POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Software Engineering II

Design Document

Project: Best Bike Paths

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

Contents

1 | Introduction

1.1. Purpose

1.2. Scope

1.3. Definitions, Acronyms, Abbreviations

1.3.1. Definitions

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

1.3.2. Acronyms

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

- **UI:** User Interface.

1.3.3. Abbreviations

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

1.4. Revision history

- Version 1.0 (23 December 2025);

1.5. Reference Documents

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

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

1.6. Document Structure

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

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

2 | Architectural Design

2.1. 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 app layer, and a data layer. This architectural style ensures a clear division of responsibilities and simplifies the evolution of the system over time. Since BBP is conceived as a mobile-centric platform, the presentation tier is implemented entirely through the BBP mobile app, which communicates with the backend via RESTful APIs over HTTPS.

Presentation Layer

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

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

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

Application Layer

The 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.
- **PathManager**: responsible for maintaining bike-path data, computing routes, and ranking candidate paths according to their condition and effectiveness.
- **ReportManager**: responsible for storing and aggregating reports, managing confirmations, and updating path-condition indicators.
- **StatsManager**: computes and stores user statistics and per-trip metrics.
- **WeatherManager**: interfaces with an external weather service to retrieve meteorological data.

All modules are accessed through the **API Gateway**, which exposes a set of RESTful sub-APIs and routes incoming requests to the appropriate Manager.

Data Layer

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

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

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

2.2. Component View

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

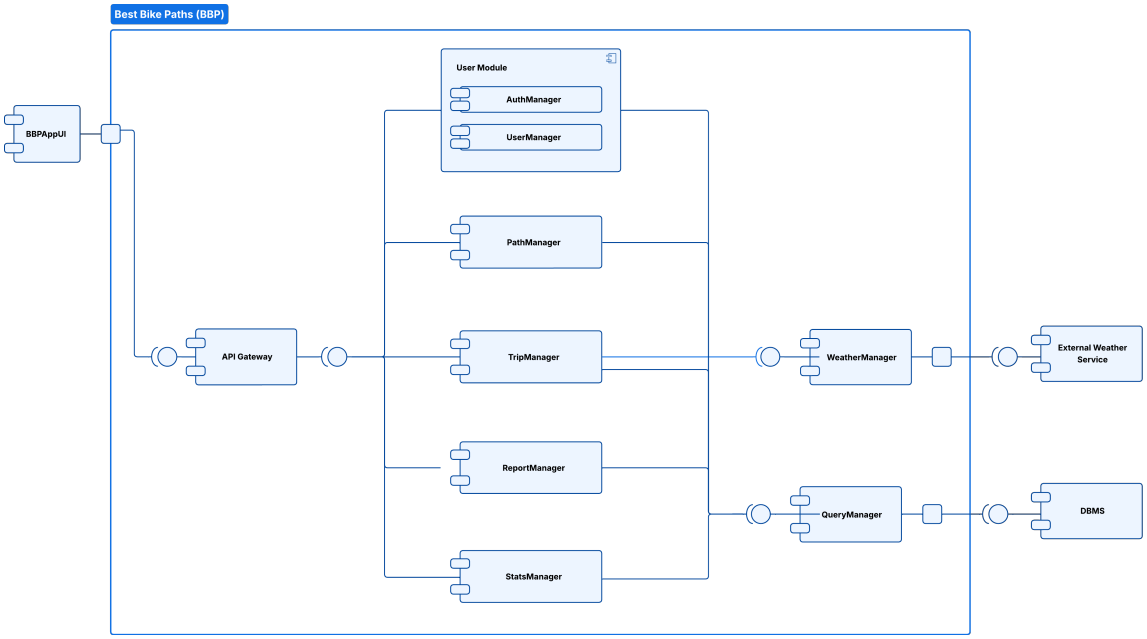

Figure 2.1: Component View Diagram

API Gateway

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

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

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

User Module

The **User Module** groups two Managers:

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

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

Trip Manager

The **TripManager** 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 (trajectory, metrics) to the backend. The TripManager generates the summary, enriches it with environmental data retrieved from the Weather Manager, and persists it through the **QueryManager**.

Path Manager

The **PathManager** is responsible for retrieving graph data from the database, computing optimal routes between two locations, and ranking alternative routes according to their quality and reported conditions. This service exposes the routing logic to the API Gateway and interacts with the **QueryManager** to retrieve pre-aggregated segment conditions (kept up to date by the ReportManager) alongside path data needed for route computation.

Reports Manager

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

Statistics Manager

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

Weather Manager

The **Weather Manager** interacts with the external weather API to obtain meteorological information. It provides a weather snapshot associated with trip start and end points. The snapshot is returned to the **TripManager**, which includes it in the final trip payload saved through the **QueryManager**.

QueryManager

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

2.3. Deployment View

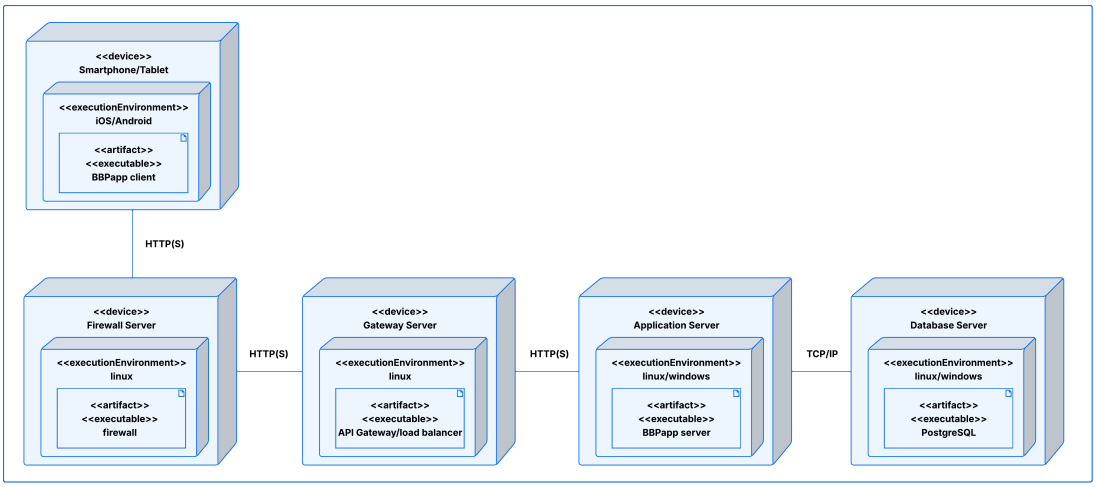

Figure 2.2: Deployment View of the BBP System

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

- **Presentation Tier:** this tier includes all devices through which end users interact with the BBP system. The primary clients are smartphones or tablets running iOS or Android, where the BBP mobile application is installed. These devices communicate with the backend exclusively via HTTPS, ensuring confidentiality and data integrity. Although the mobile app represents the main access point, any device equipped with a modern web browser and a stable internet connection could technically interact with the system, since the backend exposes standard RESTful endpoints. No application logic is executed at this level, the devices simply capture user input, display results, and forward authenticated HTTP requests toward the Application Tier.

- **Application Tier:** this tier is responsible for handling incoming traffic, enforcing security, applying routing and load balancing policies, and executing the core business logic of the system. All external requests first pass through a Linux-based server configured as a Web Application Firewall using *ModSecurity*. ModSecurity blocks malicious traffic such as SQL injection attempts, cross-site scripting payloads, and abnormal request patterns, while supporting anomaly detection. Validated HTTPS traffic is then forwarded to the gateway node running *Traefik*, which acts as the system's reverse proxy and load balancer. Traefik terminates HTTPS connections, exposes a single public entrypoint for clients, and distributes incoming requests across multiple backend replicas using load balancing strategies. Additional middleware functionalities (request logging or rate limiting) can be applied as needed at this level. The backend application itself is executed on one or more stateless Application Servers, each running the BBP RESTful backend. Since authentication tokens are included in each request header, no server-side session state is maintained, enabling horizontal scaling and dynamic replica management. Communication between Traefik and the backend replicas is confined to a protected internal network, reducing the system's exposure to external threats.

- **Data Tier:** the Data Tier consists of a dedicated server running a PostgreSQL database instance, which stores all persistent system data such as user information, paths, trips, and analytics. Backend servers interact with the database using standard PostgreSQL drivers over TCP/IP within an isolated internal network segment. Centralizing the database simplifies backup strategies, consistency enforcement, and maintenance operations, while still allowing for potential future extensions such as replication or clustering without altering the upper tiers of the system.

2.4. 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 Gateway**, which routes each request to the appropriate **Manager**. Persistence operations are centralised in the **QueryManager**, whereas weather-related data requests involve the **WeatherManager** and its external API.
The 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: email, password, and personal information. A first **local validation** step checks for malformed inputs before contacting the **backend**.
If the data is valid, the mobile app sends a registration request to the **API Gateway**, which forwards it to the **AuthManager**. This component checks that the email is not already in use by querying the database through the **QueryManager**. If the email already exists, the system returns an error and the mobile app notifies the user accordingly. If the email does not exist, the **AuthManager** inserts the new user into the database and generates an **authentication token**. The token is returned to the mobile app, which stores it securely using the device's **secure storage** mechanism. The flow concludes with a success message being shown to the user.

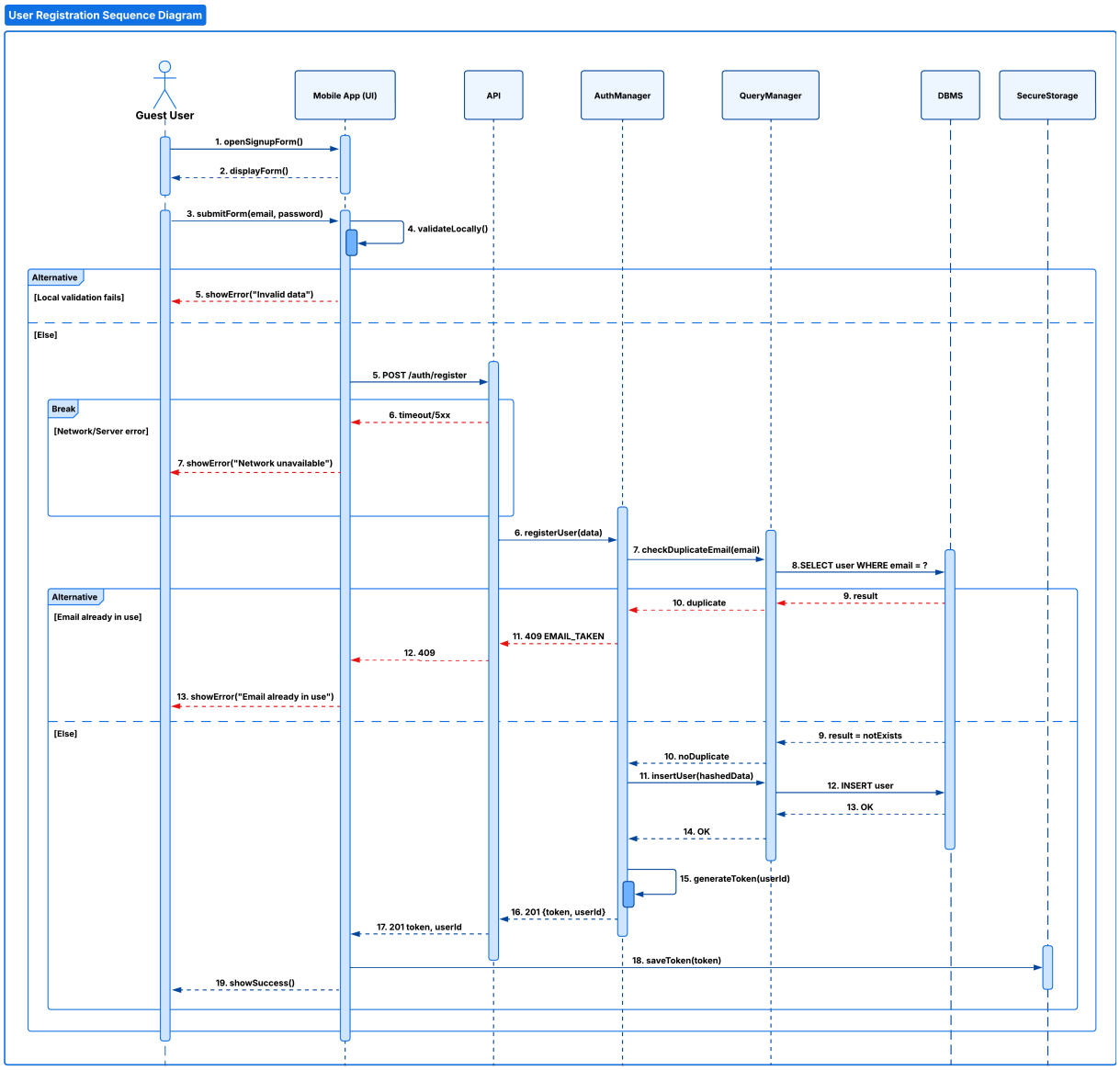

Figure 2.3: User Registration Sequence Diagram

[UC2] - User Log In

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

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

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

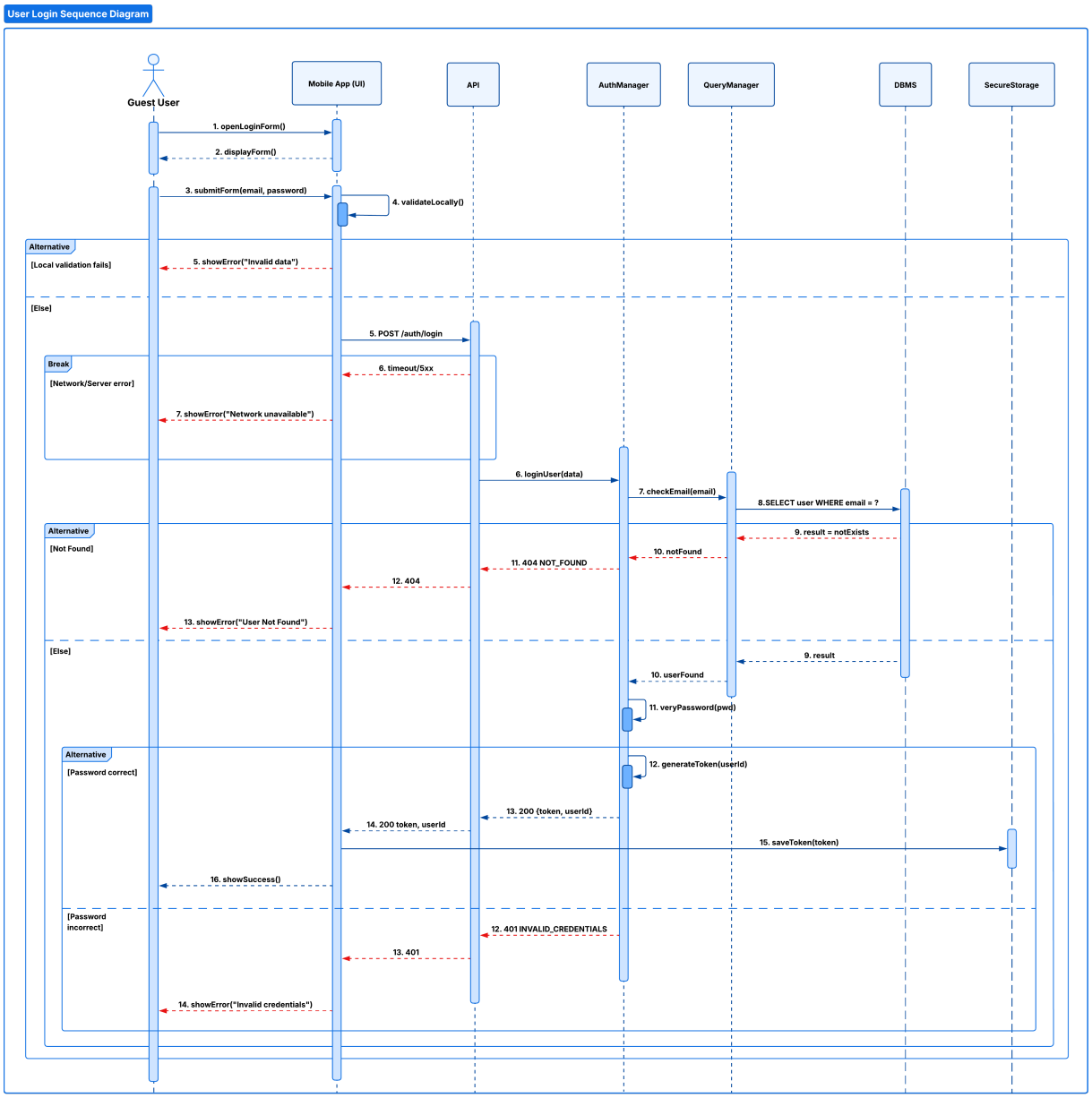

Figure 2.4: User Log In Sequence Diagram

[UC3] - User Log Out

When a logged-in user initiates the logout operation from the mobile app, the client first clears the locally stored authentication token from the **secure storage**. Afterward, the mobile app sends an HTTP request to the backend.
The **API Gateway** forwards the request to the **AuthManager**, which handles the logout process by deleting the corresponding **refresh token** through the **QueryManager**. The QueryManager executes a **DELETE** operation on the database to invalidate the stored refresh token.
If the operation succeeds, the server returns a **204 NO_CONTENT** response, and the app displays a success message to the user. If instead a network or server error occurs, the system interrupts the flow and the mobile app notifies the user with a "Network unavailable" error message.

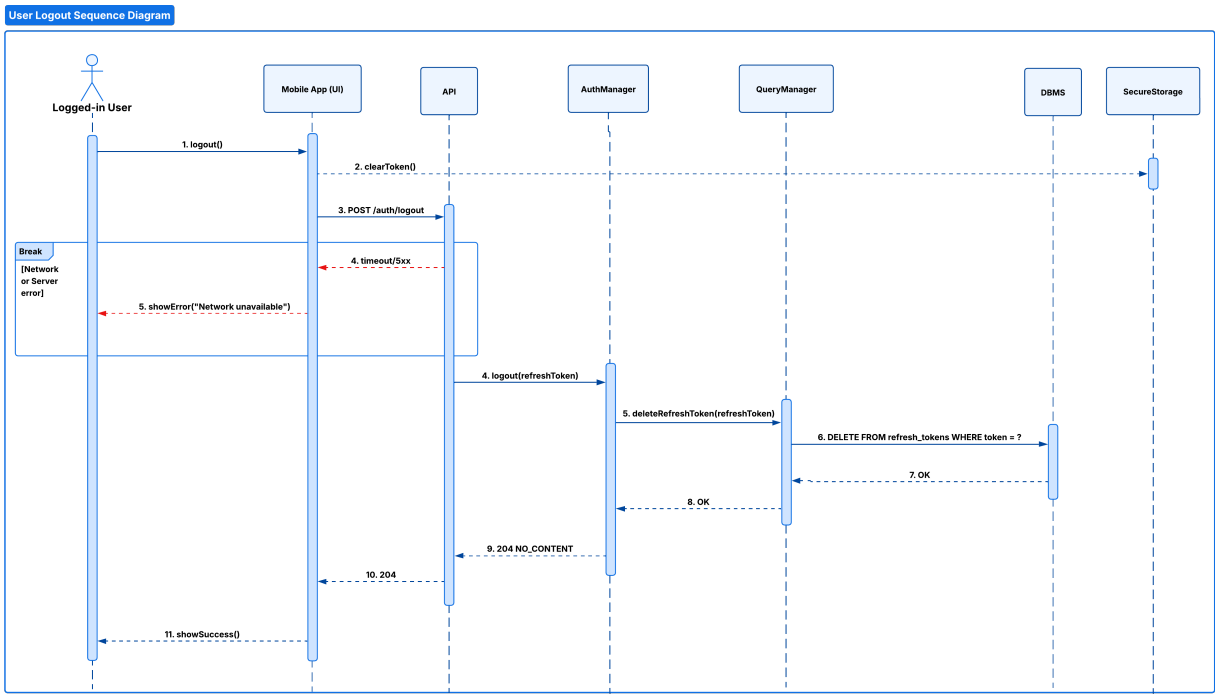

Figure 2.5: User Log Out Sequence Diagram

[UC4] - Search for a Path

A user can search for bike paths between two locations. He enters the start and end points and submits the request. The app performs a preliminary local validation and, if the data is valid, forwards the request to the backend through the **API Gateway**.
The **API Gateway** forwards the request to the **PathManager**, which loads the relevant portion of the path graph, including segment condition indicators derived from reports, from the database using the **QueryManager**. Once the graph data is retrieved, the **PathManager** computes the optimal path(s) according to the requested constraints. If valid routes are found, the API Gateway returns them to the mobile app, which displays the corresponding suggestions.
If no route satisfies the user's constraints, the **PathManager** signals a **NO_ROUTE** condition, which results in a 404 error. In the case of network or server issues, the app notifies the user with an appropriate error message.

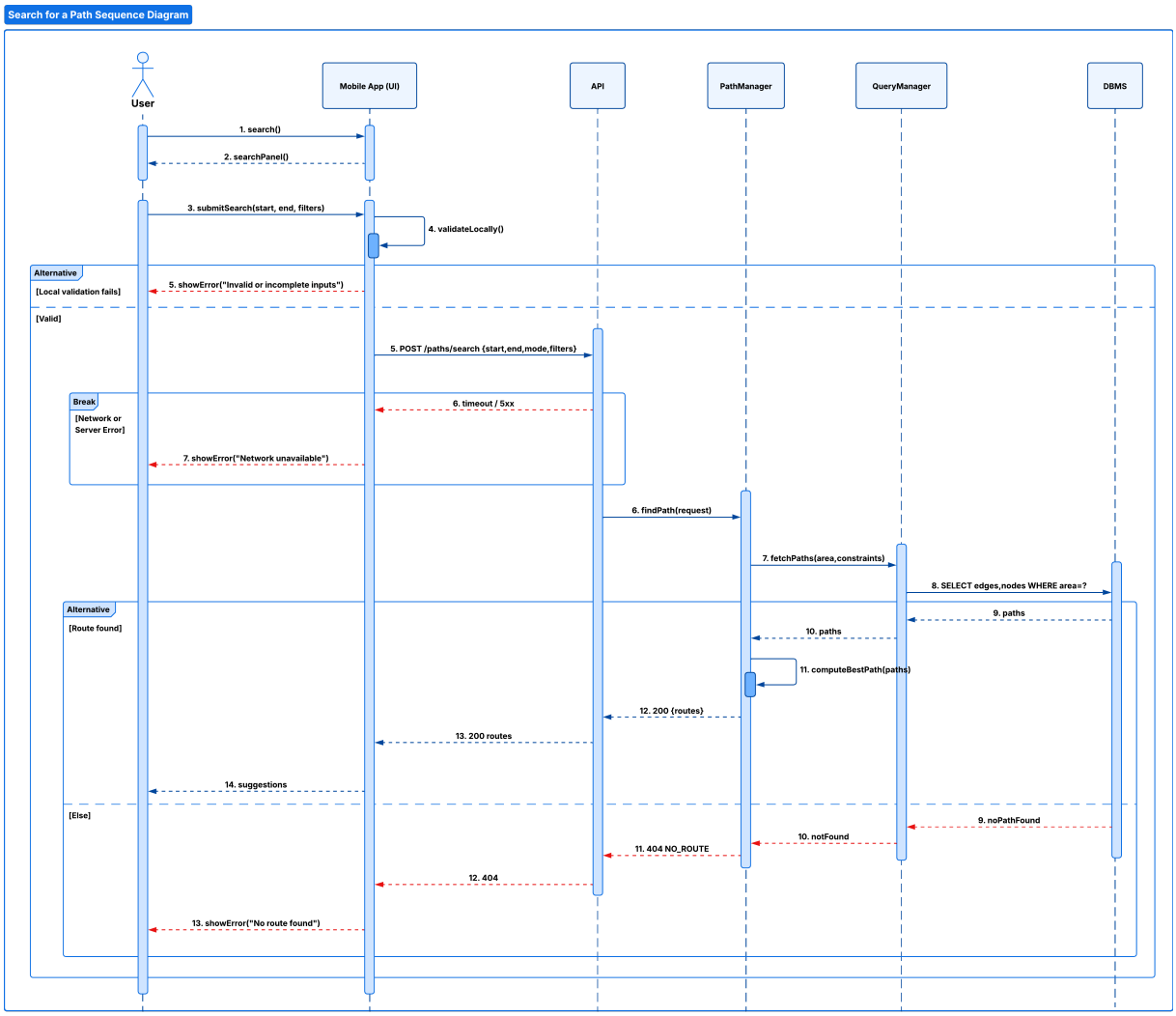

Figure 2.6: Search for a Path Sequence Diagram

[UC5] - Select a Path

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

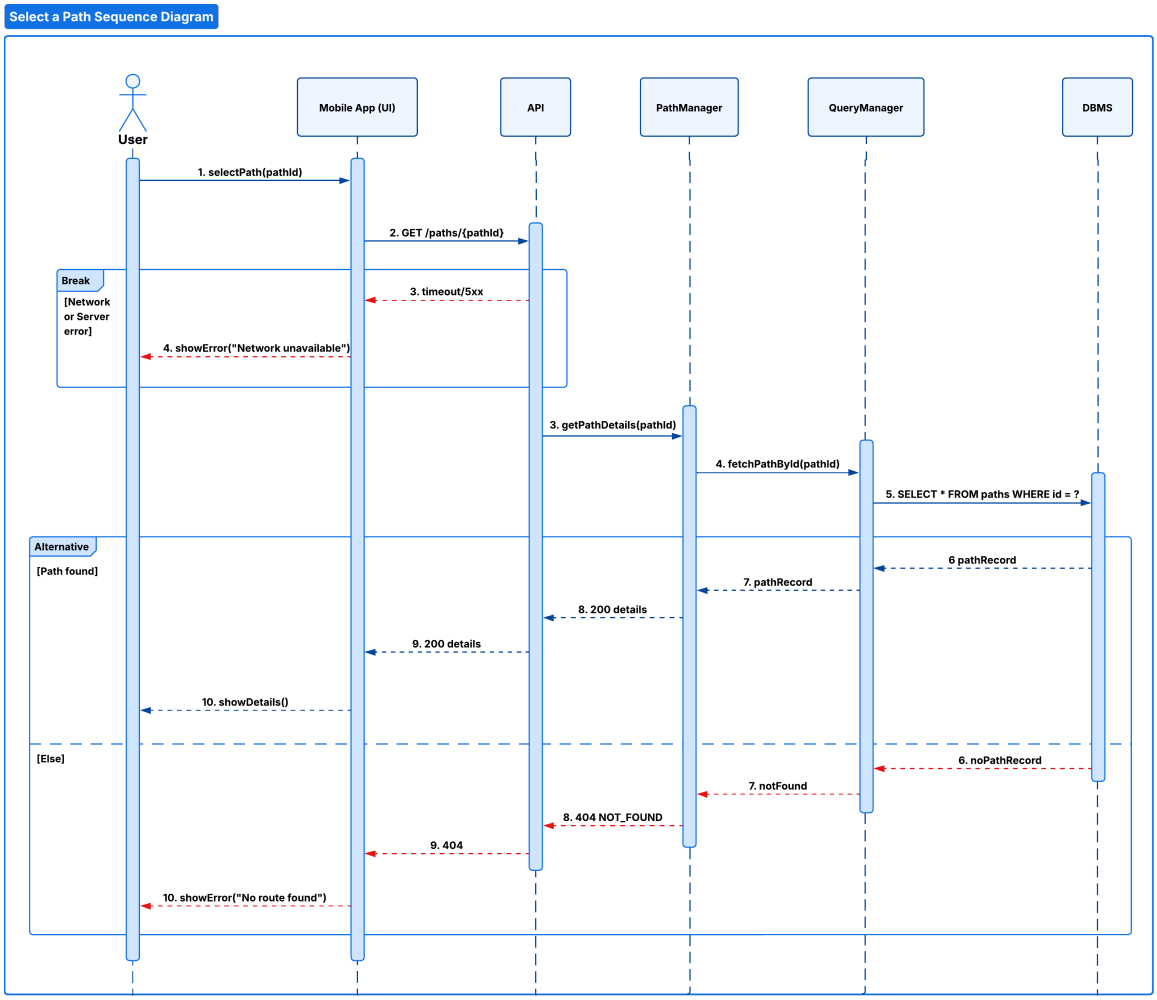

Figure 2.7: Select a Path Sequence Diagram

[UC6] - Create a Path in Manual mode

When a logged-in user wants to create a new path, he will be presented with a form to fill in the required metadata and choose between manual and automatic creation modes. If he opts for creating the path in **manual mode**, an interactive map is displayed on the mobile app, allowing the user to define the path by drawing it. The user will be able to save the path, sending the saving request to the backend.
Before sending the request, the app performs local validation to ensure that all mandatory fields and segments are correctly specified.
If the input is valid, the mobile app submits the creation request to the backend through the **API Gateway**. The gateway forwards the request to the **PathManager**, which is responsible for handling the creation workflow. The PathManager stores the new path by delegating the persistence task to the **QueryManager**, which executes the corresponding **INSERT** operation on the **DBMS**.
Once the database confirms the insertion, the **PathManager** sends the result back to the **API Gateway**. The gateway responds with a **201 Created** status and the new pathId. The mobile app then displays a confirmation message to the user. In the event of network failures or server-side errors, the app notifies the user accordingly.

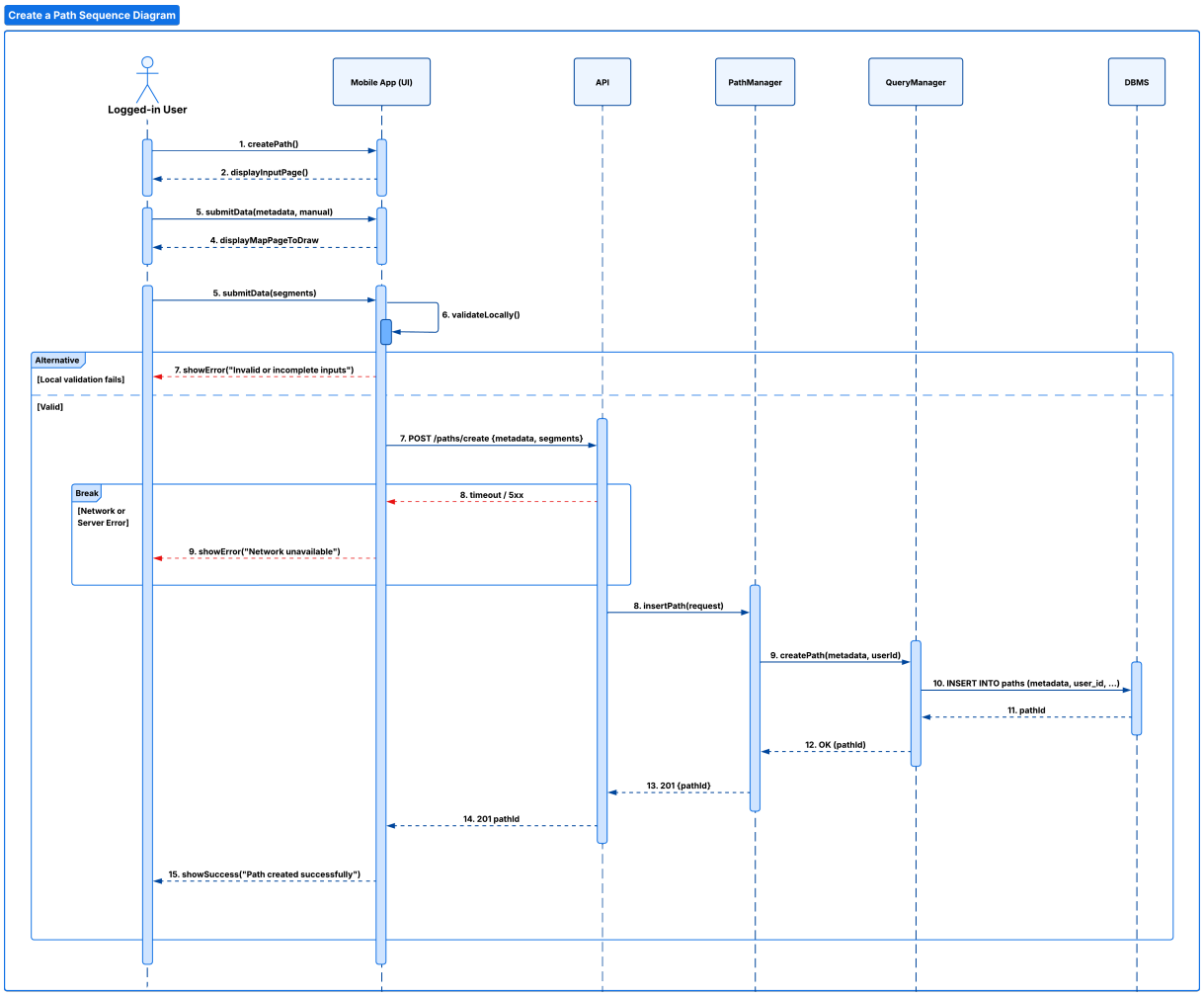

Figure 2.8: Create a Path in Manual Mode Sequence Diagram

[UC7] - Create a Path in Automatic Mode

When a logged-in user chooses to create a new bike path, he first selects the creation mode from the mobile app. The app displays a form for entering the required metadata, such as the path name, description, and other relevant details, and choosing the desired mode. If the user chooses automatic mode, the app proceeds to performs a first local validation. If the fields are invalid, the user is immediately notified.
Once the input is valid, the app attempts to activate **GPS tracking**. If activation fails (e.g., permissions or hardware issues), an error is shown. If tracking succeeds, the **GPS module** begins providing continuous location samples (latitude, longitude, speed, timestamp). The app collects these values during the entire movement loop.
If the GPS signal temporarily fails while moving, the app detects the error and interrupts the procedure, informing the user. When the user completes the movement session, GPS tracking is deactivated and the collected samples undergo a final **local validation**; if invalid, the user is notified. When both metadata and sensor samples are valid, the app sends a **POST request** to the backend. If a network timeout or server error occurs, an appropriate message is shown to the user.
The **API** forwards the request to the **PathManager**, which calls the **QueryManager** to store the path in the **DBMS**. The system inserts metadata and user association into the database, and upon successful insertion returns a **201 Created** response containing the new pathId. The mobile app receives the response and displays a success message, indicating that the new automatically generated path has been saved.

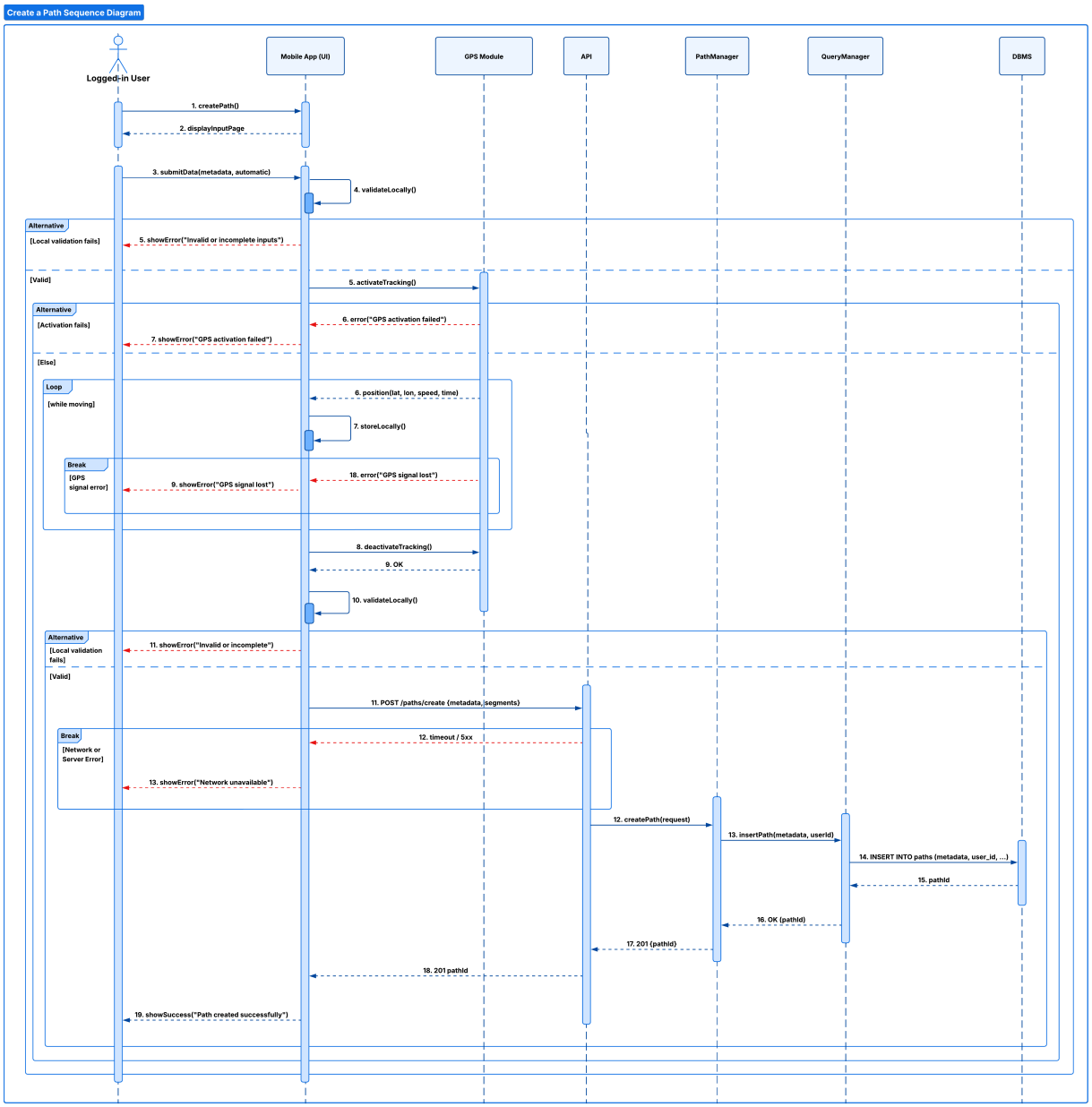

Figure 2.9: Create a Path in Automatic Mode Sequence Diagram

[UC8] - Delete a Path

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

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

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

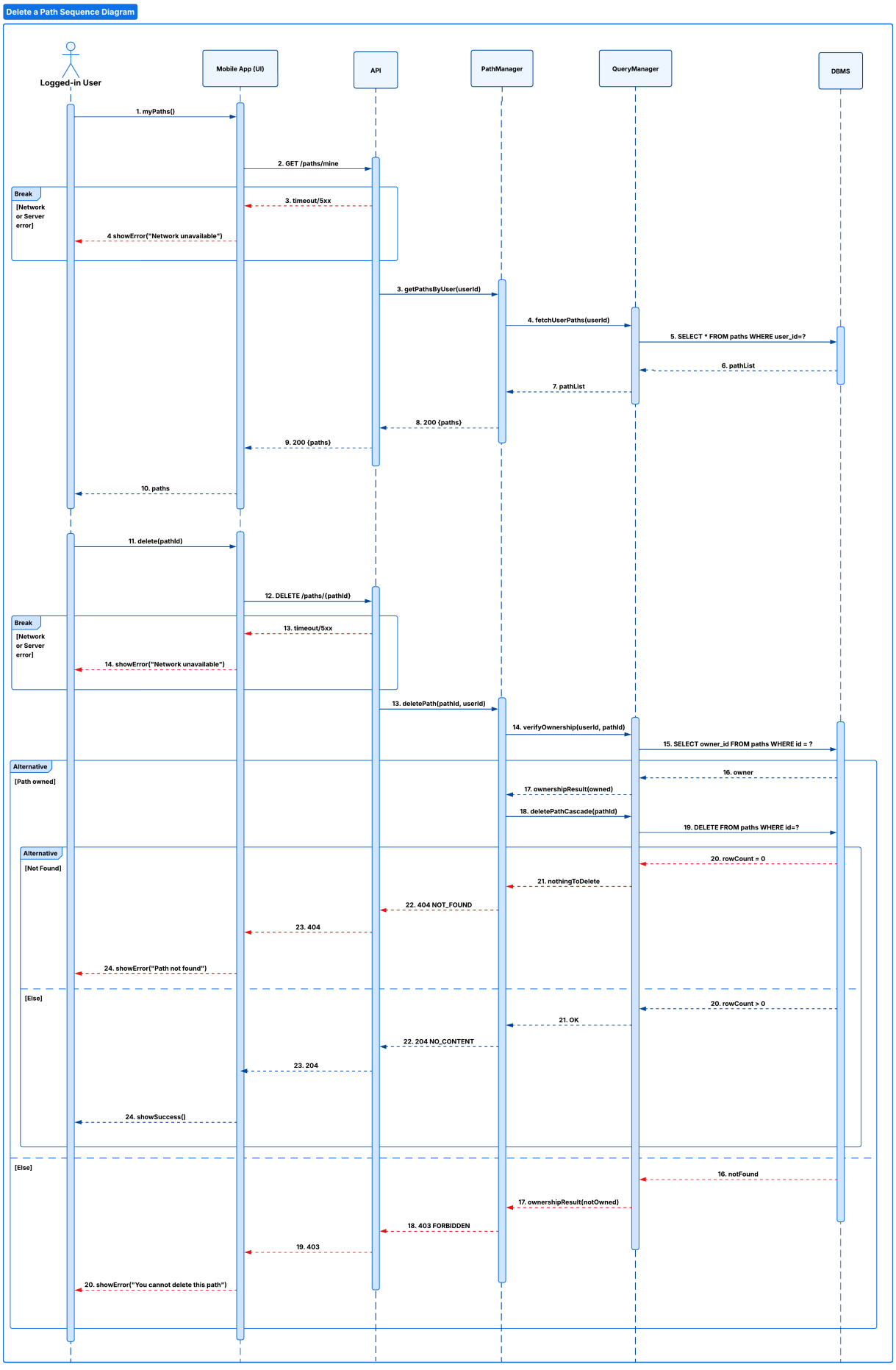

Figure 2.10: Delete a Path Sequence Diagram

[UC9] - Start a Trip as Guest User

When a guest user wants to start a trip using the BBP mobile app, he first selects a path from the available options. The app then attempts to activate **GPS tracking** to monitor the user's movement along the selected path.
If GPS activation fails, the mobile app immediately notifies the user with an error message. Otherwise, once tracking is active, the app continuously receives location updates while the user is moving and refreshes the map accordingly.
If at any point the **GPS module** reports a loss of signal, the app displays an appropriate error message to the user. No backend interaction occurs in this use case, as guest trips are not recorded or stored.

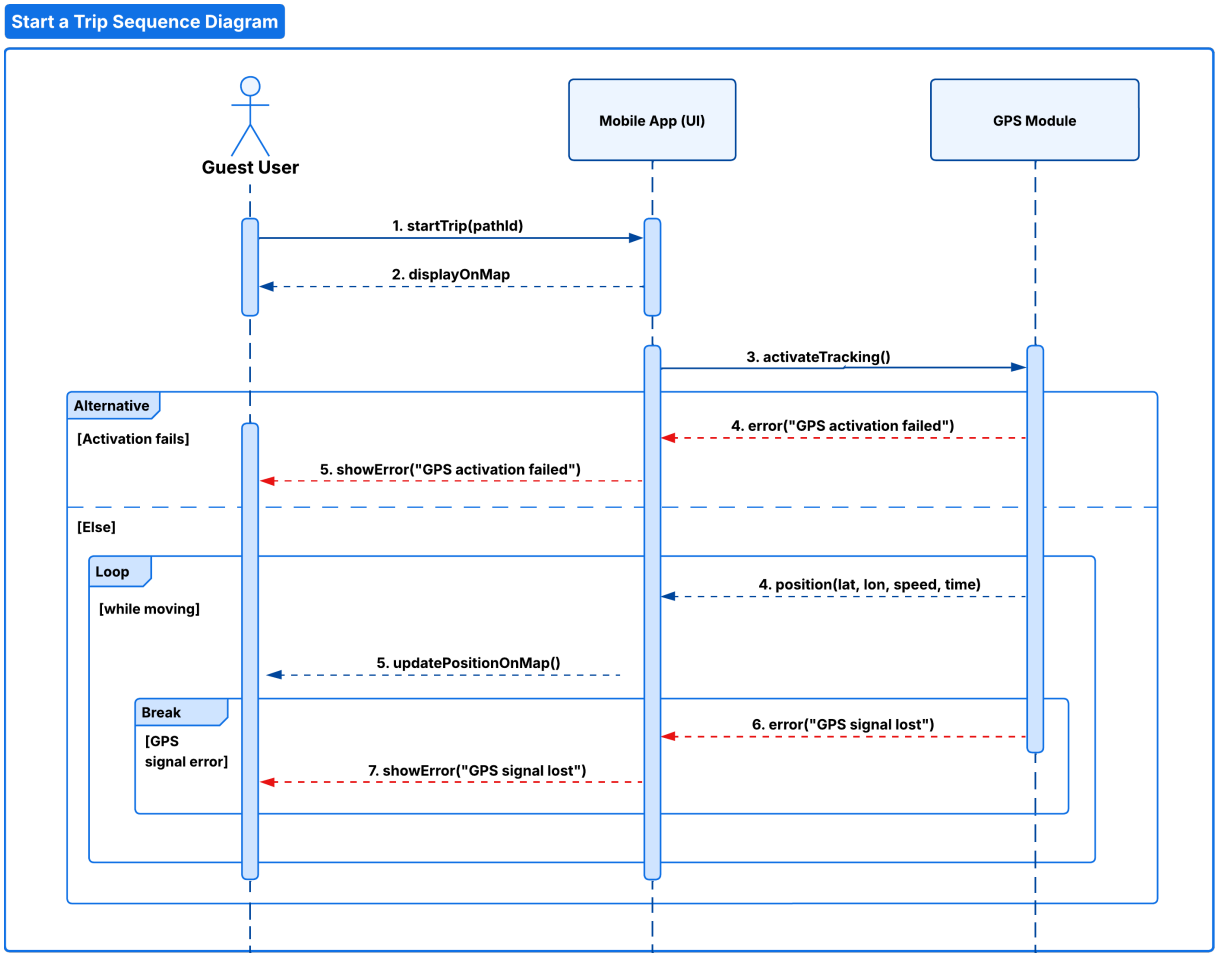

Figure 2.11: Start a Trip as Guest User Sequence Diagram

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

In this scenario, a logged-in user starts a trip by selecting a path. The interaction begins when the user initiates the trip from the mobile app, which displays the chosen path on the map. The user chooses not to enable automatic mode, and then the app activates the **GPS module**. If the GPS fails to activate, the mobile app immediately notifies the user with an error message. Otherwise, GPS tracking begins, and the device periodically emits position updates containing latitude, longitude, speed, and time. The mobile app stores these samples locally and updates the on-screen map in real time. The trip identifier is generated and maintained locally by the app during tracking and will be sent to the backend only when the trip is finalised. During the trip, if the GPS signal is lost at any point, the GPS module reports an error and the mobile app displays a corresponding warning to the user, interrupting the normal flow of position updates.

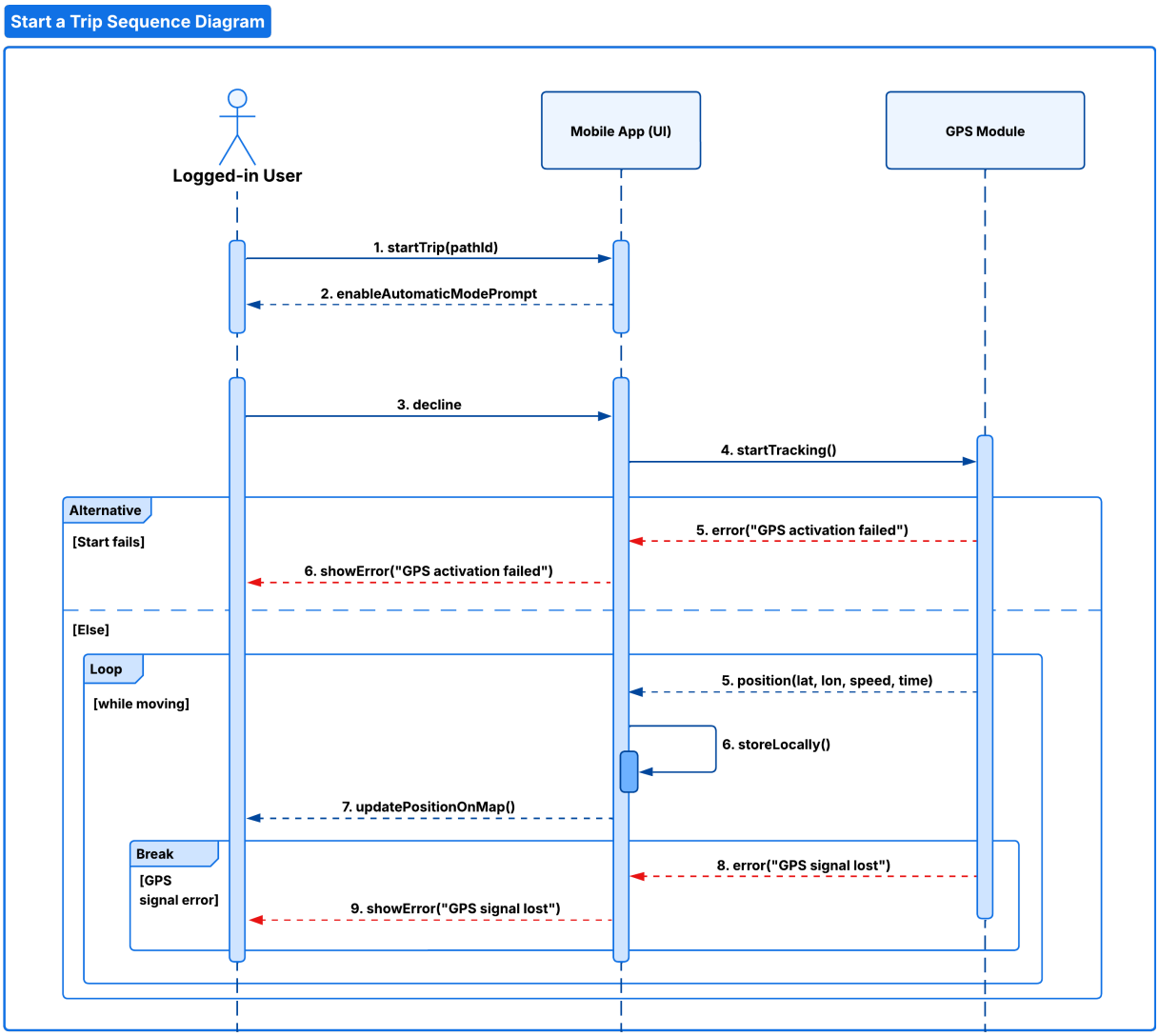

Figure 2.12: Start a Trip in Manual Mode as a Logged-in User Sequence Diagram

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

When a logged-in user selects a path and chooses to enable **automatic mode**, the mobile app first displays the map and then attempts to establish a connection with the **external sensors** required for automatic detection.
If sensor activation fails, the app immediately informs the user with an error message. Otherwise, sensors respond successfully and the app proceeds by enabling GPS tracking. Once tracking is active, the **GPS Module** periodically sends position updates which the app stores locally and reflects on the UI map. The trip identifier remains local to the app during tracking and is forwarded to the backend only when the trip is stopped and finalised.
During the trip, if the GPS signal is lost, the system interrupts the loop and displays an appropriate error message. If instead the connection to the external sensors drops during the session, the app stops the automatic process and notifies the user of the failure.

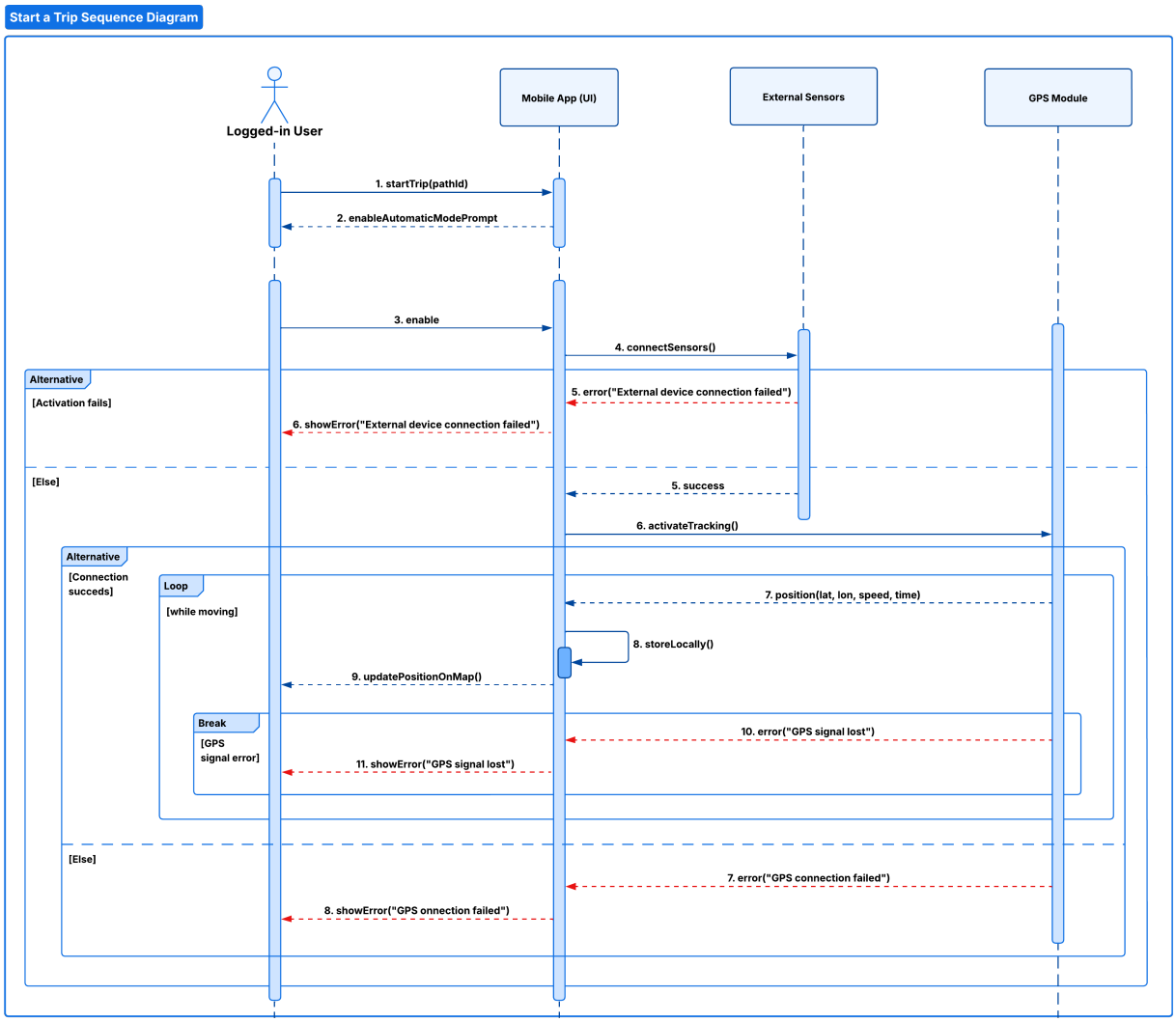

Figure 2.13: Start a Trip in Automatic Mode as a Logged-in User Sequence Diagram

[UC12] - Stop a Trip as Guest User

This sequence diagram describes how a guest user stops an ongoing trip. The interaction is entirely local, as guest users do not store trip data on the backend.
The process begins when the user selects the **Stop Trip** action. The mobile app then requests the **GPS module** to deactivate tracking. Once the GPS confirms that tracking has been successfully stopped, the app terminates the trip visualisation and returns the user to the map or home screen.
No network communication is involved, and no data is persisted, making this use case lightweight and fully handled on the device.

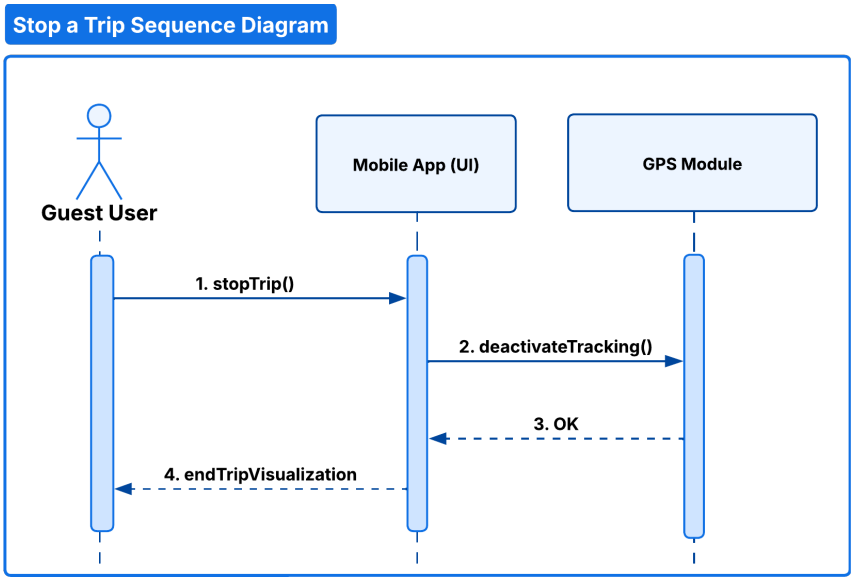

Figure 2.14: Stop a Trip as a Guest User Sequence Diagram

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

When the user wants to stop an ongoing trip, he selects the Stop Trip action from the mobile app. Upon this action, the mobile app terminates any active data acquisition (GPS tracking and, if the selected mode is Automatic, external sensor streams) and sends a stop-trip request to the backend, including the locally maintained trip identifier. The **TripManager** validates the request and retrieves the corresponding trip record through the **QueryManager**. If the request is invalid, for example, if the trip does not exist or is already closed, the system returns an appropriate error.
If validation succeeds, the **TripManager** contacts the **WeatherManager** to obtain a contextual weather snapshot. It then composes the final trip summary using distance, duration, speed metrics, sensor-derived data (when available), and the weather snapshot. The summary is then saved through the **QueryManager**.
The backend responds with a **201 Created** status and the complete summary. The mobile app then displays the result to the user. Network failures, invalid identifiers, or missing trip records trigger the corresponding alternative flows.

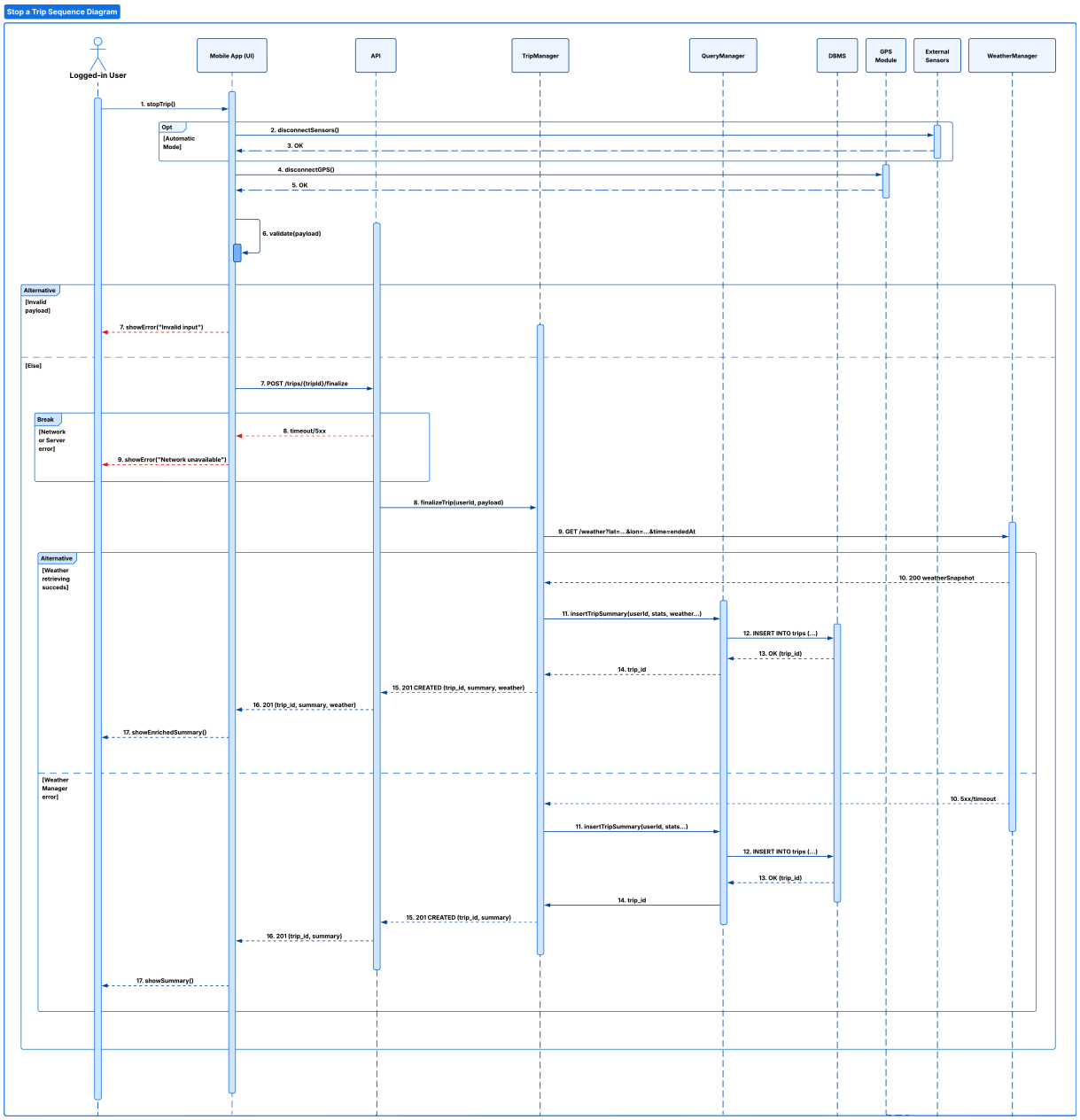

Figure 2.15: Stop a Trip as a Logged-in User Sequence Diagram

[UC14] - Make a Report in Manual Mode

The logged-in user selects a path segment on the map and opens the report-creation form. The mobile app retrieves the user's current GPS position. If the position cannot be retrieved, the app immediately shows an error.
After the user submits the form containing the report description and selected options, the mobile app performs local validation. Invalid inputs cause the app to show an error without contacting the backend.
If validation succeeds, the app sends the report payload to the backend through the **API Gateway**. Network or server-side failures lead to a timeout and an error message shown to the user.
Once the request is received, the **API Gateway** forwards the data to the **ReportManager**, which creates a report record by storing the user ID, position, and payload in the database through the **QueryManager**. If the insertion succeeds, the **DBMS** returns the generated report identifier.
Finally, the backend responds with **201 Created**, and the mobile app displays a confirmation message to the user.

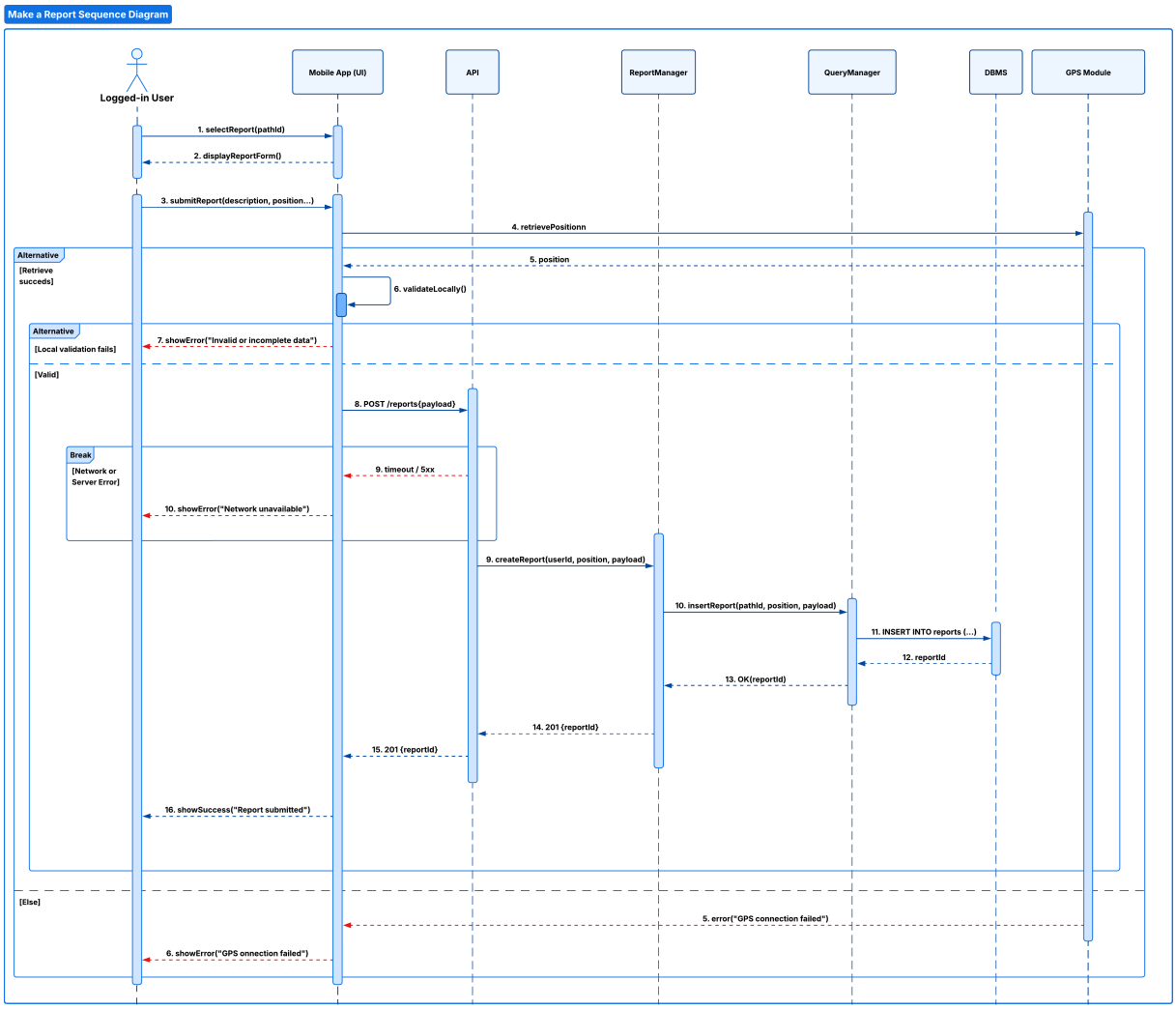

Figure 2.16: Make a Report in Manual Mode Sequence Diagram

[UC15] - Make a Report in Automatic Mode

When an obstacle is detected by the external sensors during a trip, the mobile app retrieves the current GPS position from the device. If the retrieval fails, the app displays an appropriate error message and the flow terminates.
Once the position is available, the app displays a pre-filled report form containing the detected issue. The user can review and modify the report details before submission. After the user confirms the report, the app performs local validation on the generated data. Invalid or incomplete payloads trigger a local error message and no request is sent to the backend.
If validation succeeds, the mobile app sends the report payload to the backend via the **API Gateway**. Network or server errors result in a timeout, causing the app to show a network-unavailable message.
Upon receiving a valid request, the **ReportManager** creates a new report record, storing it through the **QueryManager**, which inserts the new record into the database.
After successful insertion, the backend returns a **201 Created** response. The mobile app then informs the user that the report has been submitted successfully.

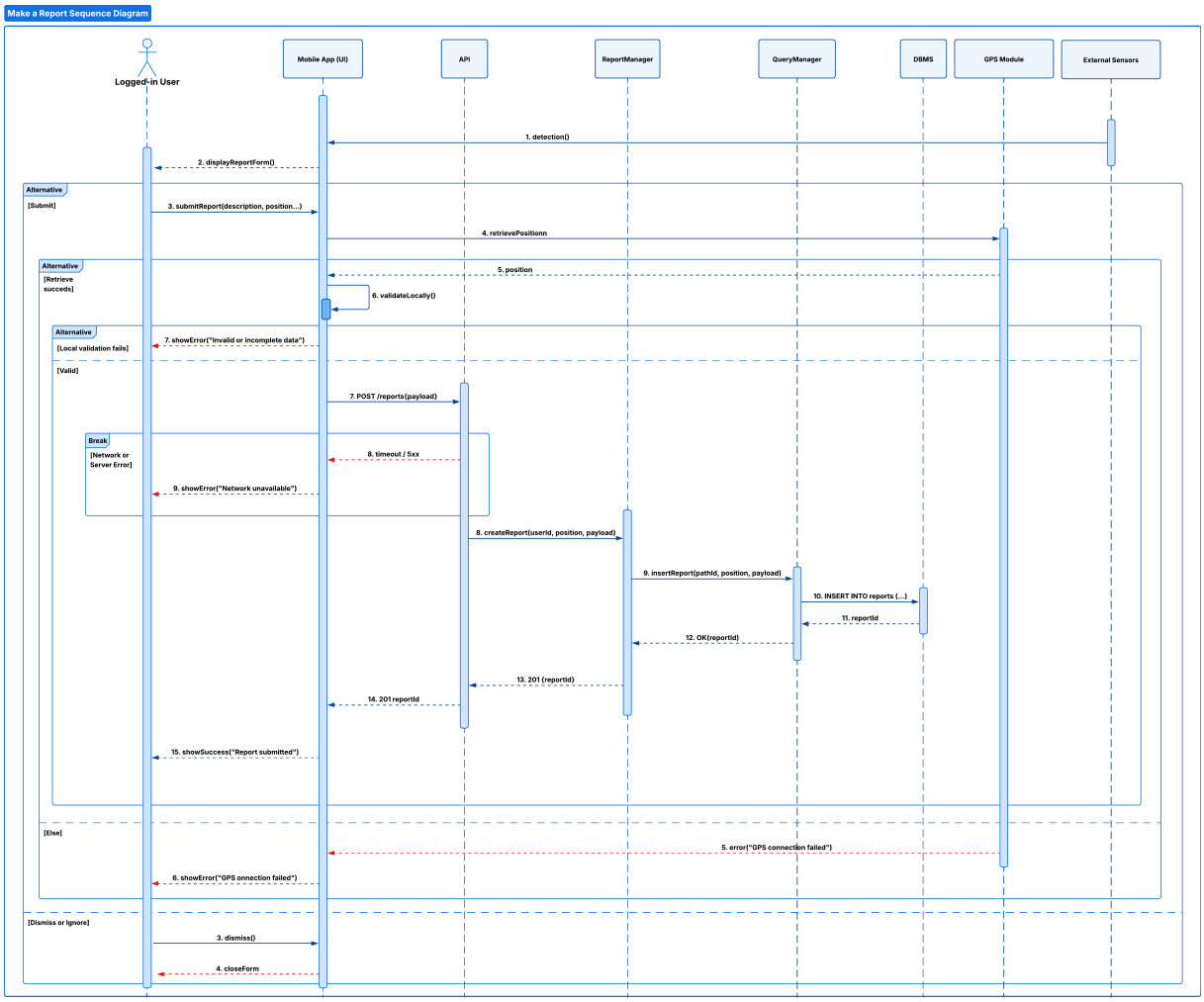

Figure 2.17: Make a Report in Automatic Mode Sequence Diagram

[UC16] - Confirm a Report

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

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

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

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

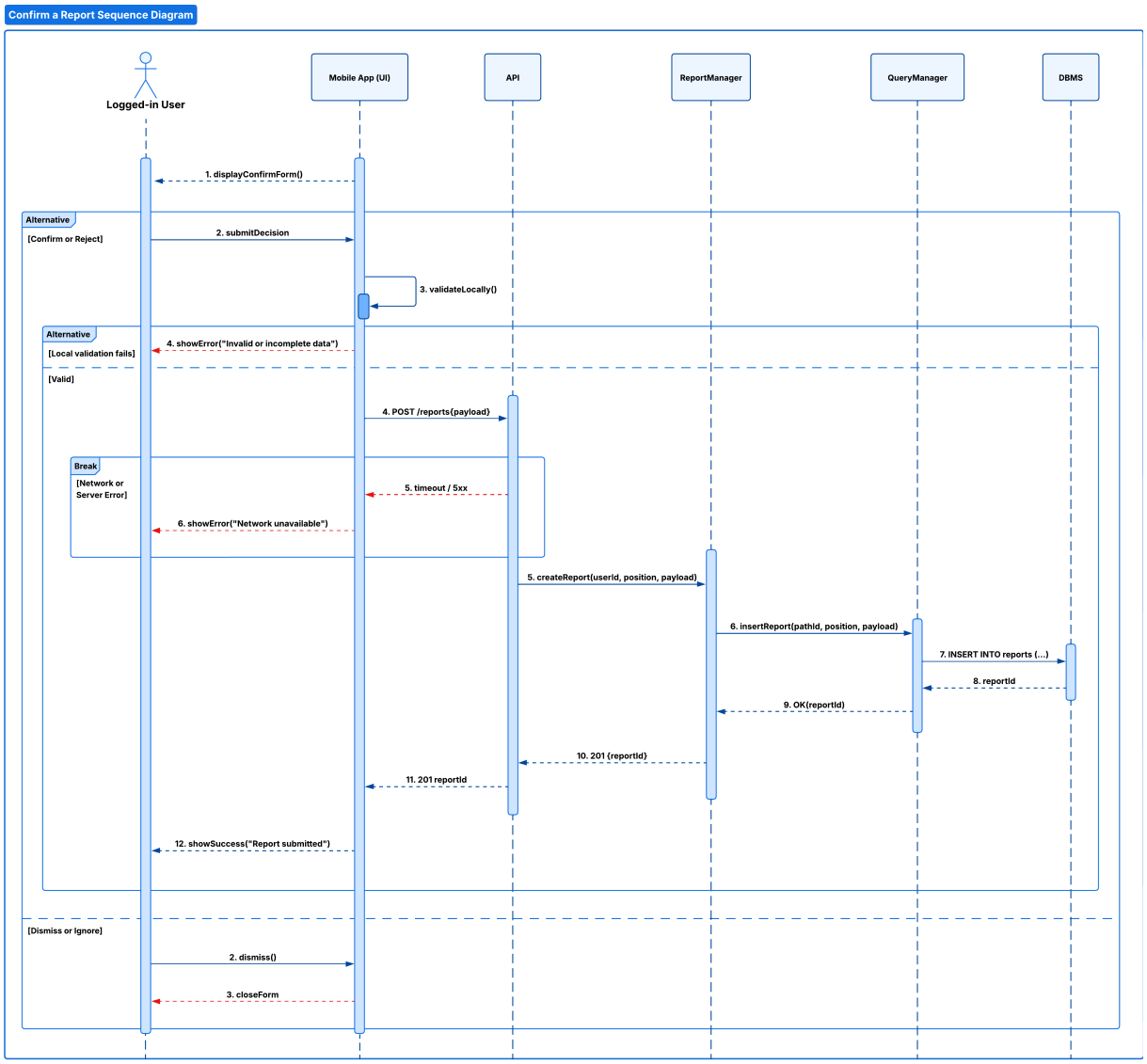

Figure 2.18: Confirm a Report Sequence Diagram

[UC17] - Manage Path Visibility

The user selects the desired path from the list of previously created paths. Then, the mobile application sends a request to the backend to retrieve the current visibility settings of the selected path. The **PathManager**, through the **QueryManager**, loads the corresponding path record from the database.
If the path is successfully found, the app displays the existing visibility configuration and waits for the user to submit the desired changes. Once the user confirms the update, the mobile application sends a request containing the updated visibility value. Upon receiving it, the **PathManager** verifies that the requesting user is indeed the owner of the path. If the ownership constraint is satisfied, the visibility attribute is updated in the database. A successful update triggers a confirmation response, and the mobile application communicates the result to the user.
If the initial fetch request fails due to network or server issues, an error message is displayed. If the selected path does not exist or no record is returned from the database, the application notifies the user accordingly. If the user attempts to modify a path they do not own, the backend returns a **403 FORBIDDEN** response, and the app signals that the operation is not permitted.

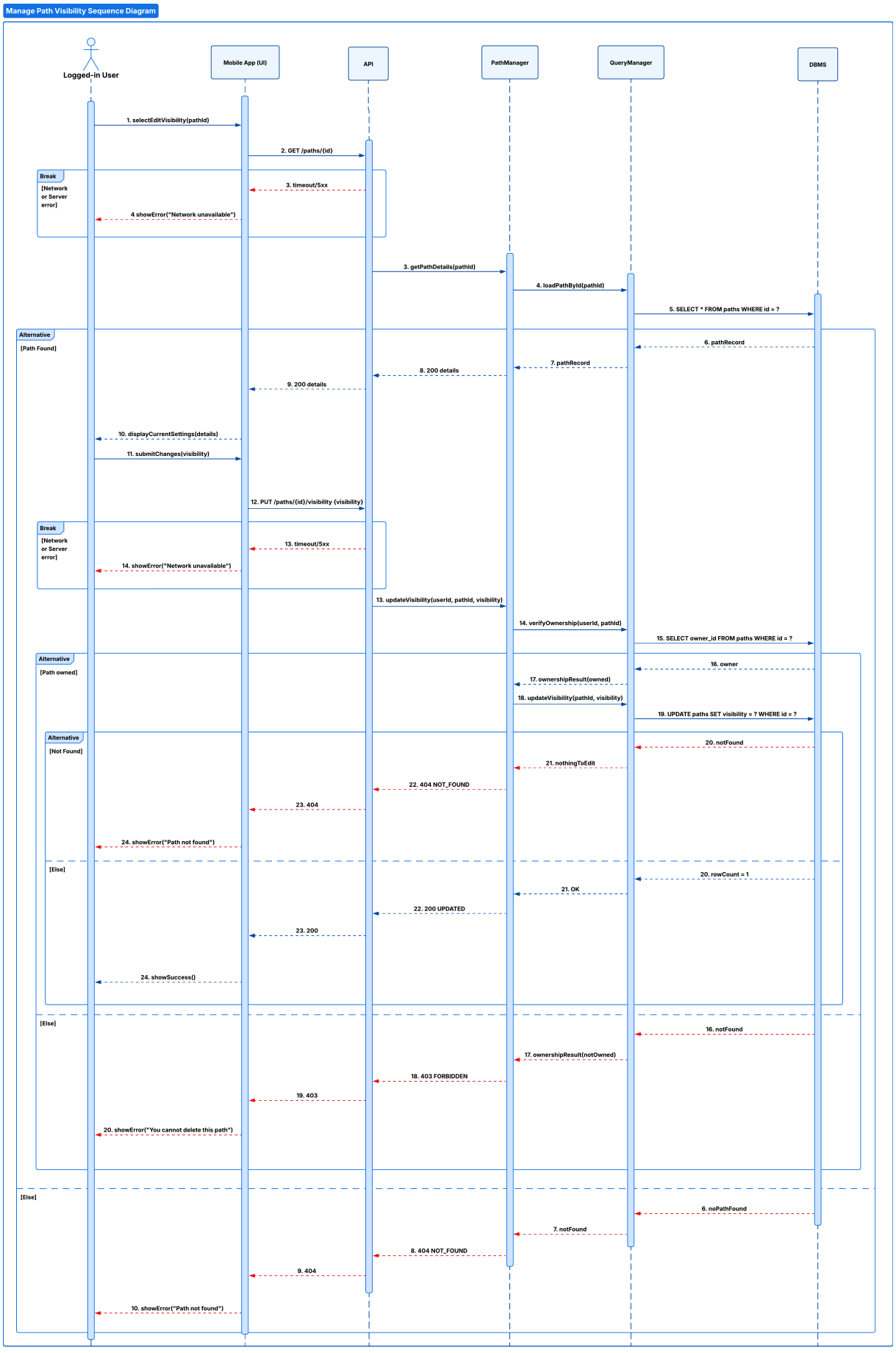

Figure 2.19: Manage Path Visibility Sequence Diagram

[UC18] - View Trip History and Trip Details

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

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

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

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

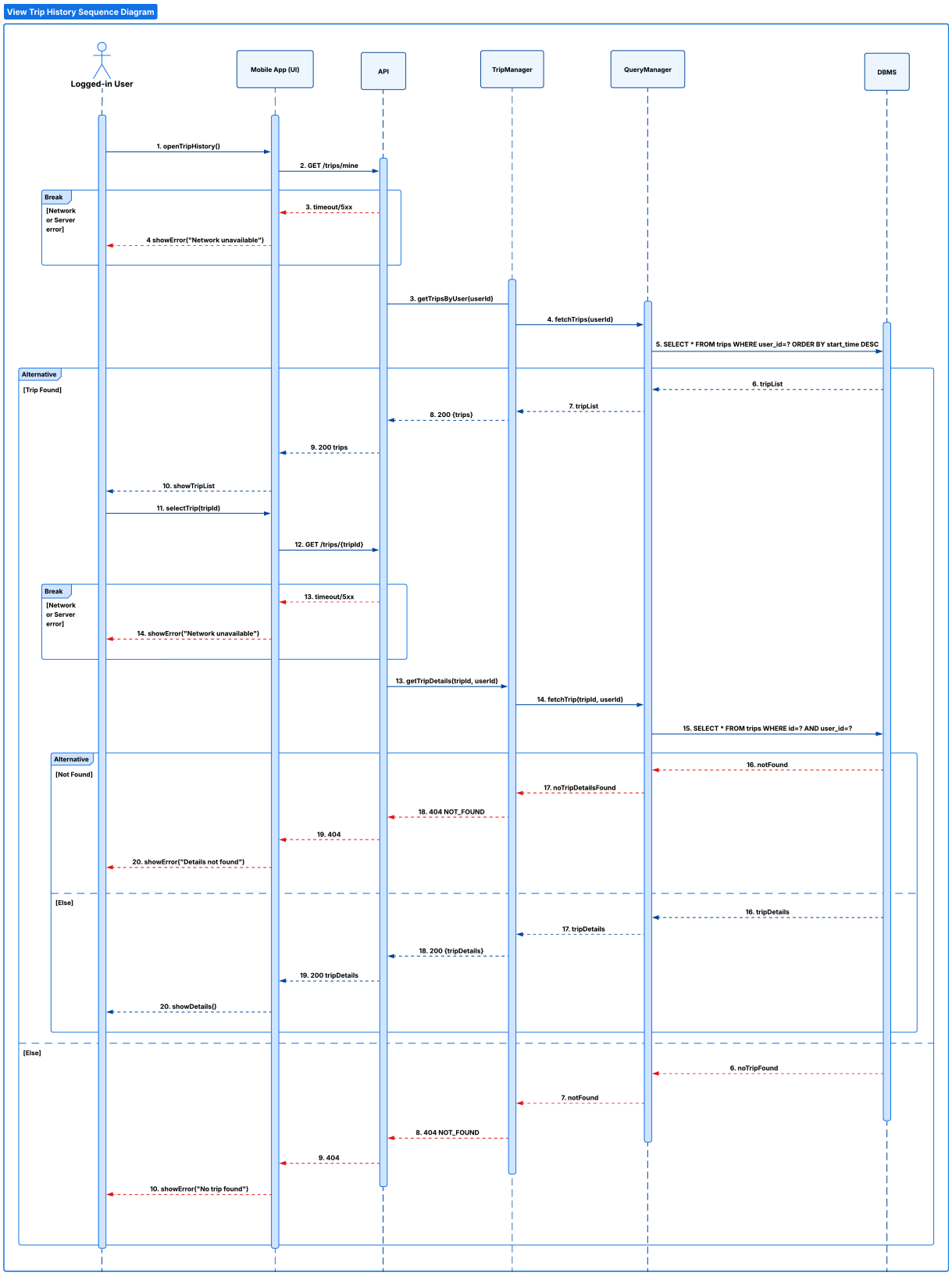

Figure 2.20: View Trip History and Trip Details Sequence Diagram

[UC19] - View Overall Statistics

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

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

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

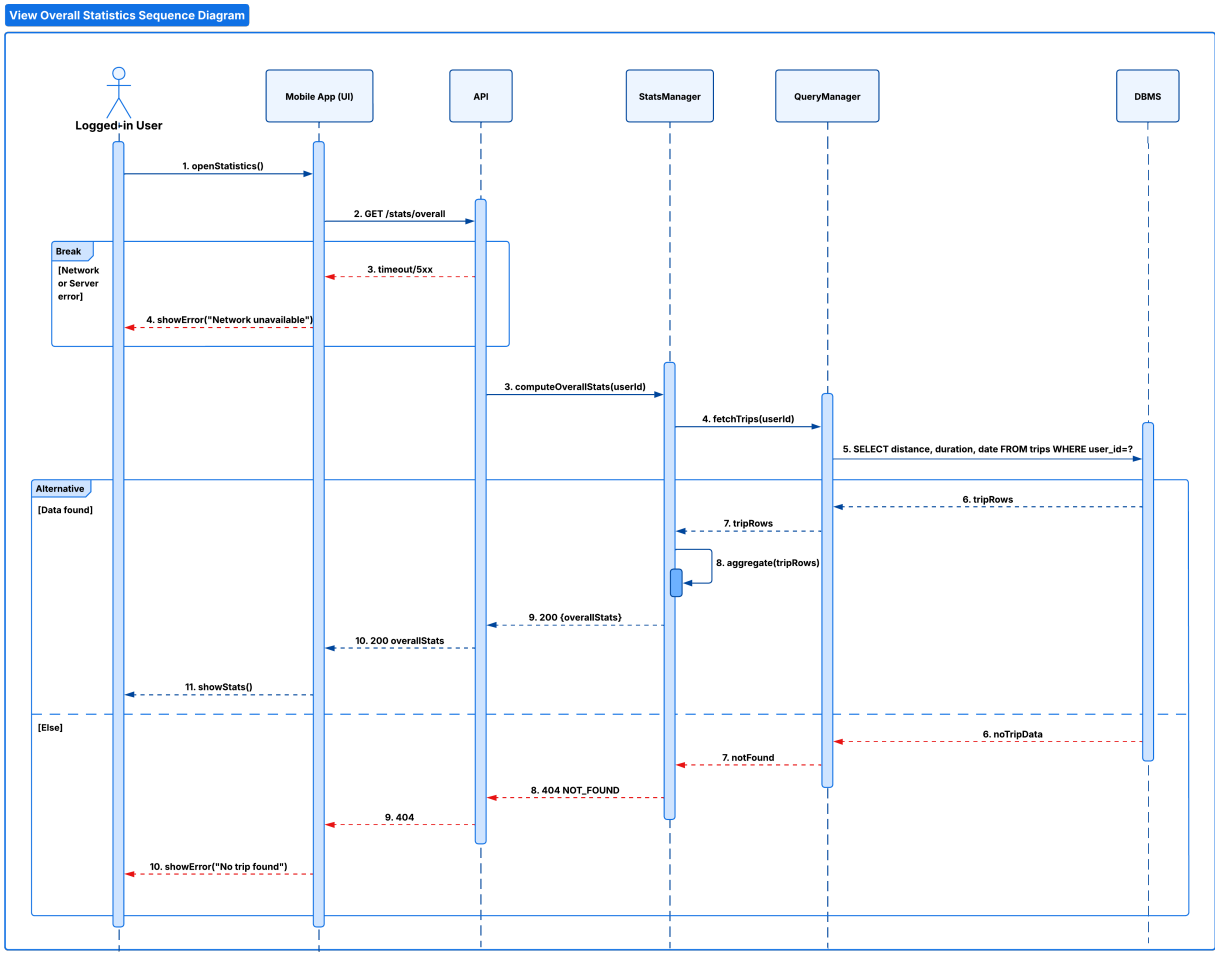

Figure 2.21: View Overall Statistics Sequence Diagram

[UC20] - View Trip Statistics

When the Logged-in user selects a trip, the mobile app sends a request for detailed metrics. Network or server failures are handled locally by showing an appropriate error.
If the request reaches the backend, the **API** delegates it to the **StatsManager**, which loads the associated trip data through the **QueryManager** by querying the **DBMS**. If the requested trip is found, the **StatsManager** computes the aggregated statistics and returns them to the mobile app, which presents them to the user.
If the trip does not exist or does not belong to the user, the backend replies with a **404 NOT_FOUND** response and the app notifies the user.

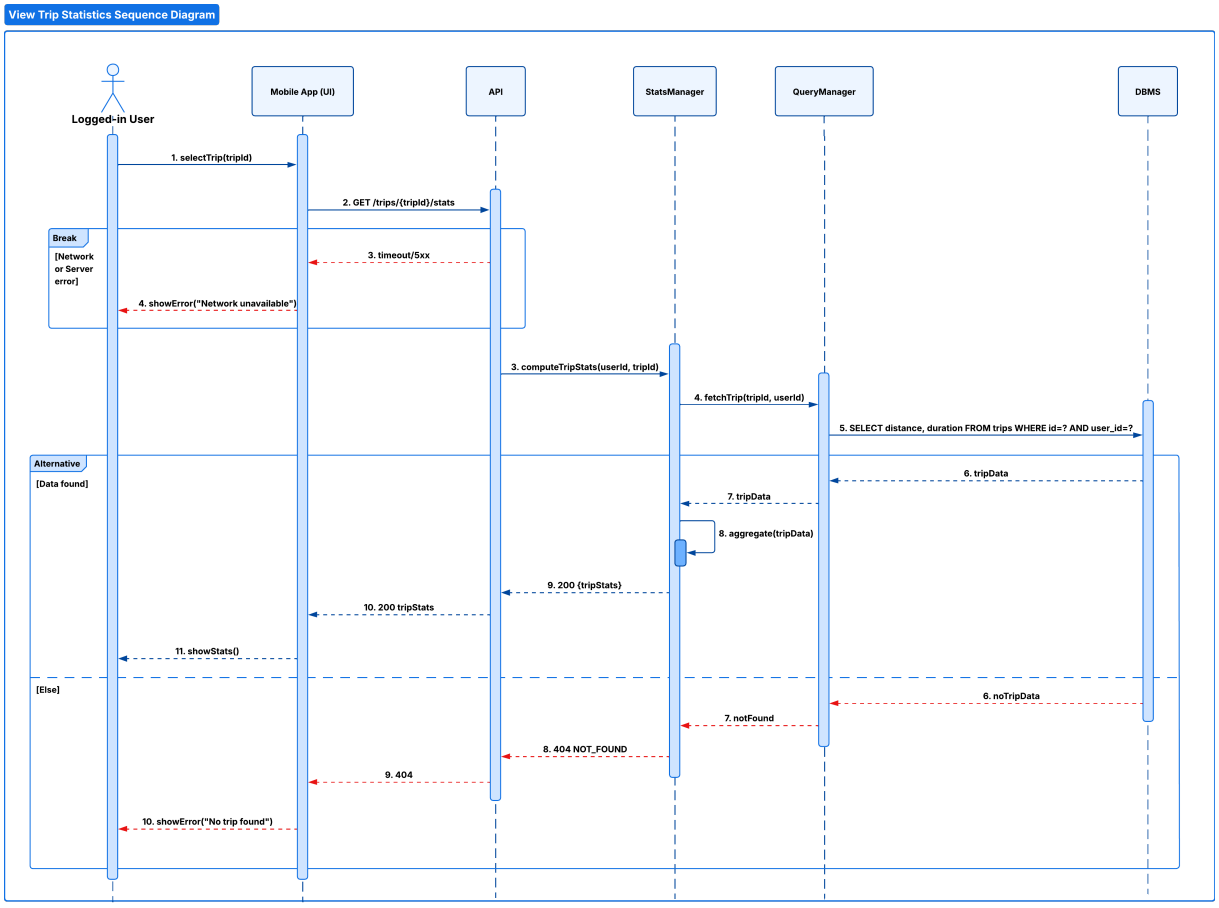

Figure 2.22: View Trip Statistics Sequence Diagram

[UC21] - Edit Personal Profile

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

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

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

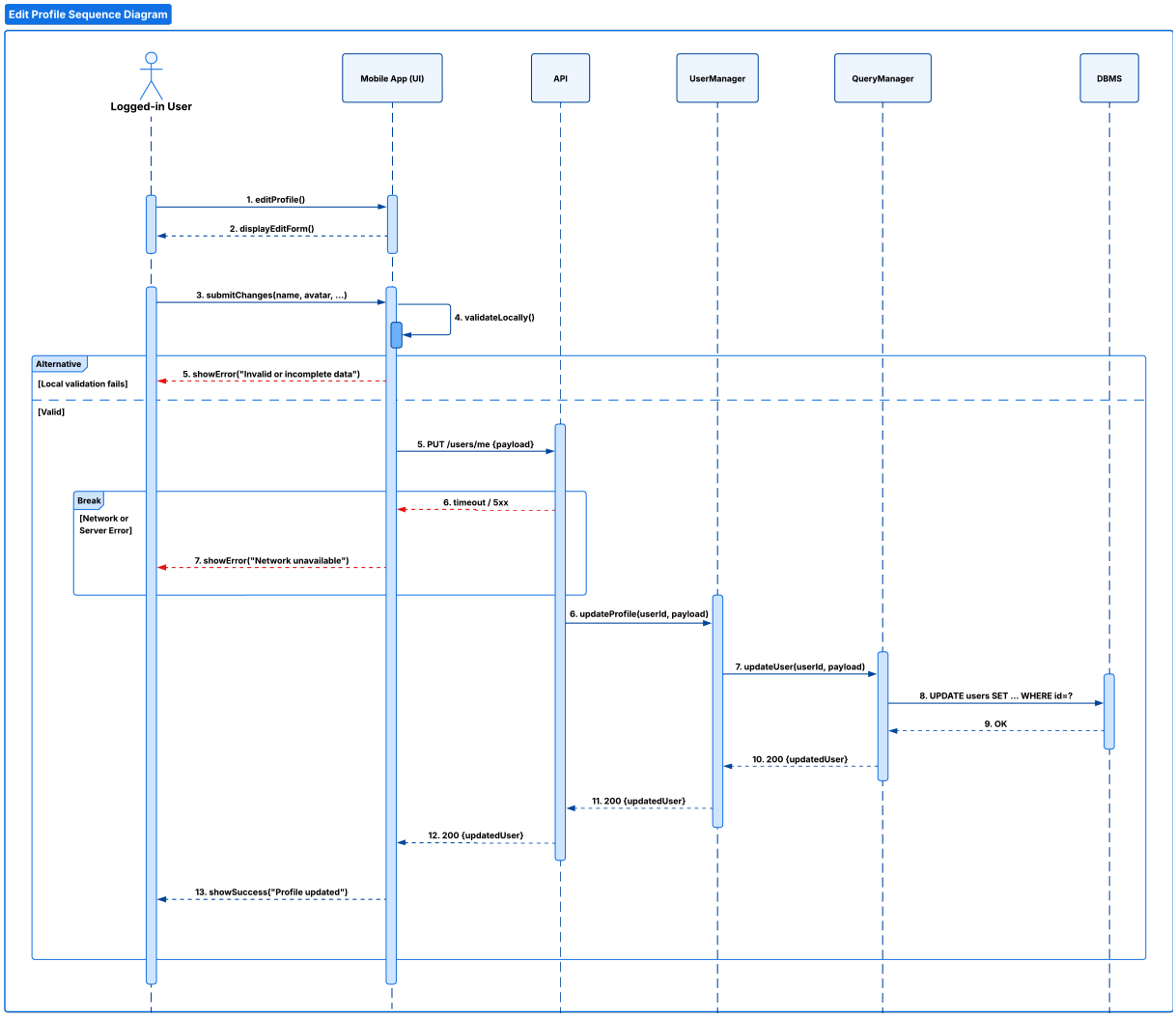

Figure 2.23: Edit Personal Profile Sequence Diagram

2.5. Component Interfaces

This section describes the operations exposed by each backend component and the interfaces through which these components collaborate at runtime.
Unless explicitly noted, all Manager methods operate in an authenticated context: the identifier of the currently logged-in user is implicitly supplied by the **API Gateway** after validating the access token, and is therefore not included in the method signatures. 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 Gateway** and relies on the **QueryManager** to retrieve and persist authentication metadata, including refresh tokens stored in the DBMS.
It exposes the following operations:

- **registerUser(email, password, profileData)**: validates registration data, checks whether the email is already in use, hashes the password, stores the user profile, persists a refresh token, and returns a newly issued access/refresh token pair.
- **loginUser(email, password)**: verifies the supplied credentials and, if valid, issues fresh access and refresh tokens and persists the new refresh token.
- **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 Gateway**. It uses the **QueryManager** for persistence and exposes:

- **updateProfile(userId, profilePayload)**: validates editable fields, prevents duplicate emails, applies the update to the stored profile, and returns the updated information.

TripManager

The **TripManager** coordinates the lifecycle of a cycling session. It receives GPS samples from the mobile app, stores trip metadata, and, upon trip completion, composes the final summary enriched with weather data supplied by the client after it retrieved a snapshot from the **WeatherManager**. All persistent data is saved through the **QueryManager**. The component exposes the following methods:

- **finalizeTrip(tripId, tripPayload)**: receives the final trip payload from the API (including the locally generated trip identifier), obtains a weather snapshot from the **WeatherManager**, composes the completed trip summary and stores it.
- **getTripsByUser(userId)**: returns the complete list of trips owned by the authenticated user, ordered by time.
- **getTripDetails(tripId, userId)**: verifies ownership of the requested trip and returns its full stored details.

PathManager

The **PathManager** implements all functionality related to bike paths: manual and automatic creation, route computation, ranking of candidate routes, path selection, deletion, and visibility updates. It is accessed through the **API Gateway** and uses the **QueryManager** to retrieve and update persistent path data.
The public operations are:

- **findPath(start, end, constraints)**: retrieves the required portion of the graph, including report-derived condition indicators for each segmen, and computes candidate routes satisfying the given constraints.
- **getPathDetails(pathId)**: returns metadata, geometry, condition indicators, and aggregated reports for the selected path and its segments.
- **createPath(pathPayload)**: validates metadata and segment lists (manual mode) or GPS-derived samples (automatic mode), persists the new path, and associates it with the creator.
- **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.

ReportManager

The **ReportManager** handles submission and confirmation of obstacle reports, both manual and automatic. It centralises validation, association to a path, and persistence. It also updates path-segment condition indicators in the DBMS that are consumed by the **PathManager** when computing rankings, without requiring additional calls to the ReportManager at query time.

- **createReport(reportPayload)**: processes manual reports, automatic sensor-based reports, and confirmations of existing reports. It validates payload and location data, inserts the report, and returns the newly created report identifier.

StatsManager

The **StatsManager** computes aggregated statistics over user trips and derives detailed metrics for individual trips. Raw and processed data are retrieved via the **QueryManager**.

- **computeOverallStats(userId)**: retrieves all trips owned by the user and computes cumulative indicators such as total distance, total duration, and average speed.
- **computeTripStats(userId, tripId)**: verifies ownership and computes per-trip statistics including average pace, speed, and time splits.

WeatherManager

The **WeatherManager** interfaces with an external weather provider and is used to enrich trip summaries with environmental context.

- **getWeather(location, timeWindow)**: queries the external weather API and returns a snapshot containing temperature, precipitation, and wind information for the specified interval.

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:

- **checkDuplicateEmail(email)**: verifies whether an email is already associated with an existing user.
- **insertUser(userData)**: inserts a new user record with hashed credentials and stores a refresh token issued for the user.
- **checkEmail(email)**: retrieves credential metadata for login attempts.
- **deleteRefreshToken(refreshToken)**: removes the refresh token associated with the terminated session.
- **fetchPaths(areaBounds)**: retrieves the relevant segments for route computation, including condition indicators already updated from reports.
- **fetchPathById(pathId)**: returns the full record of the specified path, including aggregated report data for its segments.
- **fetchReportsByPath(pathId)**: retrieves reports associated with the path segments to support condition scoring.
- **insertPath(pathPayload)**: stores new path metadata and geometry.
- **fetchUserPaths(userId)**: retrieves all paths created by the given user.
- **verifyOwnership(pathId, userId)**: checks whether the specified path belongs to the requesting user.
- **deletePathCascade(pathId)**: deletes a path and its dependent records.
- **updateVisibility(pathId, visibility)**: updates the visibility field of the selected path.
- **insertReport(reportPayload)**: persists manual or automatic reports and confirmations.
- **insertTripSummary(tripSummary)**: inserts a completed trip summary with its associated weather snapshot.
- **fetchTrips(userId)**: retrieves all trips belonging to the user, ordered by date.
- **fetchTrip(tripId, userId)**: retrieves detailed trip data, enforcing ownership.
- **updateUser(userId, profilePayload)**: stores changes to user profile fields.

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 Gateway with modular backend**. All external requests pass through a Gateway that 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 Gateway. 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 Gateway and the AuthManager. Boundary validation prevents malformed inputs and mitigates injection risks. Since authorisation depends solely on token claims, backend nodes remain stateless and interchangeable, matching the scalability goals of the architecture.

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 Gateway 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 Gateway) 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 Gateway.

- **Fault tolerance**. Calls to external services (e.g., weather provider) are wrapped with timeouts, fallbacks and default values. Even in case of partial failure, critical user actions (such as trip completion) are preserved and stored.

- **Data integrity**. The **QueryManager** encapsulates transactional boundaries to guarantee atomic updates during operations such as path creation, trip summary storage and report confirmation, isolating Managers from SQL concerns.

3 | User Interface Design

3.1. User Interfaces

3.1.1. Signup Page

3.1.2. Login Page

3.1.3. Home Page

3.1.4. Search Page

3.1.5. Profile Page

3.1.6. Settings Page

3.1.7. Statistics Page

3.1.8. Route Details Page

4 | Requirements Traceability

5 | Implementation, Integration and Test Plan

5.1. Overview

This chapter outlines the strategies adopted for the implementation, integration, and testing of the **Best Bike Paths (BBP)** platform. The primary objective is to ensure that the system meets the functional and non-functional requirements defined in the RASD document.
The chosen integration approach is **Bottom-Up**. This strategy involves developing and testing starting from low-level components (Data Layer and independent services) and then moving up towards more complex business logic components, finally reaching the presentation layer. This method allows for isolating and resolving bugs in the early stages, ensuring solid foundations for the higher-level modules.

5.2. Implementation Plan

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

5.2.1. Development Environment and Tools

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

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

5.2.2. Implementation Order

The implementation order of software components will be as follows:

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

5.3. Integration Plan

Component integration will strictly follow the Bottom-Up approach. Each phase involves integrating one or more modules into the existing subsystem, verifying their correct functioning through specific test drivers that simulate calls from higher levels.
The first step consists of integrating the `DBMS` with the `QueryManager`. Since all subsequent managers depend on data access, this subsystem constitutes the foundation of the architecture. The test driver will simulate query requests (CRUD) to verify the correct connection and manipulation of persistent data.

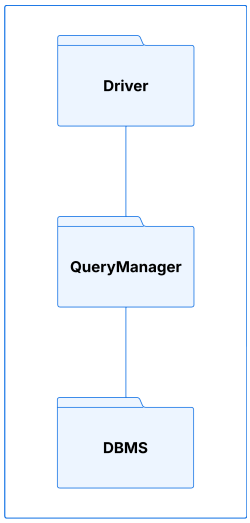

Figure 5.1: Step 1: DBMS and QueryManager Integration

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

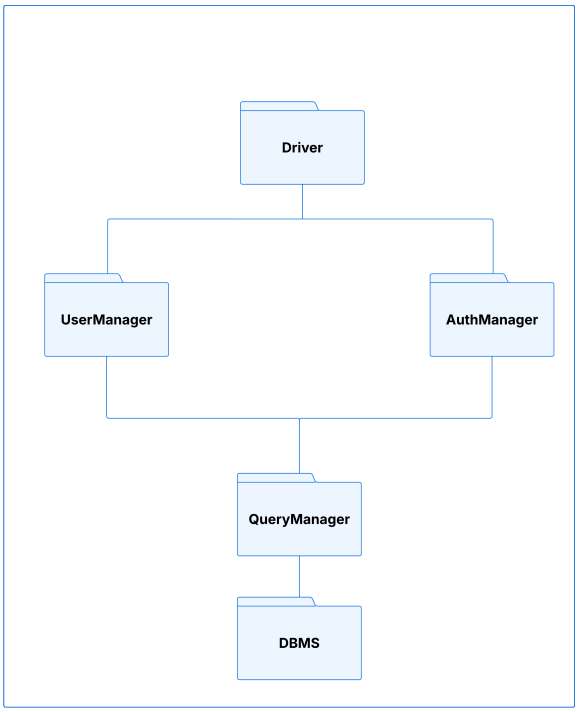

Figure 5.2: Step 2: AuthManager and UserManager Integration

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

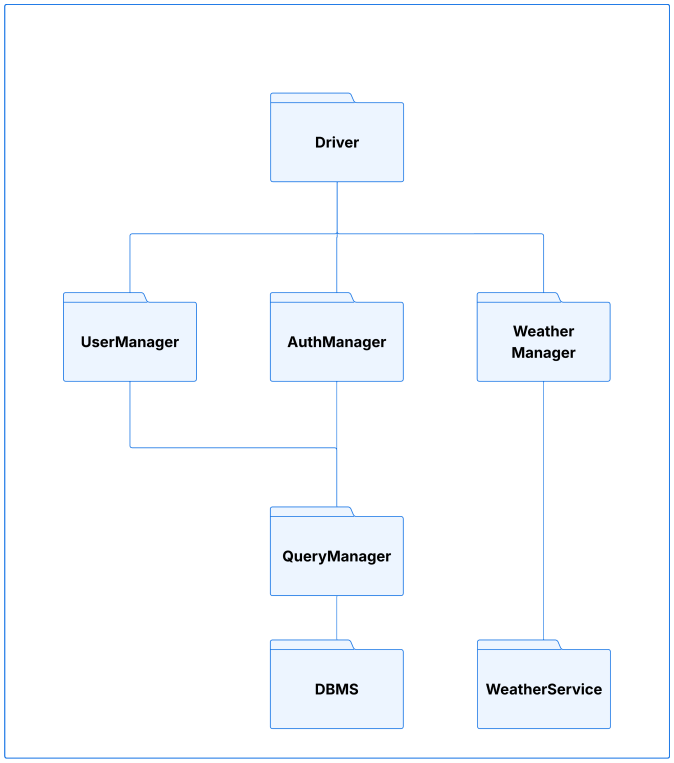

Figure 5.3: Step 3: WeatherManager Integration

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

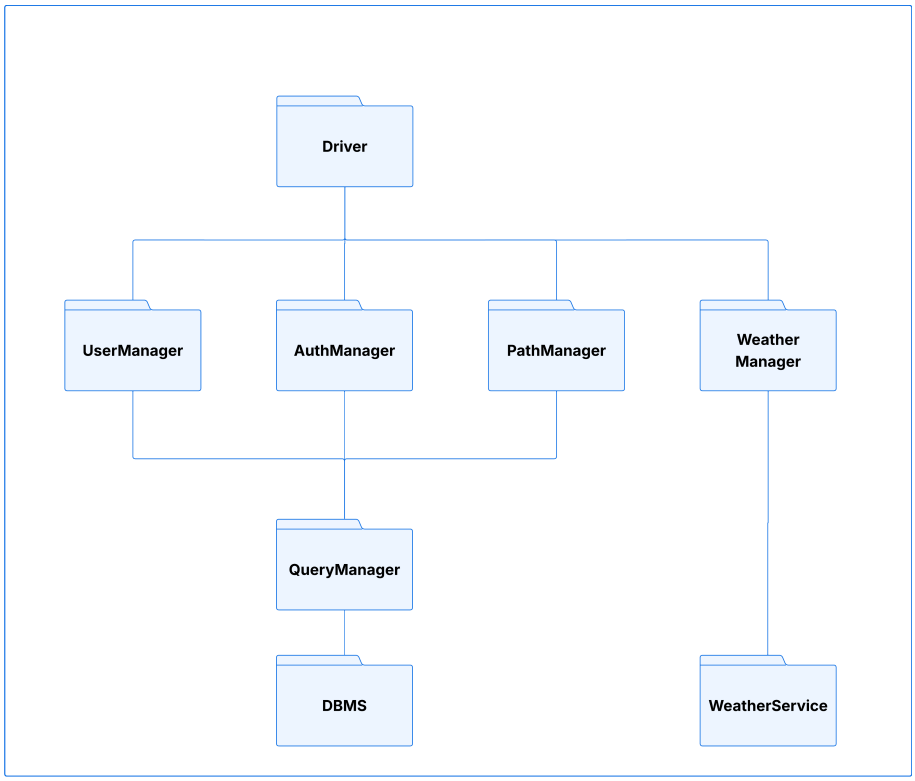

Figure 5.4: Step 4: PathManager Integration

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

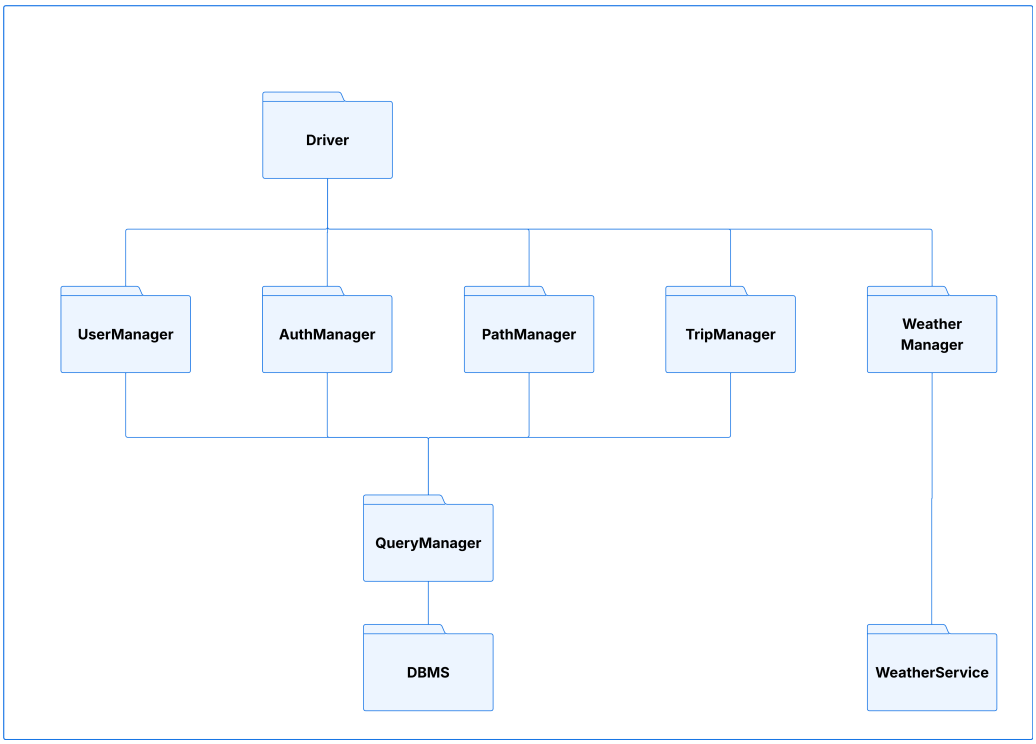

Figure 5.5: Step 5: TripManager Integration

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

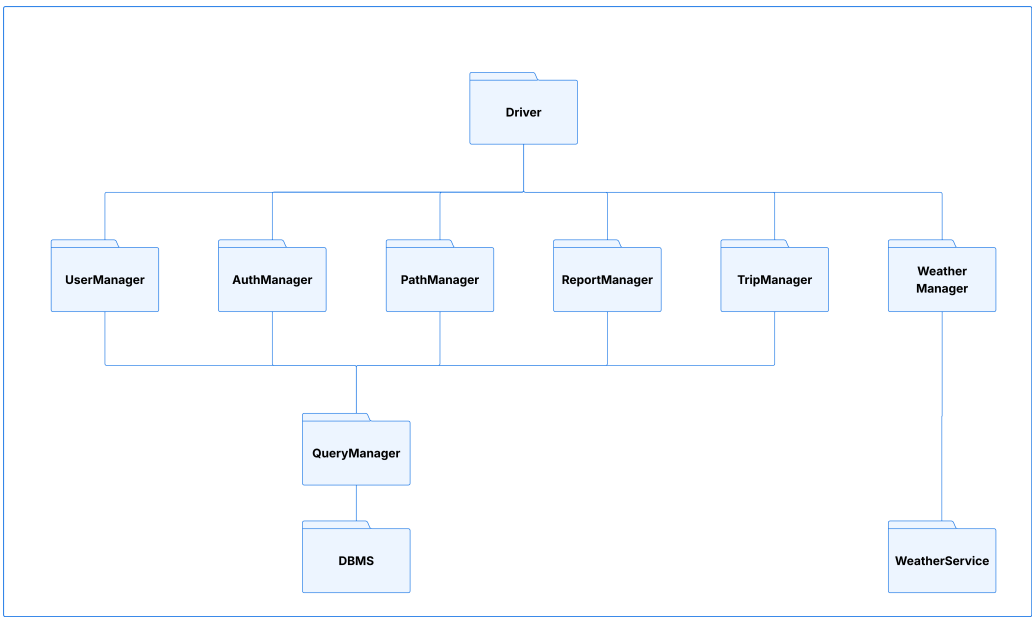

Figure 5.6: Step 6: ReportManager Integration

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

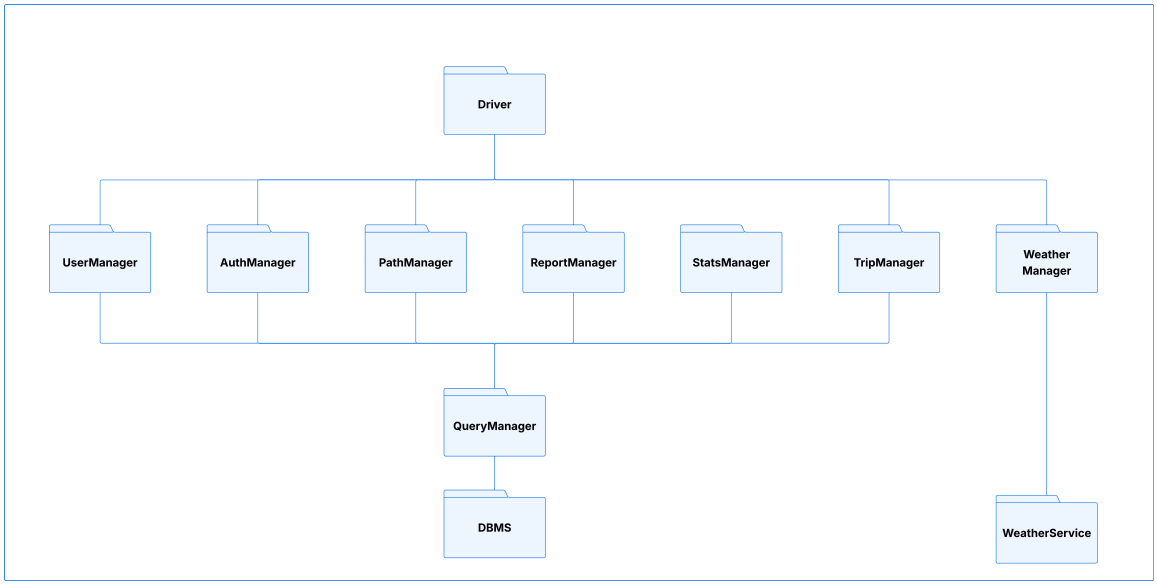

Figure 5.7: Step 7: StatsManager Integration (Complete Backend System)

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

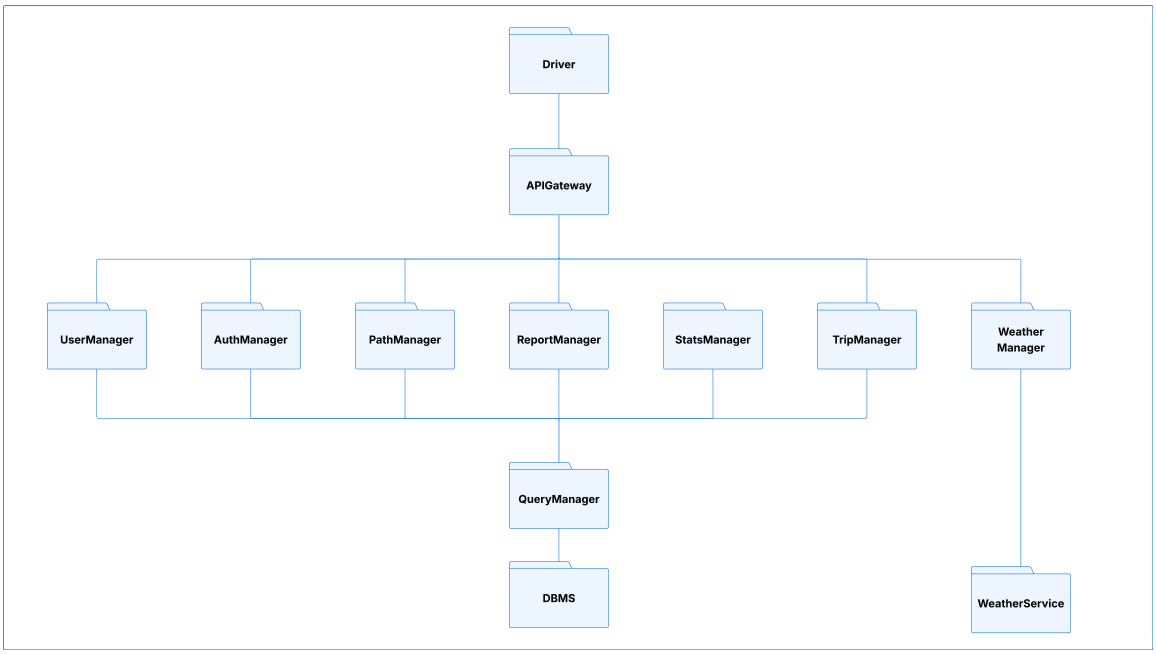

Figure 5.8: Step 8: API Gateway Integration

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

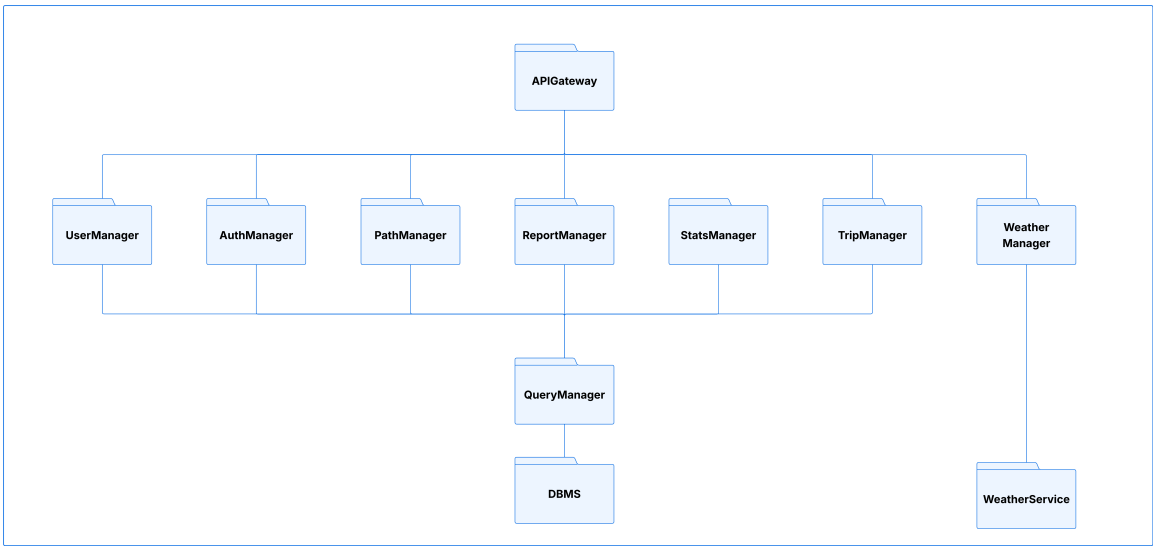

Figure 5.9: Step 9: Complete System Integration

5.4. Test Plan

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

5.4.1. Unit Testing

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

5.4.2. Integration Testing

Integration testing will be conducted incrementally following the steps described in Section 5.3. The primary objective of this phase is to verify that the interfaces between different components communicate correctly and that data flows seamlessly across module boundaries. We will focus on verifying the correct passing of parameters, the proper handling of exceptions that may propagate between modules, and the referential integrity of data as it moves from the business logic layer to the persistence layer. By testing these interactions early and often, we can identify interface mismatches and communication errors that unit tests might miss.

5.4.3. System Testing

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

and statistics retrieval, to guarantee a responsive user experience.

6 | Effort Spent

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

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

Table 6.1: Time spent on document preparation

Bibliography

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

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

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

List of Figures

List of Tables