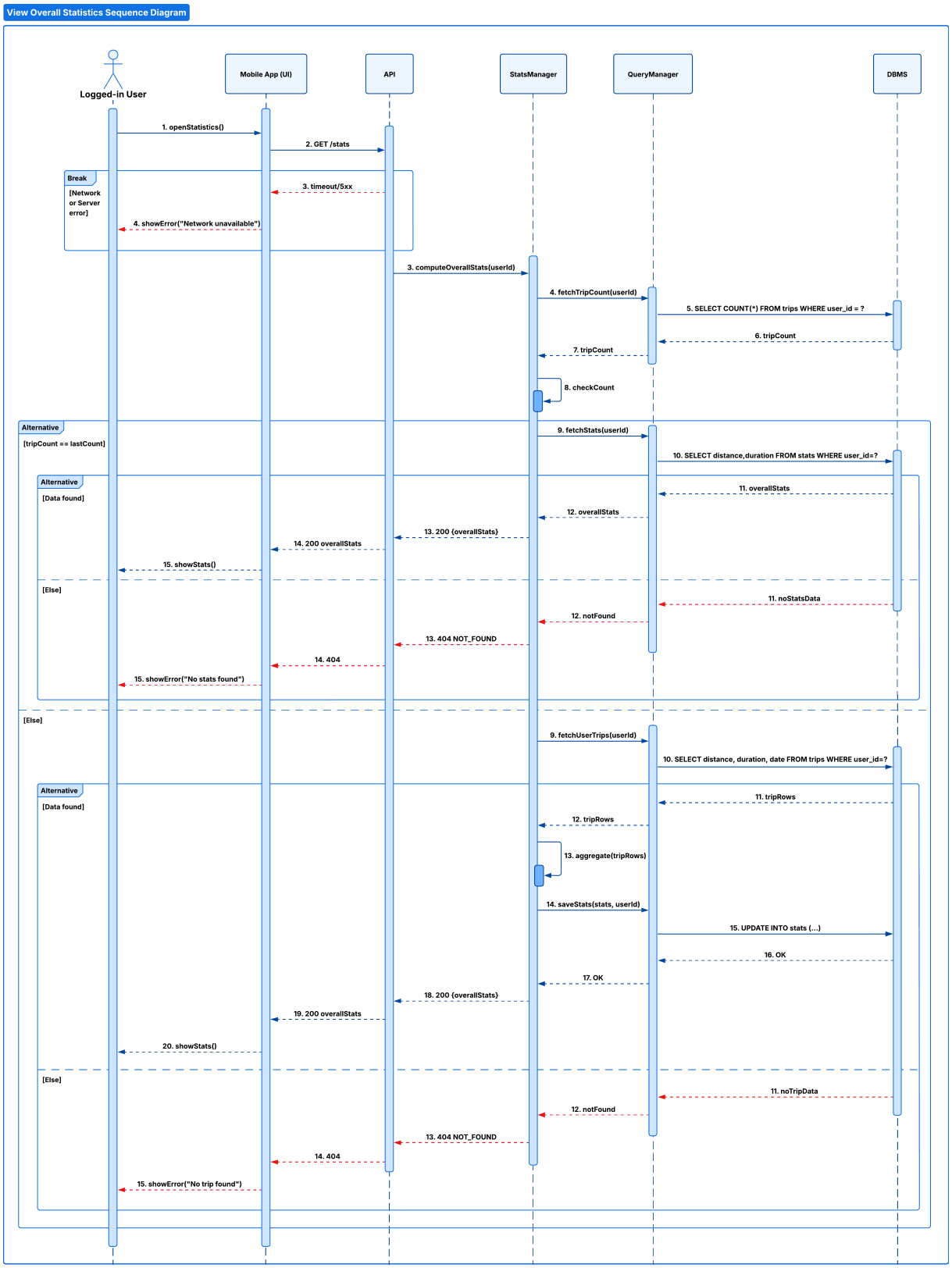

Figure 2.24: View Overall Statistics Sequence Diagram

2.5. Component Interfaces

This section describes the operations exposed by each backend component and the interfaces through which these components collaborate at runtime.
The **API Entrypoint** exposes the REST endpoints used by the mobile app and routes each request to the correct Manager or Service. Unless explicitly noted, most Manager methods operate in an authenticated context: the identifier of the currently logged-in user is implicitly supplied by the **API Entrypoint** after validating the access token, and is therefore not included in the method signatures. Public endpoints (e.g., registration, login, and path search) are invoked without an authenticated context. The **QueryManager**, in contrast, interacts directly with the relational DBMS and does not rely on authentication parameters.

User Module - AuthManager

The **AuthManager** handles user authentication, credential verification, token generation, and logout flows. It is invoked through the **API Entrypoint** and relies on the **QueryManager** to retrieve and persist authentication metadata, including refresh tokens stored in the DBMS.
It exposes the following operations:

- **loginUser(email, password)**: verifies the supplied credentials and, if valid, issues fresh access and refresh tokens and persists the new refresh token.
- **refreshToken(refreshToken)**: validates and rotates the refresh token, issuing a new access/refresh token pair.
- **logout(refreshToken)**: invalidates the provided refresh token and terminates the associated session in the DBMS.

User Module - UserManager

The **UserManager** manages user-profile data and receives requests via the **API Entrypoint**. It uses the **QueryManager** for persistence and exposes:

- **registerUser(profilePayload)**: validates registration data, checks for duplicate email/username, stores the user profile, and returns the created user.
- **getProfile(userId)**: retrieves the stored profile for the authenticated user.

- **updateProfile(userId, profilePayload)**: validates editable fields, prevents duplicate emails/usernames, applies the update, and returns a confirmation message.

PathManager

The **PathManager** implements all functionality related to bike paths: manual path creation, route search, ranking of candidate routes, path selection, deletion, and visibility updates. It is accessed through the **API Entrypoint** and uses the **QueryManager** to retrieve and update persistent path data. When manual path creation requires snapping, it invokes the **SnappingService**. The **API Entrypoint** also exposes a dedicated snapping endpoint that forwards coordinates to this component.
The public operations are:

- **searchPath(origin, destination, userId?)**: geocodes origin/destination, filters existing paths by visibility and proximity, and returns matching routes.
- **createPath(pathPayload, userId)**: validates metadata and path segments (manual or automatic mode), persists the new path, and associates it with the creator.
- **snapPath(coordinates)**: forwards coordinates to the **SnappingService** and returns the adjusted polyline.
- **getPathsByUser(userId)**: retrieves all paths created by the authenticated user.
- **deletePath(pathId, userId)**: verifies ownership and removes the specified path, triggering cascade deletion of dependent records.
- **updateVisibility(pathId, visibility, userId)**: enforces ownership and updates the stored visibility field.

TripManager

The **TripManager** coordinates the lifecycle of a cycling session. It receives trip metadata (origin/destination, timestamps, trip segments) from the mobile app and persists it through the **QueryManager**. Weather enrichment is performed via the **WeatherService** on demand.
The component exposes the following methods:

- **createTrip(tripPayload, userId)**: stores a completed trip and triggers weather/statistics enrichment when required.

- **getTripsByUser(userId)**: returns the complete list of trips owned by the authenticated user, ordered by time, enriching missing weather, statistics, and reports when needed.
- **deleteTrip(tripId, userId)**: deletes the specified trip after verifying ownership.

ReportManager

The **ReportManager** handles obstacle reporting and confirmation flows. It is invoked by the **API Entrypoint** and also by the **TripManager** when trip summaries need to be enriched with associated reports. All persistence is delegated to the **QueryManager**.

- **getReportsByPath(pathId)**: retrieves reports associated with a selected path.
- **createReport(reportPayload, userId)**: stores a new report, either manual or automatic.
- **confirmReport(reportId, decisionPayload, userId)**: creates a confirmation or rejection entry for an existing report.

StatsManager

The **StatsManager** computes and retrieves aggregated metrics for trips and overall history. It is called by the **API Entrypoint** and by the **TripManager** at trip finalization, and also during trip retrieval when statistics are missing and need to be computed on demand.

- **getOverallStats(userId)**: returns cached overall statistics if valid, otherwise recomputes and stores them.
- **getTripStats(tripId)**: returns cached trip statistics or computes them on demand.
- **computeTripStats(tripPayload)**: derives metrics (distance, duration, avg speed) from trip samples.

WeatherService

The **WeatherService** interfaces with an external weather provider and is used by the **TripManager** to enrich trip summaries with environmental context at trip finalization, and again during trip retrieval if stored weather data is missing.

- **fetchAndAggregateWeatherData(coordinates)**: samples coordinates along the trip route, fetches point data, and returns an aggregated snapshot.

GeocodingService

The **GeocodingService** resolves user-provided location strings to coordinates used by the **PathManager**.

- **geocodeLocation(locationString)**: returns coordinates for a textual location.

SnappingService

The **SnappingService** aligns user-drawn segments to the road network and returns the adjusted polyline to the **PathManager**.

- **snapCoordinates(coordinates)**: snaps a list of coordinates to the road network.

QueryManager

The **QueryManager** is the uniform access point to the DBMS. It encapsulates all SQL operations required by the Managers, ensuring consistency and separation between domain logic and data storage.
It exposes:

- **createUser(email, password, username, systemPreferences)**: creates a new user record.
- **getUserByEmail(email)**: retrieves a user by email.
- **getUserById(userId)**: retrieves a user by identifier.
- **getUserByUsername(username)**: retrieves a user by username.
- **updateUserProfile(userId, profilePayload)**: stores changes to user profile fields.
- **createRefreshToken(userId, token, expiresAt)**: persists a refresh token.
- **getRefreshToken(token)**: retrieves a refresh token.
- **deleteRefreshToken(token)**: removes a refresh token.
- **getTripById(tripId)**: retrieves a trip and its ordered segments.
- **createTrip(userId, origin, destination, startedAt, finishedAt, statistics, tripSegments, title)**: stores a completed trip.
- **getTripsByUserId(userId)**: retrieves all trips for a user, ordered by date.

- **deleteTripById(tripId)**: deletes a trip.
- **updateTripWeather(tripId, weather)**: stores aggregated weather data for a trip.
- **createPath(userId, origin, destination, pathSegments, visibility, creationMode, title, description, distanceKm)**: stores a new path.
- **getPathById(pathId)**: retrieves a path and its ordered segments.
- **getPathsByUserId(userId)**: retrieves all paths for a user.
- **getPathByOriginDestination(userId, origin, destination)**: finds a user path by origin/destination.
- **searchPathsByOriginDestination(origin, destination, userId?)**: returns matching paths by proximity and visibility.
- **updatePathScore(pathId, score)**: updates the stored path score.
- **updatePathStatus(pathId, status)**: updates the stored path status.
- **deletePathById(pathId)**: deletes a path.
- **changePathVisibility(pathId, visibility)**: updates path visibility.
- **createReport(userId, pathId, position, payload)**: inserts a new obstacle report.
- **getReportsByPathId(pathId)**: retrieves reports for a path.
- **createReportConfirmation(userId, reportId, decision)**: inserts a confirmation or rejection.
- **getOverallStatsByUserId(userId)**: retrieves cached overall statistics.
- **upsertOverallStats(userId, statsPayload)**: creates or updates overall statistics.
- **getTripStatsByTripId(tripId)**: retrieves cached per-trip statistics.
- **upsertTripStats(tripId, statsPayload)**: creates or updates per-trip statistics.
- **createSegment(status, polylineCoordinates)**: stores a segment.
- **createSegmentWithId(segmentId, status, polylineCoordinates)**: stores a segment with a provided id.

- **getSegmentsByIds(segmentIds)**: retrieves segments by id.
- **getSegmentStatistics(segmentIds)**: retrieves segment status for scoring.

2.5.1. RESTful API endpoints

The BBP backend exposes a RESTful API consumed by the mobile application. All endpoints exchange JSON payloads over HTTPS. Unless otherwise specified, requests that access or modify user-specific resources require authentication via `Authorization: Bearer <access_token>`. Refresh tokens are used to renew expired access tokens.
Each endpoint definition reports the HTTP method, the resource path, the expected request parameters (query/path/body), and the possible responses. Standard HTTP status codes are adopted to represent successful outcomes and errors.

Auth

POST `/auth/login`

- **Request Body**: `{ email, password }`
- **Responses**:
  - **200 OK**: `{ success, user, tokens }`
  - **400 Bad Request**: `MISSING_CREDENTIALS`
  - **401 Unauthorized**: `INVALID_CREDENTIALS`
  - **404 Not Found**: `USER_NOT_FOUND`

POST `/auth/logout`

- **Request Body**: `{ refreshToken }`
- **Responses**:
  - **204 No Content**
  - **401 Unauthorized**: `MISSING_REFRESH_TOKEN`

POST `/auth/refresh`

- **Request Body**: `{ refreshToken }`
- **Responses**:

  - **200 OK**: `{ success, message, tokens }`
  - **401 Unauthorized**: `MISSING_REFRESH_TOKEN`
  - **403 Forbidden**: `INVALID_REFRESH_TOKEN`, `REFRESH_TOKEN_NOT_FOUND`, `REFRESH_TOKEN_`
  - **404 Not Found**: `USER_NOT_FOUND`

Users

POST `/users/register`

- **Request Body**: `{ email, password, username, systemPreferences? }`
- **Responses**:
  - **201 Created**: `{ success, user }`
  - **400 Bad Request**: `MISSING_CREDENTIALS`
  - **409 Conflict**: `EMAIL_ALREADY_IN_USE`, `USERNAME_ALREADY_IN_USE`

GET `/users/me`

- **Responses**:
  - **200 OK**: `{ success, data }`
  - **401 Unauthorized**: `USER_NOT_AUTHENTICATED`
  - **404 Not Found**: `USER_NOT_FOUND`

PATCH `/users/me`

- **Request Body**: `{ username?, email?, currentPassword?, password?, systemPreferen }`
- **Responses**:
  - **200 OK**: `{ success, message }`
  - **400 Bad Request**: `EMPTY_PAYLOAD`, `INCORRECT_PASSWORD`
  - **401 Unauthorized**: `USER_NOT_AUTHENTICATED`
  - **404 Not Found**: `USER_NOT_FOUND`
  - **409 Conflict**: `EMAIL_ALREADY_IN_USE`, `USERNAME_ALREADY_IN_USE`

Paths

POST `/paths`

- **Request Body**: `{ pathSegments, visibility, creationMode, title, description }`
- **Responses**:
  - **201 Created**: `{ success, message, data }`
  - **400 Bad Request**: `MISSING_PATH_SEGMENTS`, `MISSING_VISIBILITY`, `INVALID_SEGMENT`, `MISSING_CREATION_MODE`, `INVALID_CREATION_MODE`, `NOT_IMPLEMENTED`
  - **401 Unauthorized**: `USER_NOT_AUTHENTICATED`
  - **404 Not Found**: `PATH_NOT_FOUND_AFTER_CREATION`
  - **409 Conflict**: `PATH_ALREADY_EXISTS`

GET `/paths/search`

- **Query Params**: `origin`, `destination`
- **Responses**:
  - **200 OK**: `{ success, data }`
  - **400 Bad Request**: `MISSING_ORIGIN`, `MISSING_DESTINATION`
  - **404 Not Found**: `NO_ROUTE`

POST `/paths/snap`

- **Request Body**: `{ coordinates }`
- **Responses**:
  - **200 OK**: `{ success, data }`
  - **400 Bad Request**: `MISSING_COORDINATES`
  - **401 Unauthorized**: `USER_NOT_AUTHENTICATED`

GET `/paths?owner=me`

- **Responses**:
  - **200 OK**: `{ success, data }` (empty list allowed)
  - **401 Unauthorized**: `USER_NOT_AUTHENTICATED`

PATCH `/paths/:pathId/visibility`

- **Request Body**: `{ visibility }`
- **Responses**:
  - **200 OK**: `{ success, message, data }`
  - **400 Bad Request**: `MISSING_PATH_ID`, `MISSING_VISIBILITY`, `INVALID_VISIBILITY`
  - **401 Unauthorized**: `USER_NOT_AUTHENTICATED`
  - **403 Forbidden**: `FORBIDDEN`
  - **404 Not Found**: `NOT_FOUND`

DELETE `/paths/:pathId`

- **Responses**:
  - **204 No Content**
  - **400 Bad Request**: `MISSING_PATH_ID`
  - **401 Unauthorized**: `USER_NOT_AUTHENTICATED`
  - **403 Forbidden**: `FORBIDDEN`
  - **404 Not Found**: `NOT_FOUND`

Trips

POST `/trips`

- **Request Body**: `{ origin, destination, startedAt, finishedAt, tripSegments, title? }`
- **Responses**:
  - **201 Created**: `{ success, message, data }`
  - **400 Bad Request**: `MISSING_ORIGIN_DESTINATION`, `MISSING_DATES`, `INVALID_DATES`, `INVALID_DATE_RANGE`, `MISSING_TRIP_SEGMENTS`, `DUPLICATE_SEGMENT_IDS`
  - **401 Unauthorized**: `USER_NOT_AUTHENTICATED`

GET `/trips?owner=me`

- **Responses**:
  - **200 OK**: `{ success, data }` (empty list allowed)
  - **401 Unauthorized**: `USER_NOT_AUTHENTICATED`

DELETE `/trips/:tripId`

- **Responses**:
  - **204 No Content**
  - **400 Bad Request**: `MISSING_TRIP_ID`
  - **401 Unauthorized**: `USER_NOT_AUTHENTICATED`
  - **403 Forbidden**: `FORBIDDEN`
  - **404 Not Found**: `NOT_FOUND`

Reports

GET `/reports?pathId=:pathId`

- **Responses**:
  - **200 OK**: `{ success, data }` (empty list allowed)
  - **400 Bad Request**: `MISSING_PATH_ID`

POST `/reports`

- **Request Body**: `{ pathId, position, reportType, description?, media? }`
- **Responses**:
  - **201 Created**: `{ success, message, data }`
  - **400 Bad Request**: `MISSING_PATH_ID`, `MISSING_POSITION`, `INVALID_REPORT_TYPE`
  - **401 Unauthorized**: `USER_NOT_AUTHENTICATED`
  - **404 Not Found**: `PATH_NOT_FOUND`

POST `/reports/:reportId/confirm`

- **Request Body**: `{ decision }`
- **Responses**:
  - **201 Created**: `{ success, message, data }`
  - **400 Bad Request**: `MISSING_REPORT_ID`, `INVALID_DECISION`
  - **401 Unauthorized**: `USER_NOT_AUTHENTICATED`
  - **404 Not Found**: `REPORT_NOT_FOUND`

Stats

GET `/stats/overall`

- **Responses**:
  - **200 OK**: `{ success, data }`
  - **401 Unauthorized**: `USER_NOT_AUTHENTICATED`
  - **404 Not Found**: `NOT_FOUND`

2.6. Selected Architectural Styles and Patterns

The BBP system adopts a **three-tier, layered architecture** composed of a thin mobile client, an application tier implementing all business rules, and a relational data tier. This structure ensures clear responsibility separation, simplifies maintenance, and supports the mobile-first design goals described in the system overview.

- **API Entrypoint with modular backend**. All external requests pass through the **API Entrypoint**, which centralises routing, authentication, input validation and error normalisation. Behind it, the backend is organised as a collection of Managers, each encapsulating a coherent vertical slice of functionality (paths, reports, trips, users). This separation reduces coupling, facilitates parallel development, and keeps inter-component interactions lightweight.
- **RESTful, stateless communication**. The mobile app interacts with the backend exclusively through resource-based HTTPS endpoints. The application tier remains stateless for access tokens: session information is kept client-side through short-lived access tokens stored in secure storage, making the system horizontally scalable and resilient to server restarts. Refresh tokens are persisted in the DBMS to support logout and rotation, introducing minimal server-side state explicitly managed by the **AuthManager**.
- **Service Layer pattern**. Each Manager acts as a well-defined service boundary that hides internal logic, aggregates operations into meaningful methods, and exposes a stable API to the **API Entrypoint**. This pattern avoids exposing domain details to the controller layer and fosters maintainability and testability.
- **DAO pattern**. The **QueryManager** centralises access to the relational DBMS and hides SQL concerns from the rest of the backend. By enforcing consistency

checks and persistence rules in a single component, the system becomes less error-prone and more adaptable to future schema evolution or database optimisation.

- **Security patterns**. Authentication is based on JWT-like tokens validated by the **API Entrypoint** (middleware), while the AuthManager handles token issuance, refresh, and logout flows. Boundary validation prevents malformed inputs and mitigates injection risks. Authorisation is enforced by combining token claims with ownership checks on persisted resources, keeping backend nodes stateless and interchangeable while preserving access control guarantees.

2.7. Other Design Decisions

Beyond the architectural style, the following design choices strengthen the BBP system's quality attributes and align with the mobile-first scope:

- **Authentication and authorisation**. Access control relies on short-lived access tokens and refresh tokens securely stored on the mobile client. Refresh tokens are also persisted in the DBMS to enable validation, rotation, and logout. The API Entrypoint enforces authentication, while domain-level authorisation (guest vs authenticated user) is performed inside the Managers.

- **Validation and error handling**. Validation occurs at the system boundary (API Entrypoint) ensuring malformed (e.g. missing JWT, invalid JSON schema) or invalid endpoint requests are filtered before reaching the business layer. On the other hand, user input is validated when the request reaches the business layer, ensuring that incomplete or incorrect data will not be saved, and the user will be correctly notified. Errors are normalised into structured HTTP responses so the mobile client can present consistent messages. Domain-specific errors (e.g., `NOT_FOUND`) are explicitly mapped and surfaced through the **API Entrypoint**.

- **Fault tolerance**. Calls to external services (e.g., weather provider) are wrapped with timeouts, and explicit error handling. When external data cannot be retrieved, the system either returns a safe default (e.g., empty weather aggregation) or propagates a domain error depending on the operation. Even in case of partial failure, critical user actions (such as trip completion) are preserved and stored.

- **Data integrity**. The **QueryManager** centralises data access and consistency checks. Atomicity for multi-step operations is enforced at the application level by validating inputs before persistence and by performing guarded updates, while keeping SQL concerns encapsulated within the data-access layer.

3 | User Interface Design

All mockups presented in this chapter are **conceptual prototypes** designed to illustrate the expected interaction flow, the structural layout of each screen, and the different behaviours available to both guest and logged-in users. While the visual style is representative of the intended interface, refinements may occur during implementation.
This chapter provides a **high-level overview** of the system's user experience. Its purpose is not to specify implementation details, but to visually and conceptually describe how users navigate the application across its main functionalities: authentication, map exploration, path search and selection, trip navigation, obstacle reporting, custom path creation and management, and profile configuration. The mockups also highlight how certain capabilities vary depending on the user state, guest or logged-in, ensuring a clear understanding of the overall interaction model.

3.1. Navigation Flow

From a global perspective, the application is organised around a small set of core areas: the authentication entry point (Welcome, Login, Signup), the map-based Home Screen, trip-related features (navigation, reporting, history), path management features (creation and created paths), and the personal area (Profile, Edit Profile, Settings). The bottom navigation bar connects Home, Trip History, Path Library, and Profile, and adopts a modern auto-hiding behaviour: it becomes hidden while scrolling downward to maximise vertical space and reappears when the user scrolls upward, reinforcing spatial orientation without obstructing content.
Navigation begins at the **Welcome Screen**, where users may authenticate or proceed as guests. In either case, they are redirected to the **Home Screen**, which functions as the central hub for all subsequent interactions. From here, users can search for paths, explore search results, begin trip navigation along a selected route, or, if logged-in, initiate the creation of a custom path. During active navigation, the interface transitions to a dedicated full-screen **Trip Navigation View**. Once the trip is completed or interrupted, the interface automatically returns to the map-based **Home Screen**. Trip exploration is accessible at any moment through the **Trip History Screen**, reachable via the navigation

bar. Here, users can review previous activities and inspect detailed trip information. A similar list-and-detail pattern governs the **Path Library Screen**, which enables users to manage their custom paths, start navigation from a created route, or adjust visibility and deletion settings. Account-related functionality is grouped within the **Profile Screen** and its associated views (**Edit Profile** and **Settings**).

Across all flows, the application clearly differentiates between guest and logged-in users: guests may browse the map and perform basic navigation, while logged-in users gain access to trip recording, obstacle reporting and validation, custom path creation and management, and trip statistics. Several cross-cutting behaviours, such as automatic ride detection, authentication prompts for restricted actions, and global error pop-ups, may appear on top of the current interface without altering the underlying navigation structure, ensuring that interaction remains consistent and predictable across different usage scenarios.

3.2. Authentication and Guest Access

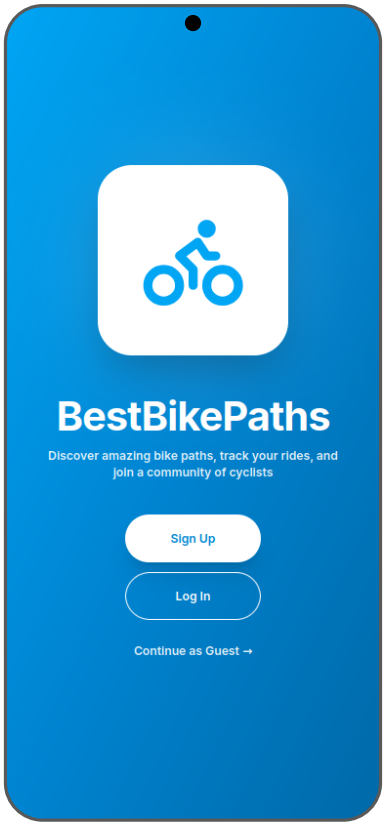

Figure 3.1: Welcome Screen

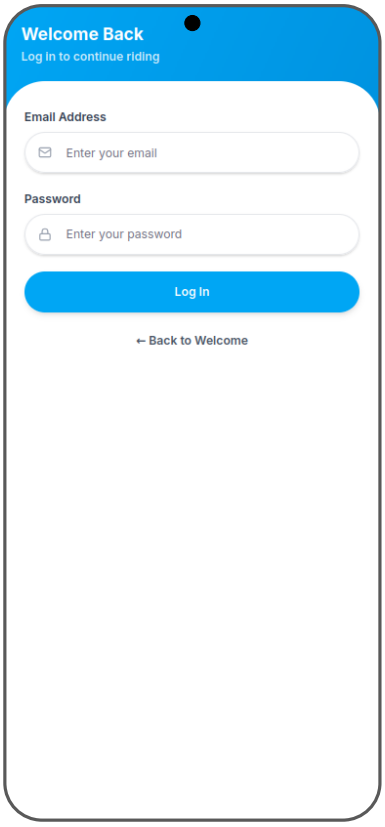

Figure 3.2: Login Screen

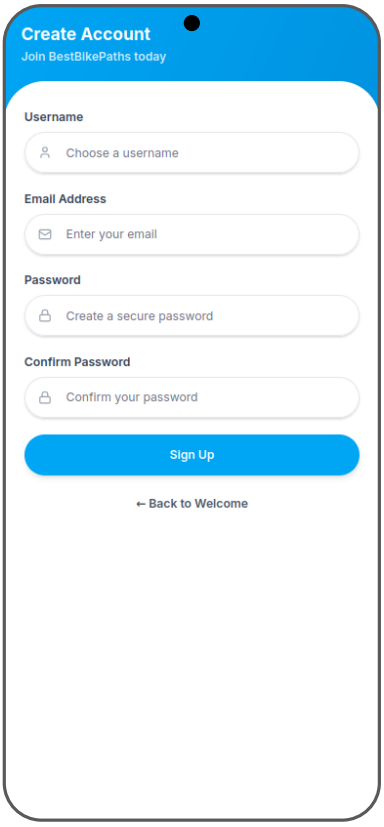

Figure 3.3: Signup Screen

Welcome Screen

The **Welcome Screen** introduces the user to the application through a minimal layout featuring the app logo, name, and a short tagline summarising its purpose: discovering bike paths, tracking rides, and engaging with the cycling community. Three navigation options are provided: **Sign Up**, **Log In**, and **Continue as Guest**. The first leads new users to the registration flow, the second allows returning users to access their existing account, and the third offers immediate exploration with limited functionalities.
This screen acts as the entry point to all subsequent interactions. It does not require backend communication and is intentionally designed to be simple, guiding users toward authentication or guest access in an intuitive way.

Login Screen

After selecting the **Log In** option from the Welcome Screen, users are taken to the Login Screen, where they can authenticate by entering their email address and password. A

secondary action allows navigation back to the Welcome Screen.
Successful authentication unlocks all features requiring a validated identity, including trip recording with persistent storage, obstacle reporting, custom path creation, and access to personal statistics. The screen focuses on clarity and ease of use, maintaining a clean layout that prioritises the login form.
Input validation and authentication errors are handled through inline messages and global pop-ups, without altering the navigation flow.

Signup Screen

The **Signup Screen** enables new users to create an account by entering a username, email address, password, and password confirmation. Input validation errors are handled through inline messages. The interface intentionally keeps the registration process lightweight, requesting only the essential information required for profile creation. Users may return to the Welcome Screen through the dedicated navigation control.
Creating an account grants access to all core functionalities of the system: trip recording, obstacle reporting and validation, custom path management, and statistics, providing a complete and personalised experience.

3.3. Home for Logged-in and Guest Users

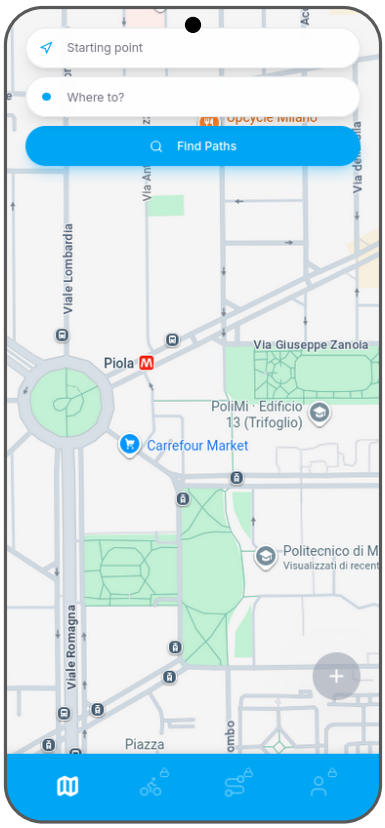

Figure 3.4: Home Screen

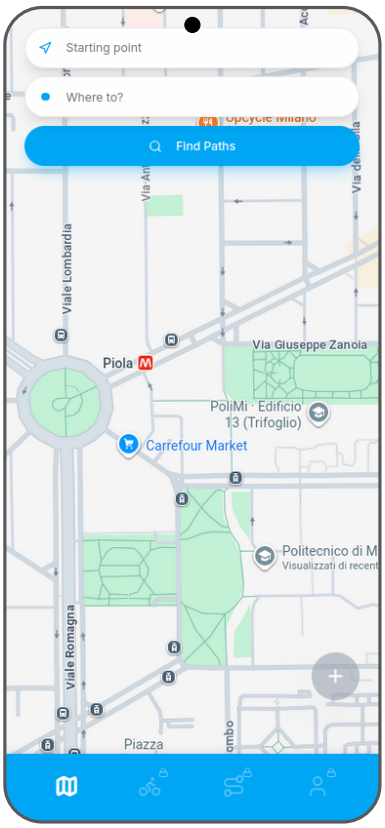

Figure 3.5: Home Screen for Guests

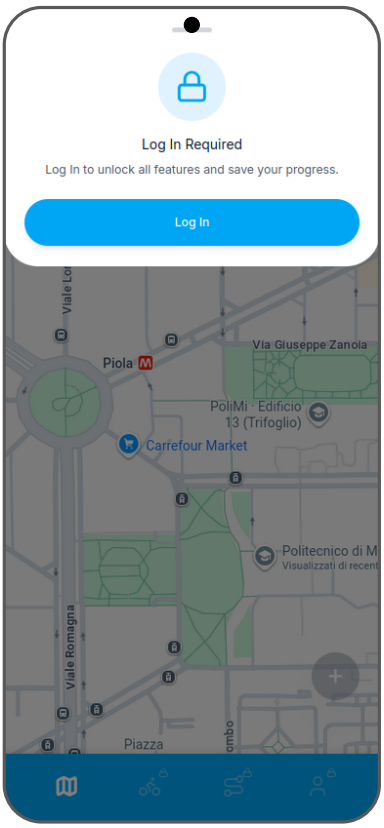

Figure 3.6: Authentication Pop-up

Home Screen

The **Home Screen** serves as the central hub of the BBP application, providing access to map exploration, path search, and all major navigation routes. At the top of the interface, users can specify a Starting Point and a Destination through two input fields. By default, the Starting Point is set to the user's current GPS location, but it may be manually edited when needed. Once both fields are filled, the **Find Paths** button retrieves all available cycling routes connecting the selected points.

The interactive map occupies most of the screen, displaying the user's position along with any computed paths. In the lower-right corner, authenticated users see a floating action button with a plus icon, which opens the custom path creation flow. For guest users, this appears visually disabled.

A persistent bottom navigation bar grants quick access to the core sections of the application: **Home**, **Trip History**, **Path Library**, and **Profile**.

Guest users may freely interact with the map and perform path searches. However, features requiring authentication, such as path creation, viewing personal trip history, or accessing the profile, are greyed out. A lock icon visually indicates restricted access. Attempting to select a disabled icon triggers an authentication pop-up inviting the user to log in. This design clearly communicates the distinction between publicly accessible features and those reserved for authenticated users.

Authentication Pop-up for Guest Users

When a guest user attempts to access a restricted capability, the system displays a modal pop-up explaining that the selected functionality is available only to logged-in users. The modal presents one primary action, **Log In**, which redirects the user to the Login Flow. While the pop-up is visible, the underlying interface is dimmed and temporarily disabled to focus the user's attention on the modal and prevent accidental interaction with the map or navigation controls. The pop-up may be closed by tapping outside the modal area, returning the guest user to their current context and allowing them to continue exploring the application uninterrupted.

3.4. Path Search

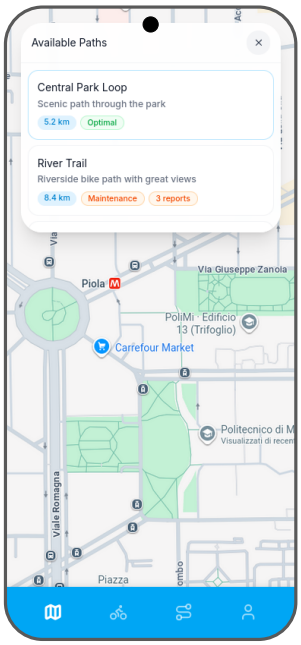

Figure 3.7: Search Results

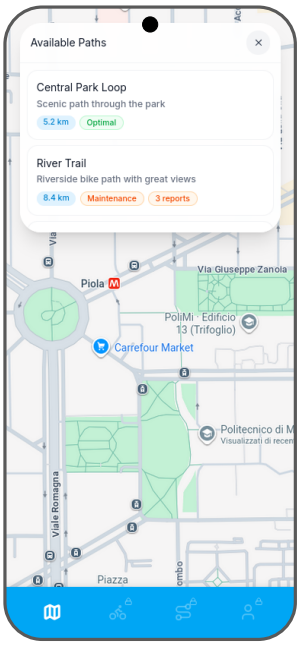

Figure 3.8: Search Results for Guests

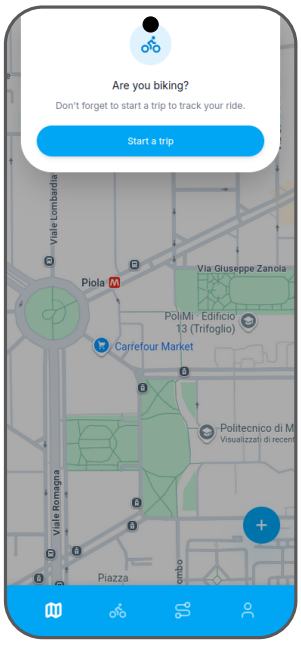

Figure 3.9: Ride Detection

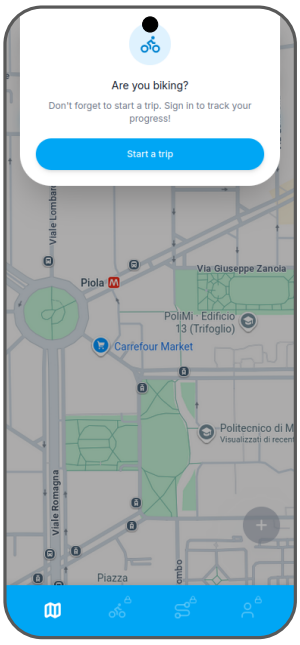

Figure 3.10: Ride Detection for Guests

Search Results

After submitting a path search from the **Home Screen**, the application displays an **Available Paths** panel anchored to the top of the map. The panel lists all suggested routes that match the selected criteria. Each entry shows the path name, a short description when available, the estimated distance and the current condition of the path (e.g., Optimal, Maintenance), accompanied by the number of aggregated reports, if any.
The underlying map remains visible as contextual background, centered on the queried area, allowing users to visually understand where each suggested route is located while scrolling through the list. A close icon in the upper-right corner dismisses the panel and restores the standard map view, enabling the user to adjust the search parameters or run a new query.
The behaviour is the same for logged-in and guest users, aside from the general limitations applicable to guests throughout the application.

Automatic Ride Detection

While the application is open, BBP continuously monitors movement patterns to detect whether the user is biking without having manually started a trip. When such behaviour

is identified, the system displays a modal prompt at the top of the map asking: **"Are you biking?"**. This prompt serves as a reminder to begin tracking so that the ride can be recorded properly.

The pop-up dims the rest of the interface and presents a single primary action. Selecting it initiates the standard trip-start flow. Tapping outside the modal closes the prompt and returns the user to the active screen without starting a trip.

Detection and interaction behave identically for guests and authenticated users. However, only logged-in users will have their recorded trips saved and included in their personal statistics, consistent with the restrictions outlined in other parts of the interface.

3.5. Path Selection and Trip Start

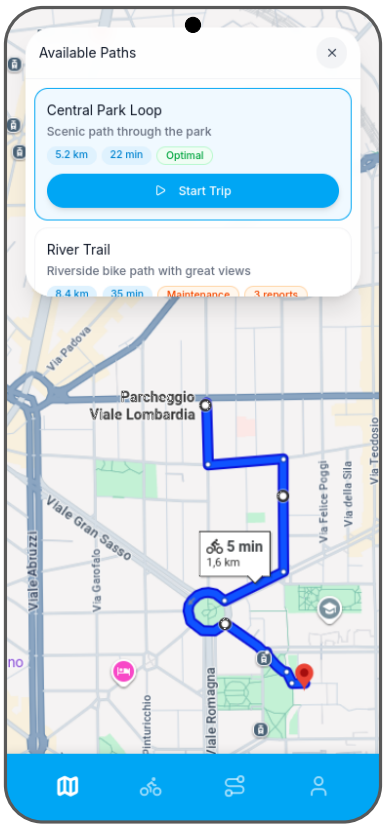

Figure 3.11: Path Selection

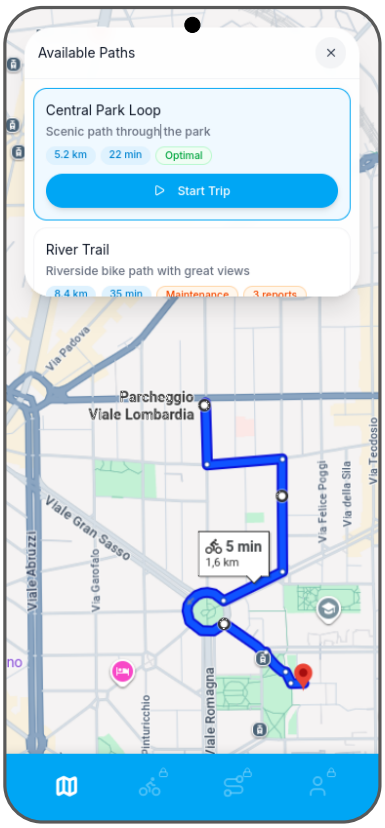

Figure 3.12: Path Selection for Guests

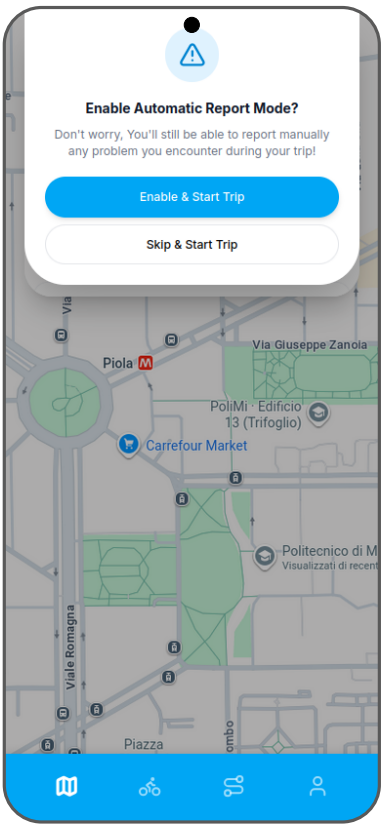

Figure 3.13: Automatic Mode Activation

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 identifies the currently active path while keeping the rest of the list unchanged. Tapping a different card updates the selection, tapping the same card again deselects it, and tapping outside the panel leaves the current state unchanged.

Once a path is selected, the system displays the corresponding route on the map using a bold blue polyline. This provides an immediate visual representation of the full route. If the path includes confirmed reports, the associated markers are also rendered on the map, enabling users to identify potentially problematic areas along the way.

A **Start Trip** button appears within the selected card only when the path origin matches the user's current GPS position. If the origin differs, the user may still inspect the route preview, but the trip cannot be started from the current location.

When the Start Trip button is available, selecting it opens a pop-up for logged-in users asking whether they wish to enable Automatic Report Mode. After choosing a mode, the trip begins.
Guest users may start navigation from a selected path, but their activity is not stored.
Pressing the close icon at the top of the results panel dismisses the list and restores the standard map view without an active selection.
Map visualisation, path highlighting, and browsing behaviour are consistent across guest and authenticated users. However, only logged-in users may access advanced features from the bottom navigation bar, which remains visually disabled for guests.

Automatic Mode Activation

After tapping the Start Trip button, logged-in users are presented with a pop-up asking whether they wish to enable Automatic Report Mode. When activated, this mode leverages the device's sensors (gyroscope and accelerometer) to detect anomalies such as potholes or rough surfaces. Upon detecting an irregularity, BBP generates a pre-filled report and displays a confirmation prompt for the user to review, edit, or submit.
Once the user proceeds, the trip begins and the system starts tracking movement along the selected route. Automatic detection does not replace manual reporting: users may submit manual reports at any time during the trip through the dedicated interface.

3.6. Navigation and Trip Completion

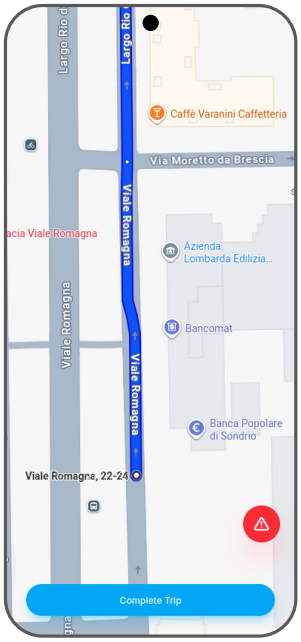

Figure 3.14: Navigation View

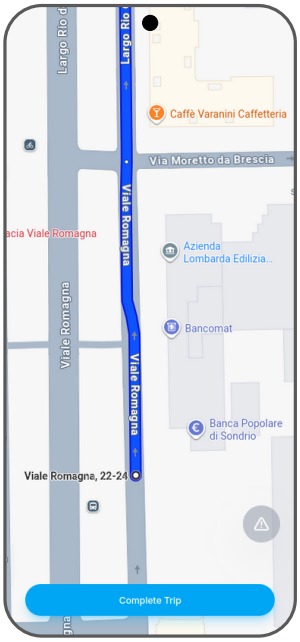

Figure 3.15: Navigation View for Guests

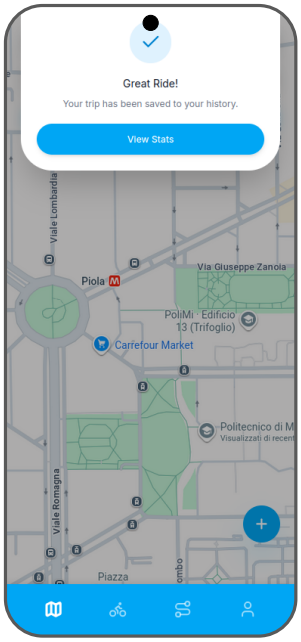

Figure 3.16: Trip Completion

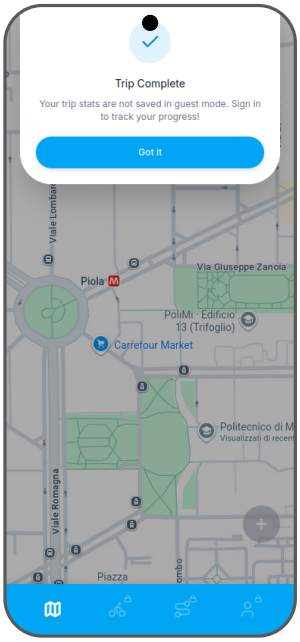

Figure 3.17: Trip Completion for Guests

Navigation View

During an active trip, the interface transitions to a dedicated full-screen **Navigation View**. The map occupies the entire display, showing the user's real-time position and the selected path highlighted with a thick blue polyline. Bottom navigation elements are hidden in this mode to ensure that the user remains fully focused on the ongoing ride.

A **Complete Trip** button is anchored at the bottom of the screen, allowing the user to manually end the session at any time. On the lower-right side, a floating report button is shown: red and active for logged-in users, enabling manual obstacle reporting, greyed out for guest users, who cannot submit reports. Attempting to press the disabled button triggers an authentication pop-up, inviting the guest to sign in. The floating placement and colour clearly communicate whether reporting is available.

A trip ends when the user presses the **Complete Trip** button, upon reaching the destination, or when the system detects a significant deviation from the planned route. Once the trip terminates, BBP stops tracking and displays a confirmation message.

For authenticated users, the recorded trip is stored in their personal history. Guest users experience the same navigation interface, but no information is saved after the session

concludes.
This view visually reinforces the functional distinction between user types: while both may follow a selected route, only logged-in users can submit reports and preserve their trip data.

Trip Completion

When a trip ends, regardless of whether it was completed manually, automatically, or due to route deviation, the system displays a completion pop-up summarising the outcome of the session.
For logged-in users, the pop-up shows the message **"Great Ride!"** along with a **View Stats** button. Selecting this option redirects the user to the Trip History Screen and automatically opens the detailed summary of the newly completed trip. Tapping outside the modal simply closes it and restores full map interaction.
For guest users, the pop-up displays **"Trip Complete"** together with a reminder that trip statistics are not saved in guest mode. A single button closes the dialog, and no option to view statistics is offered since guest sessions are never stored.
In all cases, the underlying map is dimmed while the modal is visible, preventing interaction until the user acknowledges the completion message.

3.7. Reports

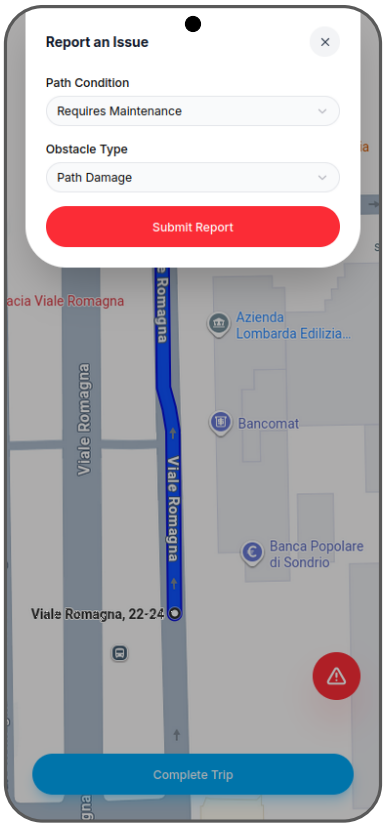

Figure 3.18: Report Submission

Figure 3.19: Successful Report

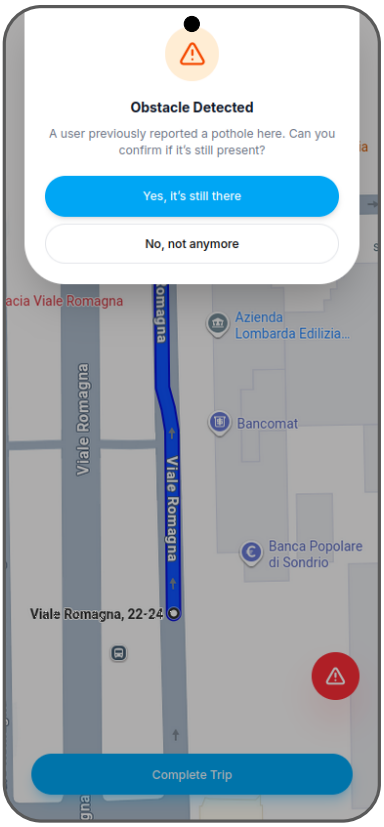

Figure 3.20: Report Confirmation

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 prompt. The pop-up overlays the current Navigation View and temporarily pauses map interaction by dimming the background.

The dialog contains two fields: **Path Condition**, allowing the user to indicate the 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 completed before sending the report. For automatically detected events, the fields are pre-filled with values inferred from sensor data, but the user may freely adjust the information, confirm it, or discard the prompt entirely.

After the report is submitted, the system displays a confirmation pop-up titled **"Report Submitted"**, thanking the user for contributing to the community. A **Continue Trip**

button closes the dialog and returns the user to the Navigation View. If the user does not interact with the confirmation pop-up, it automatically dismisses after a short timeout.
Report submission is available only to logged-in users. In guest mode, the report button remains visible for consistency but appears greyed out and cannot be pressed.

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 such cases, a confirmation pop-up appears at the top of the screen with the message **"Obstacle Detected"**, indicating that an issue was reported at that point.
The dialog asks the user to confirm whether the obstacle is still present, offering two options: one to validate that the issue persists and one to indicate that it is no longer observed. The underlying map is dimmed to draw attention to the dialog while keeping the navigation context visible. If the user does not respond within a short timeout, the pop-up automatically closes to avoid interrupting the ride.
After the user selects an option, a brief confirmation pop-up appears to acknowledge that their response has been recorded. This secondary dialog automatically closes after a short delay if the user does not interact with it.
This feature is available exclusively to logged-in users. Guest users do not receive confirmation prompts and cannot validate existing reports.

3.8. Path Creation for Logged-in Users

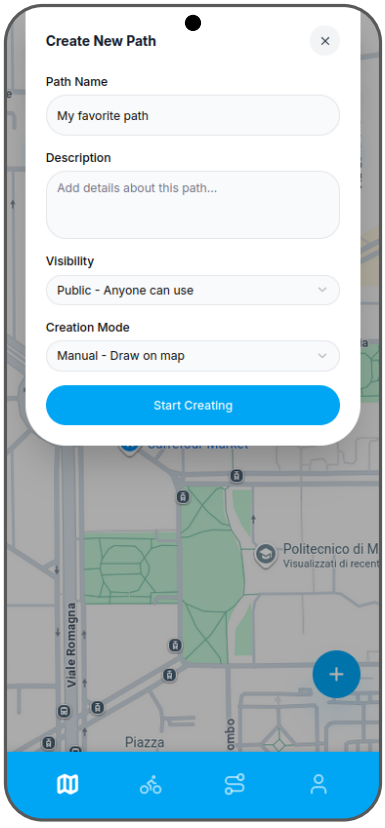

Figure 3.21: Path Creation

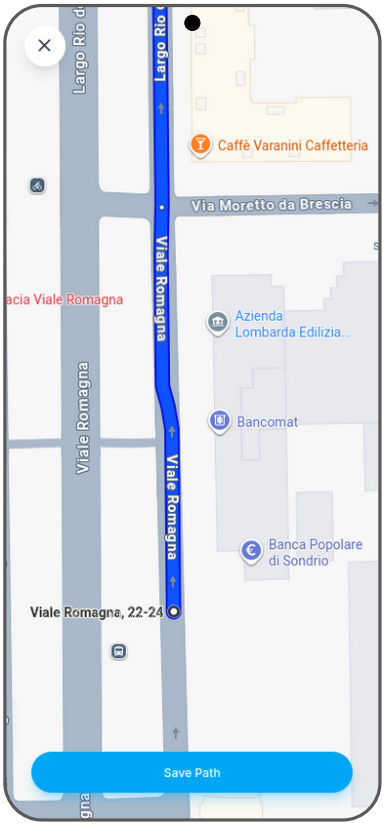

Figure 3.22: Creation View

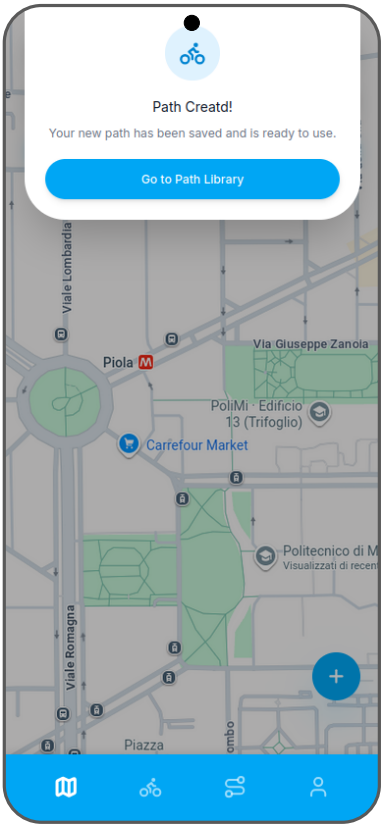

Figure 3.23: Successful Creation

Path Creation

When the user selects the path creation feature, the system displays a pop-up form at the top of the screen requesting the basic information required to define a new bike path. The form includes: **Path Name**, where the user provides the title of the route, **Description**, an optional field for adding additional details, **Visibility**, allowing the user to choose whether the path is public or private, and **Creation Mode**, which determines how the path will be constructed, either manually by drawing it on the map or automatically by recording the GPS trace during a ride.

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

Path creation is available exclusively to logged-in users. This feature is disabled in guest

mode.

Creation View

After the user begins constructing a new path, the interface transitions into a dedicated full-screen Creation View. The map expands to occupy the entire display and all navigation elements are hidden to ensure an uninterrupted creation experience.
The behaviour of this view depends on the selected creation mode. In **Automatic Mode**, the map shows the user's real-time position while cycling. As the user moves, BBP draws a blue polyline that represents the recorded segment of the path, continuously updating to reflect the evolving GPS trace. This provides an immediate visual representation of the route as it is being generated.
In **Manual Mode**, the user defines the route by interacting directly with the map. Each waypoint placed on the map extends the blue polyline, allowing the user to progressively shape the geometry of the final path. This mode offers full manual control over the route layout.
At any time, the user may cancel the creation process by tapping the button in the upper-left corner, which closes the Creation View and discards all progress. When satisfied with the constructed route, the user can save it by tapping the **Save Path** button at the bottom of the screen. This action stores the path along with the metadata previously entered in the creation form.
Access to the Creation View and the ability to save paths are restricted to logged-in users.

Creation Completion

After the new path has been saved, the system displays a confirmation pop-up titled **"Path Created!"**. The dialog informs the user that the route has been successfully stored and is now available for use.
The pop-up includes a single action button, **Go to Path Library**, which redirects the user to the section containing all the paths they have created. Here, they may inspect the newly stored path in detail or manage their collection. If the user taps outside the modal, the pop-up simply closes and the application returns to the map view.
This confirmation appears only for logged-in users, as guests cannot create or save custom paths.

3.9. Trip Management for Logged-in Users

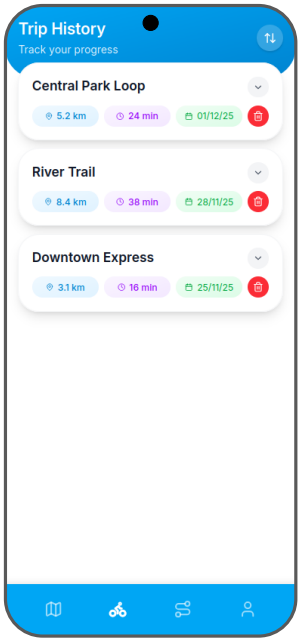

Figure 3.24: Trip History

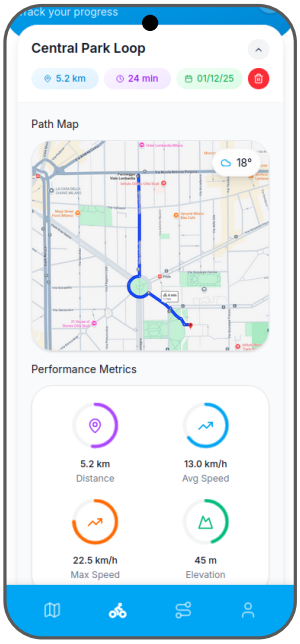

Figure 3.25: Trip Details

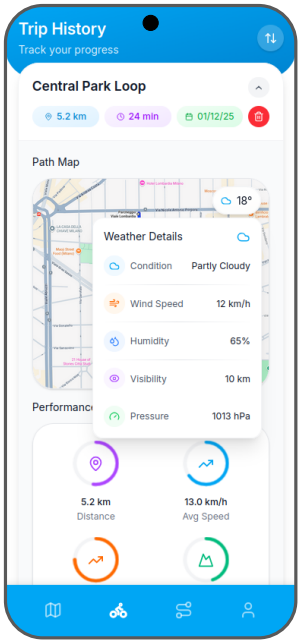

Figure 3.26: Trip Weather

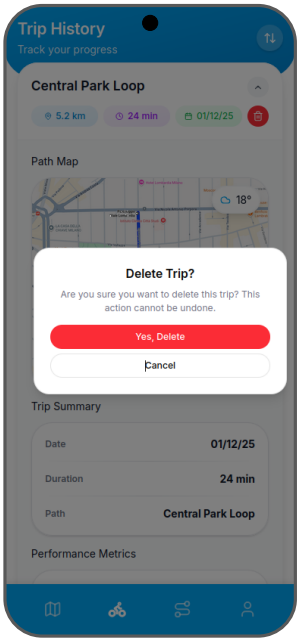

Figure 3.27: Trip Deletion

The **Trip History Screen** is accessible through the second icon in the bottom navigation bar and allows logged-in users to review all previously recorded cycling activities. At the top of the interface, a sorting control is placed beside the screen title. When tapped, it opens a compact dropdown menu that lets the user reorder the list of trips by date, distance, duration, or alphabetical order. Selecting an option immediately updates the list to reflect the chosen order.

Trips are presented as compact cards displaying the essential information for each session: trip name, total distance, duration, and completion date. A delete icon appears on each card, giving users the ability to remove a trip from their history. Tapping the icon triggers a confirmation pop-up to prevent accidental deletions.

Tapping a trip card expands it smoothly into a detailed view. In this state, the interface displays a map preview showing the route taken during the session, enriched with a weather badge placed in the corner of the map. Beneath the map, a performance panel displays additional metrics such as distance, average speed, maximum speed, and elevation.

Tapping the weather badge reveals a detailed meteorological panel containing information such as temperature, humidity, wind speed, visibility, pressure, and overall weather conditions at the time of the trip. If the recorded activity includes confirmed reports,

their corresponding markers appear along the displayed route. Selecting a marker opens a small dialog with details about the associated issue, allowing the user to inspect what was encountered during the ride.
Tapping the trip header collapses the view and returns 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.
Trip History is available exclusively to logged-in users, as guest users cannot store or view past trips.

3.10. Paths Management for Logged-in Users

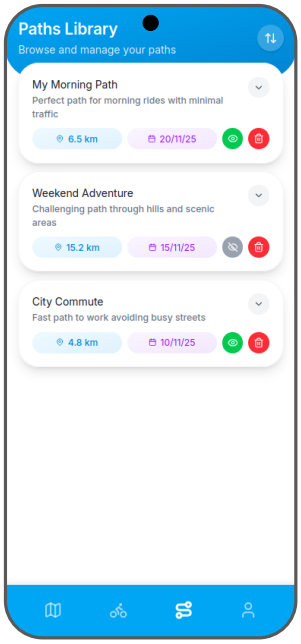

Figure 3.28: Paths Screen

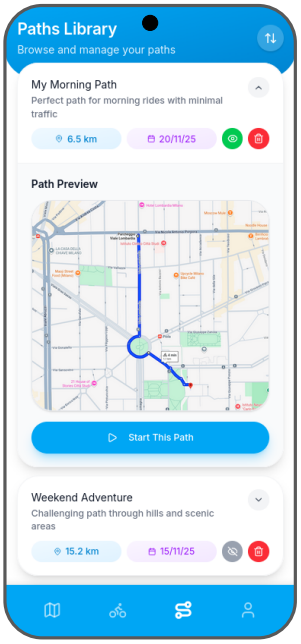

Figure 3.29: Path Details

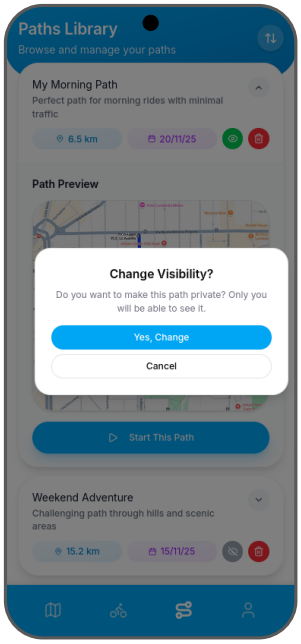

Figure 3.30: Path Visibility

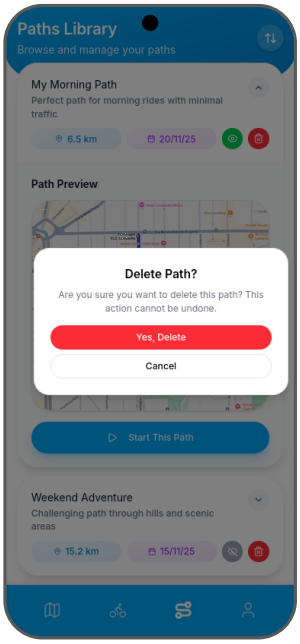

Figure 3.31: Path Deletion

The **Path Library 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 interface, 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 immediately updates to reflect the chosen ordering.
Each path is presented as a compact card containing the title, a short description, the total distance, the creation date, and two management icons: one for toggling visibility and one for deletion. Tapping a card expands it into a detailed view that includes a map preview displaying 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 starting point of the path, the trip can be started immediately.
Management actions trigger dedicated confirmation dialogs. Changing visibility opens a pop-up asking the user to confirm whether they wish to switch the path between public and private. The update is applied only after explicit confirmation. Deleting a path displays a separate confirmation dialog warning that the action is permanent, the path is removed only if the user confirms the request.
Collapsing an expanded entry restores the compact list layout, allowing users to browse

their collection efficiently.
All interactions within this screen are available exclusively to logged-in users. Guest users cannot view, create, start, or manage custom paths.

3.11. Profile and Settings for Logged-in Users

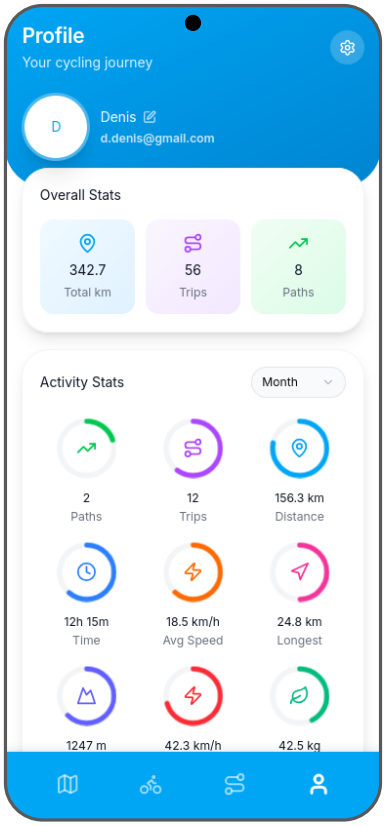

Figure 3.32: Profile Screen

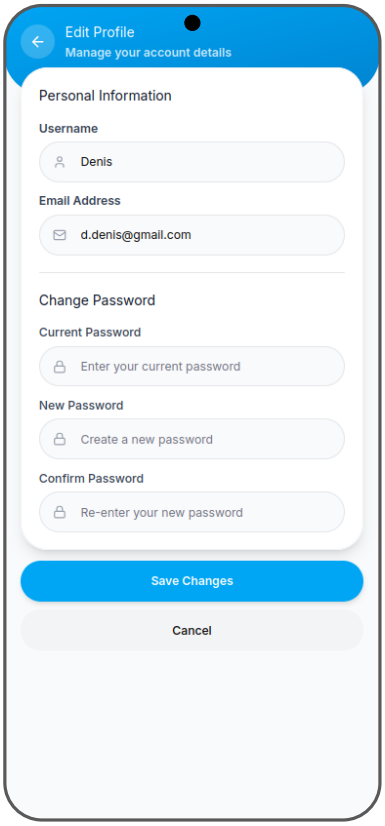

Figure 3.33: Edit Profile

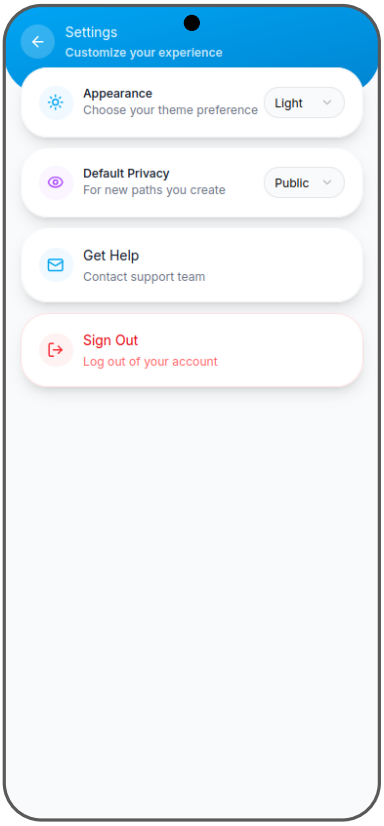

Figure 3.34: Settings

Profile Screen

The Profile Screen provides logged-in users with an 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, a profile header displays the user's avatar, name, and email address. A small edit icon next to the name allows the user to update their personal information, while a settings icon in the upper-right corner redirects to the Settings Screen, where account preferences and configuration options can be adjusted.

Below the header, the **Overall Stats** section summarises long-term activity metrics, including total distance ridden, total recorded trips, and the number of custom paths created. These values provide a compact snapshot of the user's cumulative progress.

Further down, the **Activity Stats** section offers a more granular breakdown. A compact dropdown menu allows users to filter statistics by time period, such as monthly activity. Displayed metrics may include paths and trips completed within the selected period, total

distance, riding time, average and maximum speed, and other performance indicators. Each metric is paired with a colourful icon to support quick visual identification.
All information on this screen is available exclusively to logged-in users. Guest users cannot access the Profile Screen or view personal statistics.

Edit Profile

The Edit Profile Screen enables 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 specify a New Password, and confirm it in the Confirm Password field. This ensures the new credentials are entered correctly before submission.
After making the desired changes, the user can tap **Save Changes** to apply them. The app validates the input and, if successful, stores the updated information and returns the user to the Profile Screen. A **Cancel** button allows the user to dismiss the form without saving.
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 may return to the Profile view at any time by tapping the back arrow in the upper-left corner.

Settings

The Settings Screen allows logged-in users to customise their app experience and manage personal preferences. It is accessible from the Profile Screen through the settings icon in the top-right corner.
At the top of the page, the **Appearance** option lets users choose the visual theme of the application. The current theme is displayed on the right side of the row, and tapping it opens a selector where users may switch between available modes (e.g., Light or Dark).
Below this, the **Default Privacy** setting specifies the visibility applied to newly created paths. Users may choose whether new paths should be public or private by default. This preference is automatically applied during path creation but can still be overridden.
The **Get Help** section provides direct access to the support channel, allowing the user to contact the team if assistance is required.
Finally, a **Sign Out** button appears at the bottom of the screen. Selecting it logs the user out and returns the application to guest mode.
This screen is available exclusively to logged-in users, as guest users do not manage appearance settings, default visibility, or account-related actions.

3.12. Error Messages

The application includes a global error-handling mechanism that provides consistent feedback whenever an unexpected issue occurs. In such cases, a dedicated **Error** pop-up appears at the top of the screen, displaying a clear title and a short description of the problem. The message shown in the dialog is dynamically replaced with context-specific information, such as network failures, permission issues, or invalid operations.

When the error dialog is visible, the underlying interface is dimmed to draw the user's attention to the alert. The pop-up includes a single **Close** button that dismisses the message and returns the user to the previous screen. Tapping outside the pop-up also closes the notification.

This error dialog may appear at any point in the application, including during path searches, trip management, report submission, navigation, or when modifying custom paths, ensuring that errors are communicated in a consistent and predictable manner.

Figure 3.35: Error Pop-up

4 | Requirements Traceability

Rx	Description	Components
R1	The system shall allow guest users to create an account by providing personal information and credentials.	AuthManager, QueryManager
R2	The system shall allow registered users to log into the application using valid credentials.	AuthManager, QueryManager
R3	The system shall allow logged-in users to view their profile and account settings.	UserManager, QueryManager
R4	The system shall allow logged-in users to update their profile and account settings.	UserManager, QueryManager
R5	The system shall allow logged-in users to log out of the application, ending their current session.	AuthManager, QueryManager
R6	The system shall allow guest users to start a cycling trip.	(*)
R7	The system shall allow guest users to stop a currently active trip, but shall not store any trip data after the trip ends.	(*)
R8	The system shall allow the user to start a trip only when their GPS position matches the path origin.	TripManager
R9	The system shall display a pop-up suggesting to start a trip when cycling is detected while no trip is active and the app is open.	TripManager
R10	The system shall set the current GPS position as trip origin when starting from auto-detection.	TripManager
R11	The system shall automatically stop the active trip when the user's GPS position deviates from the selected path within a certain threshold.	TripManager

Continued on next page

Rx	Description	Components
R12	The system shall allow logged-in users to start a cycling trip in manual or automatic mode.	TripManager
R13	The system shall allow logged-in users to stop a currently active trip and save the recorded data.	TripManager, QueryManager, WeatherManager
R14	The system shall collect GPS data during trip recording.	TripManager
R15	The system shall collect motion sensor data (accelerometer, gyroscope) during trip recording only when Automatic Mode is enabled.	TripManager
R16	The system shall allow logged-in users to view the list of their recorded trips.	TripManager, QueryManager
R17	The system shall allow logged-in users to view a summary of their overall cycling statistics (total distance, total time, average speed, etc.).	StatsManager, QueryManager
R18	The system shall allow logged-in users to view statistics for each trip (distance, speed, duration, etc.).	StatsManager, QueryManager
R19	The system shall display the route and reported obstacles associated with a recorded trip.	TripManager, PathManager, ReportManager, QueryManager
R20	The system shall allow logged-in users to delete a recorded trip.	TripManager, QueryManager
R21	The system shall communicate with external weather services to retrieve meteorological data related to the time and location of a trip.	TripManager, WeatherManager
R22	The system shall detect when a user is cycling based on speed and acceleration patterns.	TripManager
R23	The system shall detect irregular movements from sensor data that may suggest potholes or surface defects when Automatic Mode is enabled.	ReportManager

Continued on next page

Rx	Description	Components
R24	The system shall present automatically detected path and obstacle data to the logged-in user for manual confirmation before publishing.	ReportManager, QueryManager
R25	The system shall allow logged-in users to manually create a new bike path by drawing segments.	PathManager, QueryManager
R26	The system shall allow logged-in users to manually report obstacles or problems on a bike path while performing an active trip.	ReportManager, QueryManager
R27	The system shall allow logged-in users to manually confirm or reject the presence of obstacles reported by other users.	ReportManager, QueryManager
R28	The system shall allow logged-in users to create a new bike path in automatic mode using GPS tracking.	PathManager, QueryManager
R29	The system shall allow logged-in users to delete their previously created paths.	PathManager, QueryManager
R30	The system shall allow logged-in users to set the visibility of their created paths as public or private.	PathManager, QueryManager
R31	The system shall aggregate multiple user reports referring to the same path segment.	ReportManager, PathManager, QueryManager
R32	The system shall evaluate the reliability of each path segment based on the number of confirmations and report freshness.	ReportManager, PathManager
R33	The system shall determine the current status of a path (optimal, medium, sufficient, requires maintenance, closed).	ReportManager, PathManager
R34	The system shall allow any user (guest or logged-in) to view the detailed status and latest reports of a selected bike path.	PathManager, ReportManager, QueryManager

Continued on next page

Rx	Description	Components
R35	The system shall allow any user (guest or logged-in) to browse available public bike paths on a map.	PathManager, QueryManager
R36	The system shall allow any user to search for bike paths connecting two locations.	PathManager, QueryManager
R37	The system shall compute suggested routes based on path quality and distance.	PathManager
R38	The system shall rank suggested routes according to their safety and quality.	PathManager
R39	The system shall display the user's current GPS position during navigation along a selected path.	TripManager, PathManager
R40	The system shall send pop-ups to warn users about nearby obstacles or closed path segments during an active trip.	TripManager, ReportManager
R41	The system shall interface with map and geocoding services to translate addresses into coordinates and render paths.	PathManager, QueryManager
R42	The system shall ensure that communication with all external services (map, weather) handles temporary unavailability gracefully.	PathManager, WeatherManager

Table 4.1: Mapping between BBP Requirements and Backend Components

(*) The trip management functionality for guest users is limited to starting and stopping trips without data persistence. No communication with backend components occurs in this mode.

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 and testing strategy is a mix of a **Thread** approach and a **Bottom-Up** approach. In practice, we develop and test starting from low-level components and then progressively move upward toward more complex business-logic components.

By following a thread-based implementation, we can build and validate independent components **quickly and in parallel**, and start integration testing early instead of waiting for major modules (e.g., *PathManager* or *ReportManager*) to be fully completed. At the same time, for components that require more time and have strong dependencies such as *TripManager*, which relies on *PathManager* and *StatsManager*, or *ReportManager*, which depends on both *PathManager* and *TripManager*, a Bottom-Up progression is more suitable, since these modules can be integrated only once the underlying layers and required managers are stable.

This combined strategy helps us identify and resolve integration issues earlier, reducing the risk of costly problems emerging later in development.

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. For this purpose, we will use the free tier cloud version of PostgreSQL offered by Prisma.
- **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. Inside the QueryManager, we will implement the necessary CRUD operations as we implement each specific manager.
2. **External Services Integration**: Implementation of the `Geocoding`, `Weather` and `Snapping` services for communication with external weather APIs.
3. **Core Services**: Development of `AuthManager` and `UserManager` for identity management.
4. **Business Logic Services**: `PathManager` can be implemented independently. Then we will develop `TripManager`, `StatsManager`, and `ReportManager` in this order.
5. **Interface Layer**: Definition of REST endpoints and configuration of the external access layer (Cloudflare for DNS/TLS termination and NGINX reverse proxy routing to the backend).
6. **Presentation Layer**: Development of the `Mobile App` (developed in parallel using Mock APIs in the initial stages).

5.3. Integration Plan

Each `Manager` denotes a backend module within the Application Tier (i.e., part of the BBP backend), rather than an independently deployed microservice. Each phase integrates one or more modules into the existing subsystem and verifies their correct behavior through a dedicated test driver that simulates invocations from higher layers. Following a

mixed *thread-based* and *bottom-up* strategy, we integrate independent components early, while managers with stronger dependencies are added progressively on top of already stable lower-level modules. Whenever external dependencies are involved (e.g., Weather/Geocoding/Snapping APIs), stubs/mocks are adopted to keep tests reproducible.
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 correct connectivity, schema constraints, transactions, and the manipulation of persistent data. Additionally, database migrations will be applied automatically as part of the test setup to ensure a clean and consistent initial state across runs.

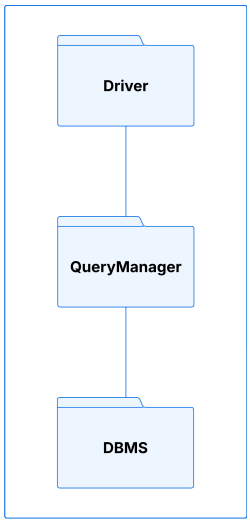

Figure 5.1: Step 1: DBMS and QueryManager Integration

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, 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, revoked/expired tokens) are handled consistently.

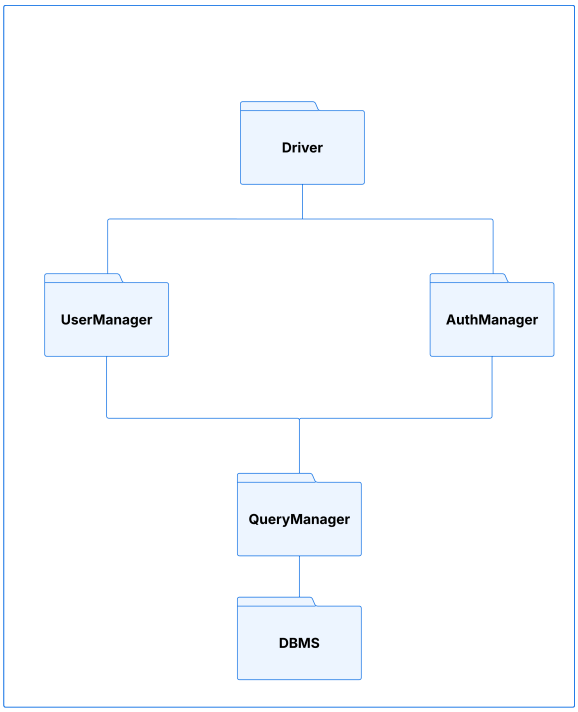

Figure 5.2: Step 2: AuthManager and UserManager Integration

In parallel, we integrate the external services used by the backend, namely `Weather`, `Geocoding`, and `Snapping`. Since these components primarily interact with external APIs and have minimal internal dependencies, they can be also tested independently. The driver will rely on stubbed endpoints to simulate both successful and failing API calls, verifying correct request formatting, response parsing, and robust handling of timeouts and HTTP errors. Meaningful error codes must be propagated when an external service is unavailable.

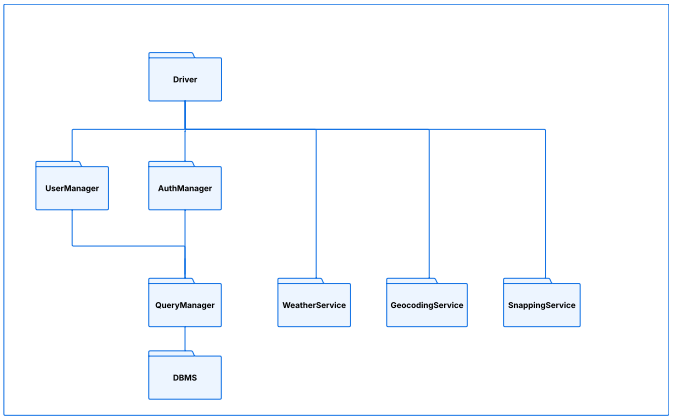

Figure 5.3: Step 3: External Services Integration (Weather/Geocoding/Snapping)

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, stale or incomplete data.

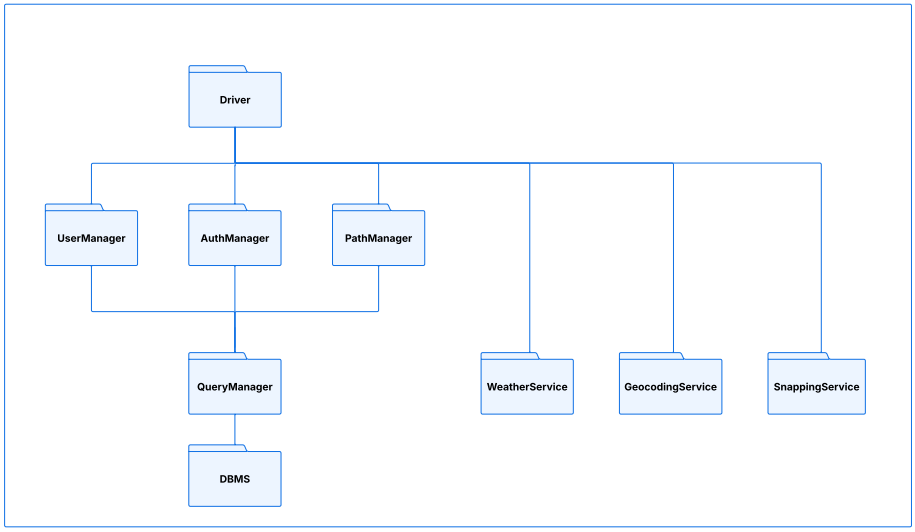

Figure 5.4: Step 4: PathManager Integration

The `TripManager` is added next. This module manages user trips and relies on the `PathManager` and the weather external service to associate contextual 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.

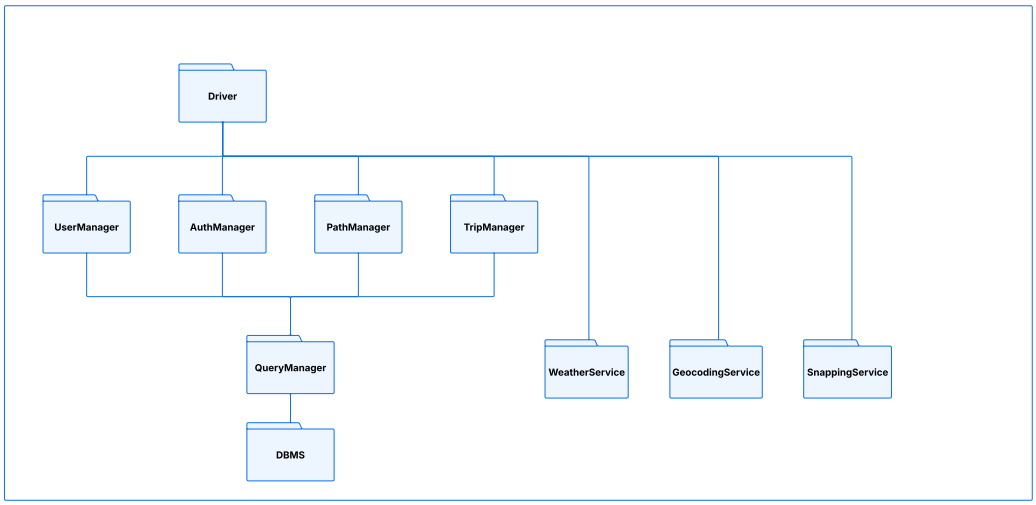

Figure 5.5: Step 5: TripManager Integration

After the `TripManager`, we integrate the `StatsManager`. This component aggregates data generated by the user's device to provide statistics. Its integration enables testing data flows based on trips and their associated routes. Integration tests will exercise interactions involving trips and paths to ensure that computed aggregates are consistent with the underlying data, preserve referential integrity, and remain efficient on realistic data volumes.

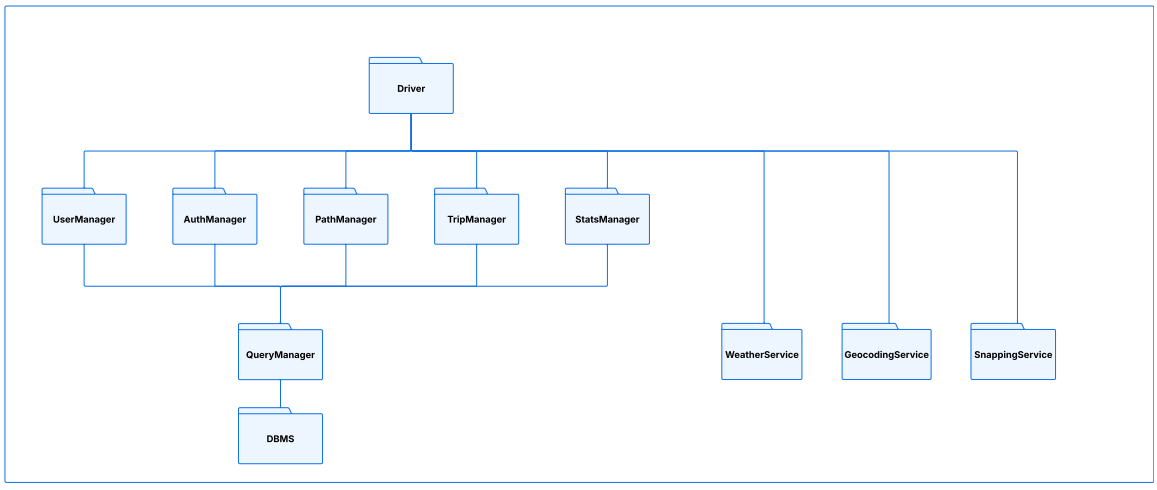

Figure 5.6: Step 6: StatsManager Integration

After the `StatsManager`, 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 are rejected, and

that report updates and confirmations are properly propagated to the `PathManager` so that route computation can take report information into account.

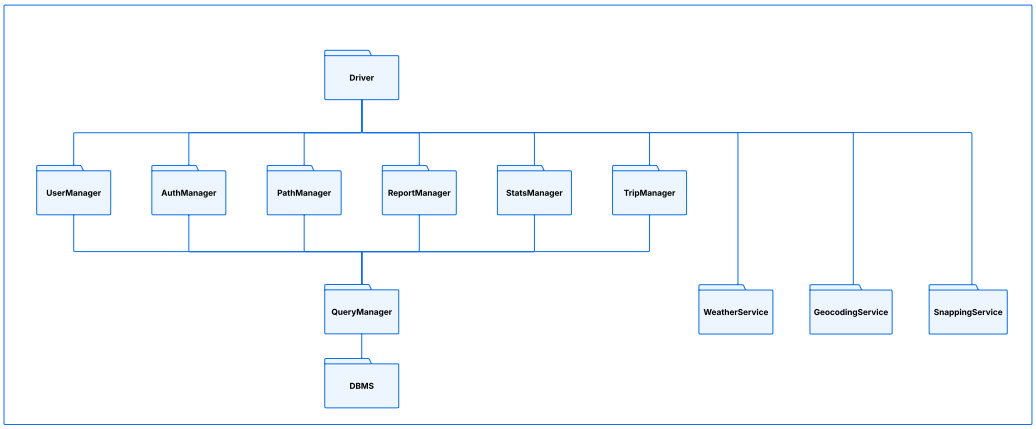

Figure 5.7: Step 7: ReportManager Integration

Finally, the mobile application (front-end) is integrated and used to perform real requests against the backend, replacing the driver. At this final stage, integration tests are executed end-to-end by exercising the main use cases from the mobile client, including real device sensors (e.g., GPS), and checking that interactions between the app, backend components, and the database behave as described in the Runtime View sequence diagrams.

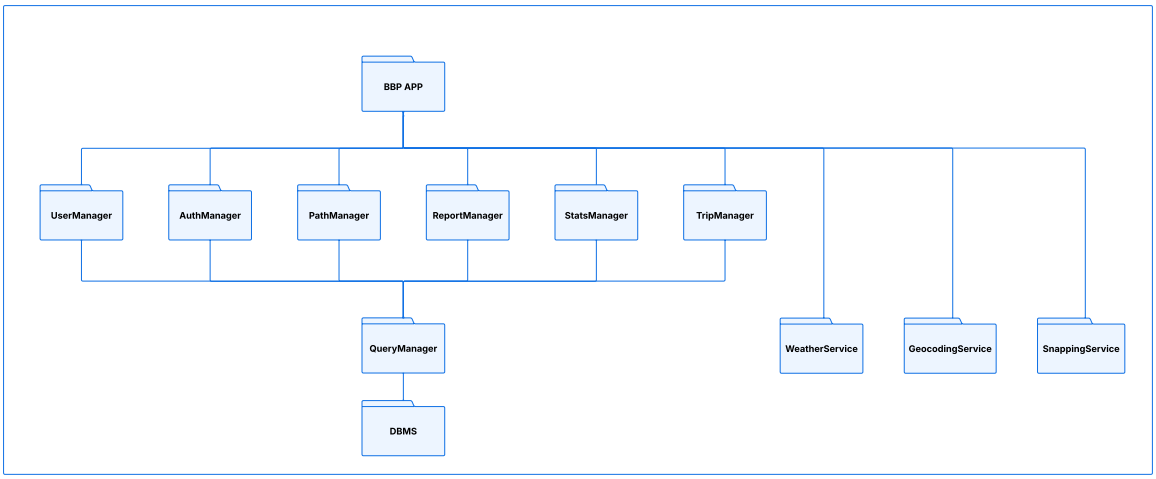

Figure 5.8: Step 8: System Integration via HTTP Test Driver

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 tests remain focused and fast. Particular attention will be devoted to error handling, input validation, and authorization checks at the module boundaries.

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. When applicable, contract checks will validate the structure of exchanged payloads (e.g., expected JSON schemas and HTTP status codes at the API Gateway).

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 References

6.1. Reference Documents

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

- IEEE Standard for Software Requirement Specifications [2];
- 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, Academic Year 2025/2026 [6];
- Slides of the Software Engineering II course available on WeBeep [7].

6.2. Software Used

The following software tools have been used to support the development of this project:

- **Visual Studio Code**: editing of source code and documentation (LaTeX), with project-wide search and formatting support [5].
- **LaTeX**: typesetting system used to produce the final RASD document in a consistent format [3].
- **Git**: version control used to track changes and support collaborative development [8].
- **GitHub**: remote repository hosting and collaboration platform used for versioning, reviews, and issue tracking [1].
- **Lucidchart**: creation of UML diagrams (use case diagrams, state diagrams, domain class diagram) [4].

6.3. Use of AI Tools

AI tools were used during the project in the same way as other supporting software tools. Their role was not to autonomously generate content, but to assist in improving the presentation of the document, supporting the organisation of ideas and enhancing overall textual coherence.
Their use was mainly limited to the drafting phase, where they helped compare different ways of explaining scenarios, simplify long paragraphs, and check whether certain sentences could be misunderstood. In several cases, interacting with an AI assistant helped clarify the underlying concepts before writing the final version of the text.

6.3.1. Tools Used

The AI tools employed during the project were:

- Gemini
- ChatGPT

6.3.2. Typical Prompts

AI tools were queried using prompts such as:

- "Rephrase this design description to make the interaction flow clearer."
- "Does this explanation of the component interaction sound ambiguous?"
- "Help restructure this paragraph describing a UI flow to improve readability."
- "Format this design description or table using LaTeX"
- "Help debug formatting or build issues related to VS Code or LaTeX"

6.3.3. Input Provided

The input given to AI tools consisted mainly of:

- Early drafts of paragraphs.
- Short text fragments requiring clarity checks.
- Sections with repeated structure where consistent wording was needed.

6.3.4. Constraints Applied

When using AI tools, the following constraints were strictly enforced:

- Preserve the intended meaning of the original text.
- Avoid introducing new design decisions or assumptions.
- Maintain terminology aligned with the definitions provided in this document.

6.3.5. Outputs Obtained

The interaction with AI tools resulted in:

- Clearer or more concise formulations of existing statements.
- Identification of potentially ambiguous sentences.
- Terminology suggestions to improve internal coherence.
- LaTeX formatting assistance for tables and code snippets.

6.3.6. Refinement Process

All AI-generated outputs were subject to a manual refinement process that included:

- Critical review of all suggestions.
- Verification against the original intent to avoid unintended changes.
- Manual integration to ensure consistency with the overall writing style.
- Alignment checks with established terminology and definitions.

7 | 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	2 hours	1 hours	5 hours	8 hours
Architectural Design	14 hours	11 hours	6 hours	31 hours
User Interface Design	13 hours	3 hours	6 hours	22 hours
Requirements Traceability	1 hours	2 hours	10 hours	13 hours
Implementation, Integration & Test	2 hours	13 hours	3 hours	18 hours
Final Review & Editing	1 hours	1 hours	1 hours	3 hours
Total Hours	**33 hours**	**31 hours**	**31 hours**	**95 hours**

Table 7.1: Time spent on document preparation

Bibliography

[1] GitHub Inc. Github. Online platform, 2025. https://github.com/.

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

[3] LaTeX Project Team. Latex: A document preparation system. Document preparation system, 2025. https://www.latex-project.org/.

[4] Lucid Software Inc. Lucidchart: Diagramming and visualization tool. Online platform, 2025. https://www.lucidchart.com/.

[5] Microsoft. Visual studio code. Source code editor, 2025. https://code.visualstudio.com/.

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

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

[8] Software Freedom Conservancy. Git. Version control system, 2025. https://git-scm.com/.

List of Figures

List of Tables